Algorithms for Vehicle Navigation

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Sabine Storandt

aus Meiningen

Hauptberichter:	Prof. Dr. Stefan Funke
Mitberichter:	Prof. Dr. Hannah Bast
Tag der mündlichen Prüfung:	21.12.2012

Institut für Formale Methoden der Informatik
Universität Stuttgart

2012

Contents

Part I.

Prologue

Preface

Nowadays, navigation systems are integral parts of most cars. They allow the user to drive to a preselected destination on the shortest or quickest path by giving turn-by-turn directions. To fulfill this task the navigation system must be aware of the current position of the vehicle at any time, and has to compute the optimal route to the destination on that basis. Both of these subproblems have to be solved frequently, because the navigation system must react immediately if the vehicle leaves the precomputed route or the optimal path changes e.g. due to traffic jams. Therefore solving these tasks efficiently is crucial for safe and precise navigation.

To determine the vehicle's position in the street network, navigation systems are usually equipped with a GPS device. GPS signals are received from satellites, which continuously broadcast their position along with the actual time. With free sight to at least four satellites, the current position specified by latitude and longitude can be computed quite exactly using the received information. Unfortunately tall buildings, obstructing foliage, clouds, tunnels and other environmental factors can disturb or even block the GPS signal, leading to very imprecise measurements. Therefore accurate self-localization based only on GPS data cannot be assumed everywhere. In the first part of this thesis we will present an alternative localization scheme – called *path shapes* – which employs the movement pattern of the vehicle instead of absolute positions. Our new approach allows for consistently high localization quality in a fully autonomous manner. This result has been published at the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2011) [FS11] and received the best paper award.

In the context of optimal path computation this thesis focuses on finding good routes for Electric Vehicles (EVs). While being environmentally friendly, EVs suffer from some inconveniences compared to conventional cars: EVs are powered by a battery with a limited capacity, hence their cruising range is restricted (actually about 150 km). Of course, cruising ranges of conventional cars are also restricted by the petrol tank size, but gas station locations are so dense that the fuel status can typically be ignored when planning a trip. In contrast to that loading stations for EVs are still sparse in most countries, and moreover reloading takes considerably more time than refuelling gas (up to several hours). Therefore the energy-consumption along a path should be as low as possible for the EV to maintain mobility and to avoid unnecessary reloading stops. Unfortunately, the tools for computing conventional shortest or quickest paths cannot be directly applied to compute energy-efficient paths. One reason is that EVs can partly compensate their limitations by recuperating energy via regenerative breaking. So there might be a surplus of energy, e.g. on downhill paths. This leads to partly negative edge costs, prohibiting the use of Dijkstra's algorithm and related speed-up techniques. Moreover the EV is not allowed to run out of energy or to recuperate energy to a level that exceeds the battery's capacity. In the second part of this thesis we will describe algorithms that take care of the special challenges related to route planning for EVs. First we will introduce preprocessing techniques that allow for computing energy-optimal paths efficiently. These approaches have been published at the 25th Conference on Artificial Intelligence (AAAI 2011) [EFS11]. Moreover – if reloading cannot be avoided – we will present ways to decide where and how often to reload. The respective results have been published at the 26th Conference on Artificial Intelligence (AAAI 2012) [SF12]. Finally we will investigate more complex optimality criteria, balancing energy-consumption, reloading effort and time or distance. The respective algorithms have been published at the 22nd International Conference on Automated Planning and Scheduling (ICAPS 2012) [Sto12b] and the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS 2012) [Sto12a].

So the overall goal of this thesis is the provision of basic algorithmic building blocks for a navigation system that enables precise and autonomous self-localization on one hand and efficient route planning for the special requirements of Electric Vehicles on the other hand, see Figure 0.1 for a schematic illustration.

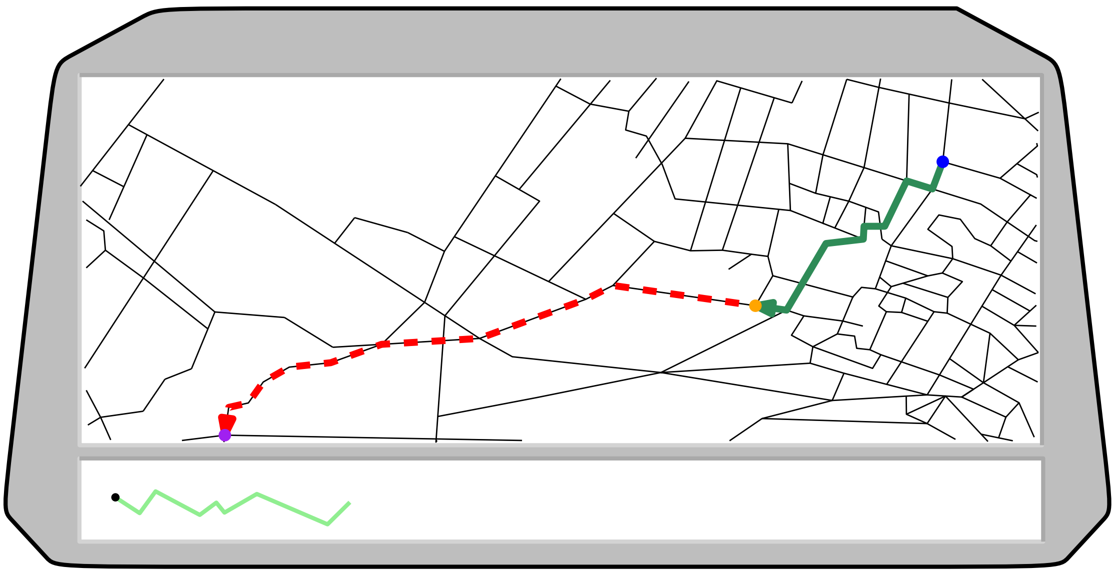


Figure 0.1.: Example of a navigation scenario: The source node is given in blue and the destination in purple. The green line in the lower part of the image is the shape of the trajectory driven so far. The matching path in the map is coloured dark green and reveals the actual position of the vehicle (yellow). The remaining path to the destination is indicated by the dashed red line, e.g. representing the most energy-efficient route.

Acknowledgement

I would like to thank my supervisor Stefan Funke for his great support. Moreover I am grateful to my colleague Jochen Eisner and all the people that were part of the FMI for the last three years.

1. Fundamental Concepts and Algorithms

In this chapter we introduce fundamental notations and definitions, and review algorithms and techniques that serve as basic building blocks for our map matching and routing approaches.

1.1. Graphs, Paths and Cycles

A graph G is a tuple consisting of a finite set of vertices (or nodes) V and a finite set of edges E. We refer to their quantities as $|V| = n$ and $|E| = m$. An edge $e \in E$ is represented by a pair of vertices $v, w \in V$. If the edge is directed (from v to w) the vertices are given as a tuple (v, w), otherwise as a set $\{v, w\}$. In a directed graph all edges are directed, such a graph is also called a digraph. We refer to the in-degree of a node $v \in V$ as $deg_{in}(v) := |\{(., v) \in E\}|$ and to the out-degree as $deg_{out} := |\{(v, .) \in E\}|$ respectively. The reverse graph $\overline{G}(V, \overline{E})$ of a directed graph $G = (V, E)$ can be derived from G by substituting each edge $(v, w) \in E$ by (w, v). If several edges occur that are described by the same tuple or set of nodes, the graph is called a multigraph. Moreover a multigraph can contain self-loops, i.e. $e = \{v, v\}\ v \in V$.

A path p is a sequence of edges $e_1, e_2, \cdots, e_k$ with $e_i = (v_i, w_i) \in E,\ i = 1, \cdots, k$ and $v_{i+1} = w_i,\ i = 1, \cdots, k-1$. A path is called simple if in the sequence of contained vertices $v_1, v_2, \cdots, v_k, w_k$ no node occurs twice. A path is called a cycle if $v_1 = w_k$. A graph without a cycle is called acyclic. For a directed acyclic graph the abbreviation DAG is used.

Moreover a graph can be augmented by a cost function c defined on the edges. Such a graph (V, E, c) is called a weighted graph. The weight of a path equals the summed costs of all contained edges, i.e. $c(p) = \sum_{e \in p} c(e)$.

1.2. Street Graphs

In a street graph $G(V, E)$ the edges represent streets and the nodes their intersection points. Edges in a street graph are directed to model one-way roads and allow for asymmetric edge weights. Street graphs typically exhibit the following characteristics:

1. Street graphs are almost planar.
2. Street graphs are sparse, i.e. $m \in \mathcal{O}(n)$.
3. The maximal node degree is small.
4. Street graphs reveal a hierarchical structure due to different types of streets from interstates, highways and freeways down to village streets.
5. Optimal paths are almost always unique.

These are important distinguishing features from other graphs to which routing algorithms are applied (e.g. uniform grid-graphs for robot navigation, public transportation networks or communication graphs). Of course, the last point depends on the choice of the edge costs and one can surely find (small) examples of street graphs which exhibit none of the mentioned characteristics. Nevertheless incorporating these assumptions when designing or engineering algorithms that operate on street graphs often leads to more efficient approaches.

1.3. Computing Shortest Paths

In a navigation scenario we typically aim for a path from the actual position $s \in V$ to a target $t \in V$ which exhibits minimal cost. The best approach to find this path crucially depends on the edge cost features. Edge costs are assigned depending on the application, either constant values $c : E \to \mathbb{R}^{(+)}$ (e.g. representing distance) or edge cost functions $c : E \to F = \{f : T \to \mathbb{R}^{(+)}\}$ (e.g. travel time dependent on departure time). Also, there might be multiple costs $c_1, \cdots, c_k$ with every c_i being either a constant or a function. We will discuss for each type of edge costs common algorithms that solve the respective routing problem.

1.3.1. Constant Edge Costs

Although constant costs must not necessarily reflect distances (but e.g. fuel costs), finding the minimal cost path from s to t in a graph $G(V, E, c)$ is called the shortest path problem (SPP). We refer to the optimal path as $\pi(s, t)$. Two basic approaches to solve the shortest path problem are the Bellman-Ford algorithm [Bel58, For62] and Dijkstra's algorithm [Dij59] that are described briefly in the following.

Bellman-Ford's Algorithm

As long as G contains no negative cycles, the shortest path is always simple and therefore any $\pi(s, t)$ contains at most $n - 1$ edges. The Bellman-Ford algorithm is based on this property.
The algorithm maintains for every node a (temporary) distance label $d()$ initialized with ∞. Then the label $d(s)$ is set to zero. In every round *all* edges get *relaxed*, i.e. for $e = (v, w) \in E$ the property $d(v) + c(e) < d(w)$ is checked and if possible $d(w)$ is updated to the new (smaller) distance value. Additionally a predecessor label is stored for each node. If relaxing an edge (v, w) leads to an update of $d(w)$, the predecessor label of w changes to v. After performing $n - 1$ rounds of edge relaxations the distance label of every node v reflects the minimal distance from s to v and the optimal path to t can be backtracked via the predecessor labels.
In case there are negative cycles in G there may be no shortest path, because there are paths with arbitrarily low costs (passing through the negative cycle repeatedly). The Bellman-Ford can be adapted to decide whether G contains such a cycle by performing an additional round of edge relaxation. There exists a negative cycle in G, iff the distance of *any* node changes in the n.th round.
The runtime for a single query is $\mathcal{O}(nm)$ as in each of the $n - 1$ rounds m edges are considered and edge relaxations can be performed in constant time (if distance and predecessor labels are stored e.g. in an array and a suitable graph representation is used). Observe that it is always sufficient to relax in every round only the edges incident to nodes that were updated in the last round (or adjacent to s in the first round). While this does not change the theoretical runtime it might significantly reduce the query time in practice.

Dijkstra's Algorithm

Unidirectional. Dijkstra's algorithm is also based on edge relaxation, but considers the nodes in a certain order to reduce the number of edge relaxations. For that purpose a priority queue Q is used that initially only contains a single tuple consisting of the source node and its distance value 0. All other node distances are initially set to ∞ as in the Bellman-Ford algorithm. Then in every round the node v with minimal distance label is extracted from Q. Hereupon all outgoing edges of v are relaxed. Whenever the distance label of an adjacent node w gets updated, this node along with the new distance is pushed into Q or in case w is already in Q the respective

distance label is decreased. The algorithm terminates as soon as Q runs empty.
For non-negative edge weights every node that is extracted from Q already has its minimal distance label. This is called the label-setting property and an extracted node is called *settled*, because no further distance updates can occur. As a consequence every node is extracted from Q at most once. Moreover every node v is pushed at most once into Q and the number of respective label decrease operations is bounded by $deg_{in}(v)$ because every edge $e = (u, v)$ only becomes relaxed once (when u gets settled). Therefore the total runtime of Dijkstra's algorithm on non-negative edge costs can be described as

$$\mathcal{O}(n \cdot T(push) + n \cdot T(ext) + m \cdot T(dec))$$

where T denotes the time needed for pushing a node into Q (push), extract the node with minimal label from Q (ext) or decrease a label of a node in Q (dec). Using Fibonacci heaps $T(push)$ and $T(ext)$ can be performed in amortized $\mathcal{O}(\log n)$ and $T(dec)$ amortized in $\mathcal{O}(1)$ [FT87]. Hence the total runtime equals $\mathcal{O}(n \log n + m)$, which is asymptotically the best known runtime for solving the shortest path problem in general graphs with non-negative edge weights.
So for graphs with non-negative edge costs Dijkstra's algorithm should be preferred over Bellman-Ford's algorithm as the asymptotic runtime is lower. Moreover Dijkstra's algorithm can be aborted when t is settled. This improves the practical runtime further, especially when the distance between source vertex s and the target t is small.
Dijkstra's algorithm can also be applied to graphs with negative edge costs. In this case it cannot be aborted as soon as t is popped out of Q, because the label-setting property does not hold anymore. Even worse applying Dijkstra's algorithm to graphs with negative edge costs might lead to exponential run time [Joh73]. Hence in such a scenario Bellman-Ford's algorithm is preferable.
Bidirectional. To decrease the search space and therefore the number of necessary operations, Dijkstra's algorithm can also be performed in a bidirectional manner. Here not only a run at s is started, but simultaneously another one at t in the reversed graph (they are also called forward and backward run). Hence every node v now has two labels $d_s(v)$ and $d_t(v)$ assigned to it. As soon as the sum of these labels is smaller than ∞ for a node, an upper bound on the weight of $\pi(s, t)$ is obtained. An additional variable δ keeps track of the minimal known upper bound during the bidirectional search process, i.e. any time an edge $e = (v, w)$ gets relaxed in the forward run δ is set to $\min(\delta, d_s(v) + c(e) + d_t(w))$ (simultaneously in the reverse run). Let d^s be the top label of the priority queue for s, and d^t the equivalent for t. Then the search can be aborted as soon as $d^s + d^t > \delta$ holds. This approach reduces the number of visited nodes during the search and is the basis for several speed-up techniques.

Johnson's Transformation

For graphs with non-negative edge costs only, there are more efficient approaches to solve the shortest path problem than the Bellman-Ford algorithm as described in the last subsection. Fortunately, it is possible to transform a graph $G(V, E, c)$ free of negative cycles into a graph $G'(V, E, c')$ with $c' \geq 0$ without changing the structure of shortest paths, i.e. $\pi_G(s, t) = \pi_{G'}(s, t)$. The transformed cost function can be found using Johnson's shifting technique [Joh77]. Here first a potential function on the nodes $\Phi : V \to \mathbb{R}$ is determined and then c' is derived from c for every edge $e = (v, w) \in E$ by $c'(e) := c(e) + \Phi(v) - \Phi(w)$. Of course the potentials have to be chosen appropriately to assure that $c'(e) \geq 0 \quad \forall e \in E$ and to maintain all shortest paths. In [Joh77] it was proven that the following approach leads to the desired result: First G is augmented by a dummy node x and edges (x, v) for all $v \in V$ with zero costs. Then Bellman-Ford's algorithm is performed starting at x. After termination the potential of a node

$v \in V$ is set to the computed distance from x to v, i.e. $\Phi(v) = d(v)$. As running Bellman-Ford's algorithm once takes time $\mathcal{O}(nm)$ and each edge cost can be transformed in constant time, G' can be derived from G in time $\mathcal{O}(nm)$.
Of course, the transformation is a preprocessing step and therefore has to be performed only *once*. All subsequent queries can then be answered in G' – there, no negative edge weights occur – with Dijkstra's algorithm in time $\mathcal{O}(n \log n + m)$.

Optimal Paths in DAGs

While for general (di)graphs the best theoretical known bound to solve the SPP is $\mathcal{O}(n \log n + m)$, in a DAG such queries can be answered in *linear* time. For that purpose the nodes are considered in a topological ordering, i.e. every node gets a label $l : V \to \{1, \cdots, n\}$ such that $\forall e = (v, w) \in E$ we have $l(v) < l(w)$, and then the nodes are ordered with respect to their labels (increasingly) and stored in a list $\mathcal{L}$. Because the graph is acyclic, such an ordering always exists. Moreover the computation of l and the resulting sequence of the nodes can be performed in linear time using topological sorting [CLRS90].
For given s and t the inequality $l(s) \leq l(t)$ must be satisfied, otherwise there is no path $\pi(s,t)$. To compute the optimal path again distance and predecessors labels are used, initialized as proposed for Bellman-Ford's and Dijkstra's algorithm. Here at first all outgoing edges of s get relaxed. The next vertex is chosen as the successor of s in $\mathcal{L}$ and then all its outgoing edges are relaxed. This process is repeated until the successor of a node equals the target t. At that point the optimal distance value for all nodes v with $l(v) \leq l(t)$ equals $d(v)$. Obviously the distance value of a node v gets updated at most $deg_{in}(v)$ times, hence the overall runtime is in $\mathcal{O}(n+m)$.
Of course, street graphs are normally far from being acyclic. But the construction of auxiliary graphs which obey this property is an important ingredient for certain map matching and routing algorithms and will be used later.

1.3.2. Speed-Up Techniques

Answering optimal queries efficiently is of great interest, not only for navigation systems to respond immediately if the vehicle deviates from the precomputed route or changes the desired destination, but also for a server system that is flooded with a large number of routing requests. Unfortunately the straightforward application of Dijkstra's algorithm is too slow for these tasks with a single run taking several seconds on the street network of Germany. Hence the search for speed-up techniques is an important challenge. One of the first approaches in this direction is the A* algorithm [HNR68]. It is a greedy and goal-directed approach, that uses lower bounds on path weights to find a better processing order of the nodes, decreasing the search space remarkably. A* does not rely on any preprocessing and hence is especially useful if edge costs might change over time (e.g. considering the influence of strong head/tailwind on fuel costs). On the other hand allowing a preprocessing step in which auxiliary data is computed and stored, the speed-up is often several orders of magnitudes larger than goal-directed search only. There are a wide range of such speed-up techniques available today, e.g. arc-flags [Lau97], landmarks [DW07], highway hierarchies [SS05], transit nodes [BFM+07], and contraction hierarchies [GSSD08], to name only a few. Several of these and other techniques can be combined to achieve even smaller query and/or preprocessing times, e.g. arc-flags augmented with contraction hierarchies (called SHARC), see [BD09]. We refer to [BDS+10, DSSW09] for a detailed overview.
For our applications, arc-flags and contraction hierarchies will play a major role. They are comparatively simple preprocessing techniques, therefore they can be adapted to other scenarios as well.

Arc-Flags

Arc-flags [Lau97] are based on the observation that the set of all shortest paths from one region to another (far away) region contains only very few edges. This follows from the fact that long drives to a certain direction require almost always the use of a specific highway or autobahn. Therefore the idea behind arc-flags is to identify this set of useful edges a priori and explore only these during query processing.
To assign arc-flags, the nodes are first partitioned $V = P_1 \uplus P_2 \uplus \cdots \uplus P_l$. Then a list of boolean flags is assigned to each edge, one flag for every partition P_i. This flag will be set true if the edge lies on a shortest path $\pi(v, t)$ with $v \in V$ and $t \in P_i$. While answering a query, we only have to consider edges marked true for the target's partition. Naively, we would have to compute the shortest path between any pair of nodes to determine all arc-flags. But even for small graphs this is too time-consuming to be practical. Using the observation that any path between two nodes v, w with $v \in P_i, w \in P_j, P_i \neq P_j$ has to go through at least one node on the boundary of P_i and P_j respectively, we can restrict ourselves to shortest paths between boundary nodes. Of course, now the flag for edges with their source and target in the same partition has to be set true for P_i in order to still allow queries to any node inside the partition.

Contraction Hierarchy

The contraction hierarchy (CH) technique was introduced in [GSSD08]. The basic idea is adding shortcut edges to the graph to speed up shortest path queries. To identify a useful set of shortcuts, the following approach is used: In a preprocessing phase an importance value is assigned to each node, and nodes are sorted in increasing order of their importance. Afterwards the nodes get removed/contracted one by one in that order, while preserving all shortest path distances among the remaining nodes by inserting additional edges (so called shortcuts). More precisely, an edge $e = (u, w)$ is added when contracting a node v if u and w are adjacent to v, and the only shortest path from u to w is uvw. The cost of e results from the chained costs of the edges (u, v) and (v, w). If the shortest path differs from uvw, then a so called witness path is found which testifies that the shortcut can be omitted. After all nodes have been removed, a new graph G' is constructed, consisting of all nodes and edges of the original graph and all shortcuts. An edge $e = (v, w)$ (original or shortcut) is called upwards if the importance of v is smaller than that of w and downwards otherwise. Moreover a path is called upwards (downwards), if it consists of upwards (downwards) edges only. In G' all shortest paths have the nice property of being the concatenation of an upward and a downward path. Hence all s-t-queries can be answered by a bidirectional Dijkstra computation, with the forward run (starting at s) considering only upward edges and the backward run (starting at t) considering exclusively downward edges. We call the respective sets of edges *the upward/downward graph induced by s/t*. This strategy prunes the respective search spaces dramatically and leads to a speed-up of more than two orders of magnitude for answering a query.

To decide in which order the nodes are contracted, several heuristics have been developed, evaluating the importance of a node. As one goal is to keep the resulting graph as sparse as possible, using the *edge difference* [GSSD08] is a popular measure. The edge difference of a node v is the actual number of shortcuts to be added minus the number of edges that can be removed when contracting v. Also, weighted versions of edge difference have been evaluated which penalize the addition of shortcuts even more, see [GKS10].

1.3.3. Edge Cost Functions

Edge cost functions are often used in the context of computing the path with minimal travel time. Because the traffic volume and therefore the possible travel speed varies over the day, the overall time to reach a certain destination differs dependent on the departure time, e.g. during rush hours it might be beneficial to use byroads instead of congested main roads and to avoid crowded junctions. This can be modelled by assigning cost functions to the edges that represent travel time dependent on the current time. An important characteristic of this functions is the FIFO property.

Definition 1.1 (FIFO-property)**.** *A function $f : \mathbb{R} \to \mathbb{R}$ satisfies the FIFO (first-in-first-out) property, if $\forall x \leq y$ we have $x + f(x) \leq y + f(y)$.*

In the context of street networks with time-dependent edge costs this means that one cannot arrive earlier at the end of a street segment when starting later. A typical counterexample for the FIFO-property occurs in rail journey planning. Here the same trip can be finished earlier when starting later with a faster train, e.g. an ICE instead of a local train.
If all edge cost functions obey the FIFO-property Bellman-Ford's algorithm can be applied similar to the constant case, but the edge costs have to be determined at the moment of edge relaxation because they are now dependent on $d(v)$ for an edge (v, w). The same holds for Dijkstra's algorithm, assuming all cost functions are non-negative. Therefore the theoretical runtime remains unchanged if the cost function can be evaluated in constant time.

Some speed-up techniques are also transferable to this scenario; A* search can be applied even for bidirectional query answering as described in [NDSL12]; in [Del08] it was shown how SHARC can also be adapted to edge cost functions, later [BDSV09] proposed a way to apply CH to such graphs as well. During the CH-graph creation a shortcut for the path u, v, w can only be omitted, if for *every* point in time there exists another path from u to w which is quicker than u, v, w. Hence the witness search is realized in [BDSV09] by a *profile Dijkstra* run which assigns functions to the nodes instead of constant labels. These functions represent minimal travel time profiles from the source node. To determine such a profile, the minimum of several functions has to be computed.

Definition 1.2 (Lower Hull)**.** *Given a set of functions $f_1, \cdots, f_k : D \to A$ we call $g : D \to A$ the lower hull/minimum function of $f_1, \cdots, f_k$ if $\forall x \in D : g(x) = \min\{f_1(x), \cdots, f_k(x)\}$ and denote it by $g = lh(f_1, \cdots, f_k)$.*

Computing the lower hull function can be very expensive dependent on the complexity of the given functions. Moreover if the respective shortcut (u, w) has to be inserted, its cost is derived by chaining the functions $c(u, v)$ and $c(v, w)$. This might result in a growth of complexity. For example consider two polynomials of degree k, chaining them leads to a polynomial with degree k^2. Similarly, chaining two step functions with different interval boundaries leads in the worst case to a combined function with about twice the number of steps. The same holds for the lower hull of such functions. Hence the space consumption and the evaluation time of a shortcut function increases with every contraction. In [BGNS10] several heuristics were presented that help to reduce preprocessing time and space consumption by an order of magnitude. As the memory requirements are still challenging, the authors also describe approaches for answering queries approximately.
Exact query answering also has to be adapted slightly as the backward search seems difficult with the arrival time at the destination being unknown a priori. Hence in [BDSV09] the backward search is replaced by marking all edges on downhill paths ending in t. Then the forward search considers uphill edges *and* marked ones and therefore can compute the optimal path in

an unidirectional manner. This approach can also be used in combination with other unidirectional speed-up techniques and for answering one-to-many queries as outlined in [EFH+11]. In [BGNS10] a more complicated interval search for the backward phase was proposed that operates with upper and lower bounds for the arrival time.

1.3.4. Multiple Edge Costs

In many routing applications the objective cannot be described sufficiently by only a single edge weight. For example, a driver is certainly willing to reach the destination as fast as possible, but he might be also interested in keeping the fuel costs small or in a certain budget. Often the metrics (e.g. travel time and distance) are somewhat conflicting, i.e. minimizing both values at the same time is impossible. Therefore one either aims for a fair trade-off of both values or wants the path minimizing one of the values but not exceeding a given bound on the other.
In the first case we typically ask for the minimal cost path for a linear combination of the edge weights. If these coefficients are known beforehand, the problem reduces to the single edge weight case and all available speed-up techniques for this scenario apply as well. If the linear combination is revealed at query time only, speed-up techniques using a preprocessing phase have to be modified. In [GKS10] the authors show how CH can be adapted to the scenario with two edge weights to achieve very fast query answering; but they allow only to choose one of the coefficients from a predefined discrete interval.
In the second case – minimizing one metric while putting limits on the other(s) – we ask for a so called pareto-optimal solution.

Definition 1.3. *Let $G(V, E, c)$ be a graph and c a vector of costs $c_1, \cdots, c_k : E \to \mathbb{R}$. For $v, w \in V$ we say a path $p = v, \cdots, w$ dominates another path $p' = v, \cdots, w$ if $\forall i : c_i(p) \leq c_i(p')$ (and at least one strict inequality holds). A path is called pareto-optimal if no dominating path exists, i.e. it is superior to all other paths with respect to at least one of the costs.*

Identifying a pareto-optimal solution with minimal costs c_1, which satisfies $c_i \leq R_i$ for given resource bounds $R_i, i = 2, \cdots, k$ is known as the constrained shortest path problem (CSP). Already for $i = 1$ this problem is NP-hard and therefore the existence of polytime algorithms for solution retrieval seems doubtful. In the context of two metrics (and hence a single resource bound) we call c_2 also the resource consumption of a path/edge, and denote it with r and the costs with $c = c_1$ for simplicity.
Due to its relevance for real-world applications, various approaches have been developed to solve the (two-dimensional) CSP problem exactly or with an approximation guarantee. CSP can be formulated as an ILP, giving rise to relaxations combined with gap closing algorithms. Moreover, path ranking and enumeration algorithms have been applied (see e.g. [MB09] and [MZ00] for a more detailed overview). Further common methods are the label setting and label correcting algorithms, as well as dynamic programming, which will be discussed briefly in the following.

Dynamic Programming

As proposed in [Jok66], the CSP problem can by solved with a dynamic programming (DP) approach: Let $c_{i,w}$ denote the minimal costs for an $s-w$-path with a resource consumption smaller or equal to i. It can be computed recursively using

$$c_{i,w} = \min\{c_{i-1,w}, \min_{e=(v,w)\in E}\{c_{i-r(e),v} + c(e)\}\}$$

and initial values $c_{0,s} = 0, c_{0,w} = \infty$ if $w \neq s$ and $c_{i,w} = \infty$ if $i < 0$. As we are interested in $c_{R,t}$, we have to store all $c_{i,w}, i = 0, \cdots, R, \quad w = 1, \cdots, n$ in the dynamic programming table, leading to a space consumption of $\mathcal{O}(nR)$ and a runtime of $\mathcal{O}(mR)$. Observe that this approach can only be applied if the resource consumption for all edges is really greater than zero. If this is not the case an alternative dynamic programming formulation has to be applied: Let $r_{i,w}$ be the minimal resource consumption of an $s-w$-path with costs equal to i. We can compute $r_{i,w}$ as follows:

$$r_{i,w} = \min_{e=(v,w)\in E}\{r_{i-c(e),v} + r(e)\}$$

The optimal solution is detremined by the minimal i with $r_{i,t} \leq R$ with i being minimal. Therefore the computation stops at $i = L_{OPT}$, leading to a to pseudopolynomial runtime of $\mathcal{O}(mL_{OPT})$. This formulation can be extended to a PTAS using scaling and rounding of the edge costs, see [Has92].

Labeling

The label setting (LS) method, introduced by [AAN82] in the context of minimum spanning trees with constraints, can be viewed as a variant of the DP approach, but has the advantage of not expanding dominated paths. LS assigns to each node v the list of all pareto-optimal tuples $(c(p), r(p))$ for an s-v-path p. This can be achieved by using an approach that adopts the idea of Dijkstra's algorithm for computing shortest paths. Here, we store labels in a priority queue (PQ). A label can be seen as triple consisting of a node ID, cost and resource consumption. The PQ sorts the labels in the increasing order of costs. We start with the PQ containing only the label $(s, 0, 0)$. In every round we extract the label with minimal cost and check for the respective node v if any of its outgoing edges $e = (v, w)$ leads to a new pareto-optimal solution (c, r) for w. If this is the case, we push (w, c, r) into the PQ. If (c, r) dominates any solution that was already assigned to w the dominated solution gets pruned.
Knowing all pareto-optimal solutions assigned to t after termination, the cheapest one that does not exceed the maximal allowed resource consumption R can easily be extracted. Of course if no labels whose resource value exceed R get pushed into the PQ, the search can be stopped when t is popped out the first time.
In the bidirectional version of the label setting computation (LSC) also a backwards search from t is performed simultaneously. Whenever the two search networks meet or a new label is assigned to a meeting node the best possible combination of an upward and a downward label gets selected, i.e. the one with minimal summed costs, that is not dominated by any other combination and the cumulated resource consumption does not exceed R. If the cost of the selected combination is lower than the previous cost bound C, a new upper bound on the cost of the optimal path is found. It can be used to prune the remaining search space further. As soon as both PQs become empty, the optimal path has been found. While LS often outperforms DP in practice, the theoretical runtime and space consumption are the same.

Speed-Up Techniques

As the described approaches are normally both time-and space consuming, several attempts have been made to reduce the search space. One of the first developed methods is known as *simple pruning*, which is based on assigning resource labels [AAN83]. Here the resource label r_{min} of a node v is the minimal resource consumption of an s-t-path that visits v. Obviously, all nodes with $r_{min}(v) > R$ can never be on a feasible path, and hence these nodes as well as their adjacent edges can be excluded a priori. Checking this condition for all nodes in G can be done very efficiently by running two conventional Dijkstra computations on the resource consumption starting in s, and t (on the reversed graph). Afterwards the two resulting labels $r_s(v)$ and $r_t(v)$

for each node get summed to receive $r_{min}(v)$. Nodes with a single label already exceeding R do not need to be pushed into the respective priority queue.
If the labels r_s and r_t are kept for all feasible nodes, this pruning method can also be extended to minimize the number of polls (extract min operations of the priority queue) during the label setting. Namely, a label (w, c, r) is only pushed into the PQ during the forward phase if $r + r_t(w) \leq R$ (analogously $r + r_s(w) \leq R$ in the backward run).
Note that if the allowed resource bound R is rather large or if there exist many edges with a resource consumption of zero, simple pruning might not eliminate enough nodes and edges to obtain a subgraph on which queries can be answered efficiently in practice.

In [MB09] the authors introduce the Aggressive Edge Elimination(AEE) procedure, which is based on Lagrangian relaxation techniques. With the help of AAE they could solve instances previously intractable, and tackle rather large networks (up to about 2.2 million nodes and 6.7 million edges). Note, that they do not focus on street networks, but on grid-like graphs in their experimental evaluation. The paper of [KMS05] explores the idea of adapting speed-up techniques for conventional shortest path computations to the CSP scenario. The authors show, that goal-directed search leads to a significant speed-up in street networks. In [DW09] the authors use a modified version of SHARC to speed up (also multicriteria) queries, but have to restrict themselves to a subset of all solutions in order to cope with space consumption.

1.4. Experimental Settings and Datasets

To validate our approaches and algorithms, we will present experimental results on a number of real-world instances. The basic characteristics of our test graphs are given in Table 9.2. All graphs are based on OpenStreetMap[1] and where necessary for the application they are augmented with SRTM height information [2].

Region	Abb.	n	m	avg. path length
Taunus	(TAU)	11220	24119	5.6km
Schwäbisch Hall	(SH)	100242	213096	27.6km
Massachusetts	(MA)	294345	731874	101.5km
Winnenden	(WIN)	500011	1074458	52.4km
Baden-Württemberg	(BW)	999591	2131490	79.6km
Southern Germany	(SG)	5588146	11711088	184.6km
California	(CAL)	11283833	22918849	523.9km
Germany	(GER)	15015877	30771648	452.5km
Japan	(JAP)	25970678	54141580	612.6km

Table 1.1.: Overview of the used test graphs. The average path length is based on the shortest paths of 10.000 randomly chosen pairs of source and target vertices.

Our implementations are written in C++, compiled with optimization level 3. Graphs are internally represented using offset arrays. Experiments were conducted on two different architectures: A server (AMD Opteron 6172 with 2.1 GHz and 96 GB RAM) for memory intensive preprocessing and for large test graphs and a laptop (Intel i3-2310M processor with 2.1 GHz and 8 GB RAM). We indicate for every experiment at the relevant point which hardware was used. Timings were always taken on a single core only.

[1] www.openstreetmap.org/
[2] www2.jpl.nasa.gov/srtm/

Part II.

Map Matching and Self-Localization

”You don’t know where you’re going until you know where you’ve been.”

(popular saying, presumably about map matching)

1. Motivation

Map matching is the problem of pinpointing an (imprecise) description of a trajectory to a concrete path in the map. This is not only of interest for self-localization of a mobile user, but also to estimate traffic flows on the basis of a family of such trajectories. Conventionally the path description consists of a sequence of fuzzy location measurements, e.g. received by GPS. The goal is to find the path in the map, that is most likely the one that led to these measurements. Note, that the accuracy and efficiency of map matching algorithms depend heavily on the density and precision of such measurements. Unfortunately GPS (and all other present-day localization schemes) cannot guarantee the availability of such high quality data everywhere. The basic question is whether we can employ other information to answer map matching queries, which do not depend on communication with third parties. Our key concept is to use the movement pattern – or in other words the shape of the path – to enable fully autonomous self-localization and accurate map matching in street networks (see Figure 1.1 for a schematic illustration of the concept). The models and results presented in the following have been published at GIS 2011 [FS11].

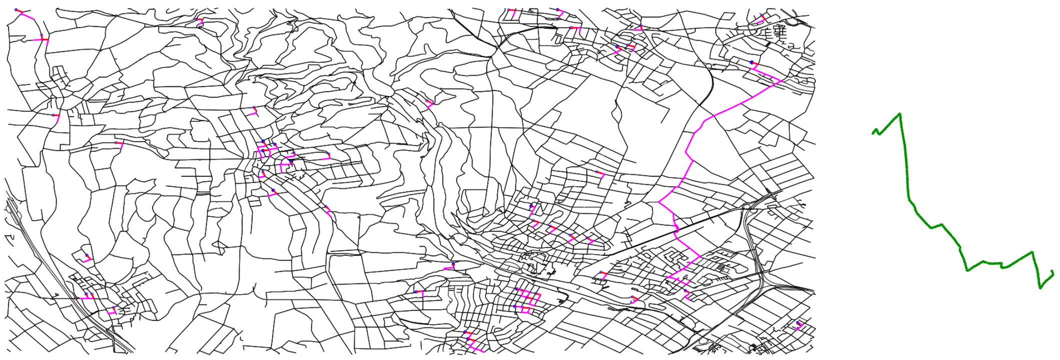


Figure 1.1.: Path shape (green) and its matches (longer than 200m) in the Taunus map for a fuzzy equality model. Blue circles indicate path origins, first edges are marked red.

1.1. Present-day Localization Schemes

Nowadays, there are plenty of ways to estimate the own position in the world. We briefly discuss the most common methods – GPS, GSM, and WPS – and outline their advantages and disadvantages in the context of vehicle navigation.

Global Positioning System (GPS). GPS is administered by the US government. It is based on a system of satellites, which continuously broadcast their orbit position and the current time. A GPS-receiver with free sight to at least four satellites can use this information to calculate its position on earth (in all three dimensions) very accurately. Moreover the imprecision of the signal is revealed as well, hence normally one can extract the latitude, longitude (and height), and an error radius around this position. Therefore the input for performing map matching is a sequence of measurements consisting of disks (or balls), that the trajectory must visit in the given order.

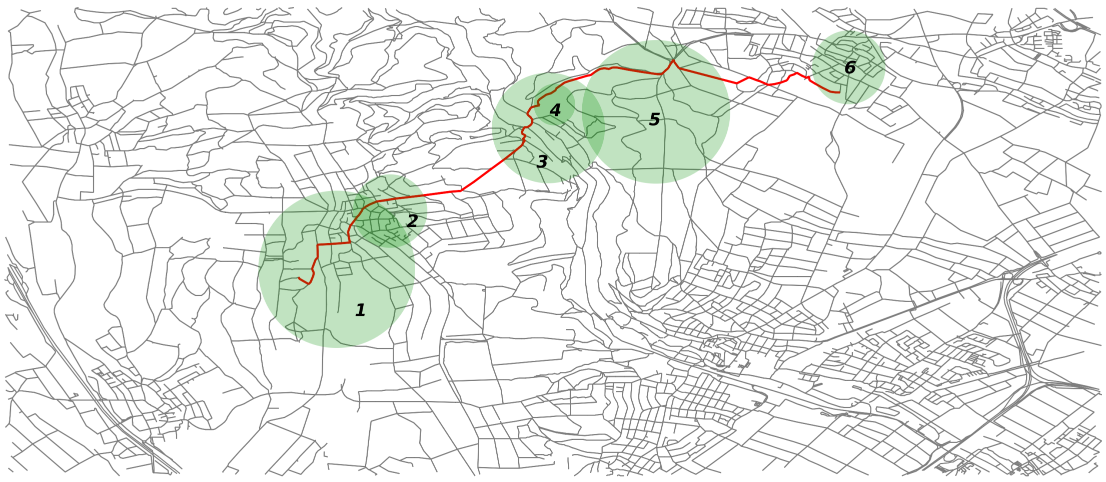


Figure 1.2.: Examples for measurements derived from GPS signals.

The number of GPS-receivers increased dramatically in the last decade, because apart from navigation systems more and more mobile phones are equipped with such a device. Most GPS units allow for a rather precise localization, with an error only up to 10 meters. But signal reception can be blocked or disturbed, e.g. when driving through a tunnel or due to high buildings, terrain characteristics or obstructing foliage. Therefore the positional error might be larger – or there might be no positional information at all.

Global System for Mobile Communication (GSM). GSM is a digital cellular system. Mobile phones connect to it by searching for a a nearby cell, where each cell corresponds to the coverage area of a base station antenna. Thereby the radius of the cell can vary from some meters to several kilometres dependent on the height and the location of the antenna. In order to use GSM for self-localization, the user does not only have to carry a mobile phone, but also the exact positions of the contacted base stations need to be available. Databases are provided e.g. by Google's MyLocation[1]. While moving along some trajectory the sequence of accessible base stations gets stored. If there are several ones at the same time, ties are broken by considering the received signal strength (RSS). So the input for a map matching query is again a sequence of locations together with an imprecision radius for each (now corresponding to the coverage area of the respective antenna). In general GPS provides more accurate data than GSM, because the error radius is smaller and the coverage of GSM in sparsely inhabited areas is often insufficient. Nevertheless so called picocells allow for using GSM for indoor navigation, while GPS performs poorly in such a scenario.

Wi-Fi-Based Positioning System (WPS). WPS uses geocoded wireless access points to provide positional information. The approach is similar to GSM: Now a device with wifi capability must be at hand. On that basis the sequence of wifi networks can be extracted, that the user could log-in to while moving around. The required positional data can be obtained from several companies, e.g. Skyhook[2]. Again this system is predestined for use in urban areas and therefore is not suitable as a stand-alone solution for navigating vehicles in road networks. But it supports other applications like indoor navigation and building classification.

[1]http://www.google.com/mobile/maps/
[2]http://www.skyhookwireless.com/

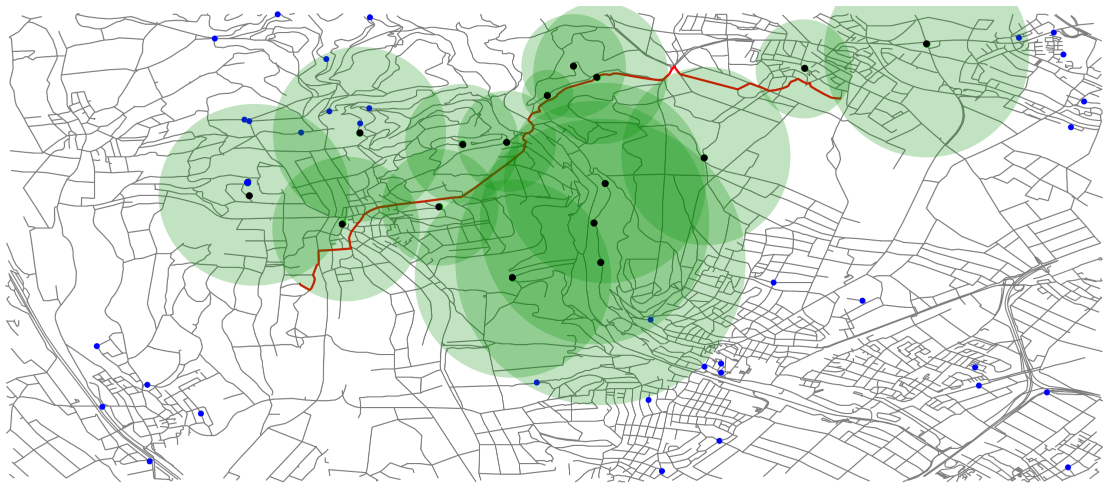


Figure 1.3.: Examples for measurements derived from GSM/WPS signals. Base station antennas are indicated by blue dots, visited ones by black dots.

1.2. Related Work

Map matching is well-studied in different variations, see [QON07] for an overview. Basically we are confronted with the following problem: Given a graph $G(V, E)$ augmented with a cost function $c : E \to \mathbb{R}_0^+$ and a sequence of location measurements $m_1, \cdots, m_k$; each measurement m_i describes a disk D_i, represented by a centre point c_i and an error radius r_i. The goal is to find a path in G which connects m_1 and m_k and visits all disks in the given order. As there is naturally a variety of paths fulfilling that condition, we aim for the one with optimal score. The score could be e.g. the objective function value of an integer program, like described in [Yan10], or favour simply the shortest one, see [EFH+11]:

Layered Graph Map Matching. In [EFH+11] an approach is described which is based on the implicit construction of a layered graph between consecutive disks. Let V_i be the set of nodes inside disk D_i. All nodes $v \in V_1$ get initialized with a distance label of $d(v) = 0$, as they are all potential starting points. Then the first disk is connected to the second one by running Dijkstra from V_1 until all node in V_2 are settled. Afterwards a second Dijkstra run is started at the nodes in V_2, keeping their actual distance labels from the last run as initialization. Again Dijkstra runs until all nodes in V_3 are settled. This approach is continued until the first node in the last disk D_k is settled. At that point the shortest path is found that visits the disks in the given order. This path can be backtracked piecewise by following the predecessor labels of the single Dijkstra computations between subsequent disks. The authors show that this method allows for very accurate reconstruction of the correct trajectory based on GPS or GSM data if the number of measurements is large enough.

Fingerprinting. To compensate the large error radius in GSM and WPS measurements the RSS might be incorporated as additional input. The technique taking advantage of RSS is called fingerprinting. Here the RSS signature – the fingerprint – of different locations inside a cell is stored in a database. During query answering the (k) locations with their fingerprint being closest to the actual RSS are extracted and the current position gets estimated on that basis. Deterministic variants e.g. proposed in [VCdL+06] use the average of the fingerprint positions for that purpose. Probabilistic methods (see [IY10]) use fingerprints for small areas instead of single locations and calculate the probability of a certain RSS inside a fingerprint region on query time. Augmented fingerprints, containing local attributes like sound and light, help to recognize logical locations as described in [ACRC09].

Note, that all these papers are based on absolute location information. But as indicated before, our idea will be to make use of the shape of a path for map matching purposes.
Inertial Navigation. In other navigation domains – where vehicles can move almost freely e.g. ships, planes, missiles or robots[BDW95] – navigation based on the movement trajectory is already an established approach known as inertial navigation system (INS). Here normally a starting position has to be known and the current position is calculated based on speed and direction of the moving object as well as the elapsed time since the departure (also known as dead reckoning). Note, that using INS already small measurement errors translate into large positional errors and moreover cumulate over time, as the new position is always computed on the basis of the last one. Hence in regular intervals the actual position has to be corrected using e.g. GPS; therefore the autonomy of the system is compromised. In contrast to ship and plane navigation a vehicle cannot move unhindered around but has to stick to streets, therefore the effect of measurement errors might be mitigated in our scenario. Even better we gain information while driving around, hence the positioning might become more and more accurate over time – in contrast to the INS paradigm.

Curve Matching. The closest approach to our application can be found in the area of curve matching. Here the goal is to compare shapes in order to be able to pick the 'most similar' one out of a set of shapes for a given reference. Thereby the notion of similarity depends on the application and the kinds of possible shapes. Curve matching is commonly used e.g. for handwriting and object recognition. Transferred to our scenario we are given a collection of polygonal curves (represented by the underlying graph) and want to select the one, that matches our reference curve (the given trajectory) best. A reasonable similarity measure might be the Frechet-Distance which was already applied to planar maps in [AERW03]. Another possibility is the application of the fast marching method as presented in [FB03]. The Direction-Based Frechet-Distance [dBCI11] and the metric presented in [ACH+90] also allow for partial matchings which might be very helpful in our envisioned scenario. But all of these methods require integral calculus and have a runtime of $\Omega(ab)$ with a and b denoting the number of vertices on the reference curve and the graph respectively. Hence they are not suitable for comparing a single curve to a large set of curves efficiently.

1.3. Outline

We will first motivate our new map matching concept called *path shapes* and describe very basic features like how to gather data, and how to represent and compare such shapes (see Chapter 2). Moreover we will introduce a variant of Dijkstra's algorithm which takes shapes into account. We will conclude these preliminaries with a number of experiments that prove the soundness of the concept, i.e. we show that path shapes provide sufficient information for accurate self-localization and map matching. The only drawback revealed by the experiments is the very large runtime for the proposed naive approach. Therefore in Chapter 3 we will introduce a preprocessing technique which interleaves established data structures for text search (namely generalized suffix trees) and Dijkstra's algorithm. On that basis we performed in Chapter 4 new experiments where path shapes are transformed into words. This results in a speed up of several orders of magnitude. We conclude the map matching topic with a discussion of our results as well as some ideas for future directions and application domains (see Chapter 5).

2. Path Shapes - An Alternative Localization Scheme

In a navigation scenario the vehicle's position in a mere spatial sense (specified by latitude and longitude) is not really of interest – but the actual street and the closest node in the underlying street network provide a sound basis to compute the optimal path to the destination from the current location. The previously described localization schemes take this into account by mapping the nodes to their coordinates and using proper data structures to retrieve the set of nodes and edges in a nearby area efficiently. Note, that in a verbal path description, like *'drive 100m straight, then turn left, drive until the next junction and turn sharp right, ...'*, this indirection over absolute positions is not necessary. Instead a sequence of relative movements is given, which allows someone with knowledge about the starting point and the network to reconstruct the travelled route accurately. Our key idea is to extend this concept and use exclusively *the shape of the driven path* to identify it in the map. Observe, that such a path description is naturally induced by the structure of the street network. Moreover the retrieval of path shapes requires only devices, which operate *autonomously*. Hence, self-localization is no longer dependent on communication with third parties. Therefore we can expect consistent data quality, no matter if the vehicle drives through a tunnel, between tall buildings or in a rural area. Moreover autonomy maintains the privacy of the mobile user. In contrast to that the usage of GSM and WPS reveals the own (imprecise) position at the moment the location of a certain base station is requested.
Exploiting the structure of paths in a street network, we show that path shapes can serve as stand-alone scheme for accurate map matching and self-localization. We focus on shortest paths, even though in principle we could consider all paths. This restriction seems to be sound as users tend to move on at least piecewise shortest paths. If those pieces are long enough to make the respective path shape unique in the network, restricting to shortest paths is fine.

2.1. Path Shape Extraction

A good representation of path shapes and respective map matching approaches depend on the availability and precision of input data. But where can we expect to obtain shape data from?
Most of the current cars are equipped with ESC (electronic stability control), which monitors lateral acceleration, vehicle rotation and individual wheel speeds. The very same data can be extracted via vendor specific protocols of the CAN-bus (a standard communications network in vehicles). These information can be used to determine relative turning angles, while the odometer of the car provides distance measurements. So a built-in navigations system can extract path shapes on the basis of such data. Also, modern cars with a complex on-board computer might have an incorporated electronic compass providing absolute turning angles.
Observe that all the devices needed to extract the path shape are fully autonomous, hence no communication with third parties is required. But as a consequence a path shape contains neither any absolute positional information nor an absolute direction and – being the result of a physical measurement – we cannot expect it to be precise and in particular to match exactly the path shapes present in the road network. So the basic questions are if an imprecise path shape can be identified uniquely in the network and if so how long it has to be to achieve uniqueness. To

answer these questions we first have to come up with reasonable path shape representations and robust ways to compare such shapes.

2.2. Modelling Paths as Shapes

As a first step we have to make paths in the street network comparable to shapes extracted by the car's navigation system. The path embedded in the network can always be seen as the ground truth and on the basis of the absolute coordinates of the vertices its shape can easily be obtained. It is very unlikely, though, that this ground truth path can be constructed exactly from (typically imprecise) measurements of any kind. The polylines that are reconstructed from measurements could for example look like the bold red polyline in Figure 2.1, centre, or the bold blue line in Figure 2.1, right. In the polyline in the centre, the global shape of the polyline is still preserved whereas in the right polyline the global shape is destroyed even though locally the sequence of turns and straight line sections look very similar to ground truth. It depends on the type of measurement error which of the two notions of shape similarity are more appropriate. Our methodology developed in the following will deal with both. The notion of similarity as implied by the red polyline in the centre of Figure 2.1 is pretty much what people have looked at in the area of *curve matching* according to Frechet distance, for example. The notion of similarity implied by the blue polyline on the right is closer to what we expect from the availability of relative turning angles only, where angle measurement errors can accumulate over time.
In the following we will show how to model and compare polylines such that similarity – be it global or local – can be detected. It is obvious that for short paths the notions of global or local similarity approach each other. This will also be reflected in our experimental results later on.
Observe, that the raw shape data of the trajectory alternates between distances and angles, e.g. $100m$ *straight,* $30°$ *left turn,* $37m$ *straight,* $49°$ *right turn.* This appears to be unhandy for imprecision-tolerant comparison, therefore we simplify the representation by sampling the shape uniformly. For that purpose we cut the shape into segments of 1m each. This allows us to represent the shape as sequence of angels only – one angle for each segment. According to the different notions of path shapes, we propose two different representations that reflect the kinds of similarity. If the shape should be preserved globally, we always consider the angle difference towards the very first driven segment. If we can only hope for local matches, we use the relative angle towards the previous segment instead. Formalized this leads us to the following definition:

Definition 2.1 (GAR/LAR). *Given a path shape S of length l(in meters) and let $s_1, \cdots, s_l$ be the respective segments of S derived by uniform resampling. Then S can be represented as sequence of angles $a_1, \cdots, a_l$ using the*

- *Global Angle Representation (GAR):* $a_i = \angle s_i s_1, i = 1, \cdots, l$
- *Local Angle Representation (LAR):* $a_1 = 0$, $a_i = \angle s_i s_{i-1}, i = 2, \cdots, l$

with $a_i \in \{-179, \cdots, 180\}$ *for all* $i = 1, \cdots, l$.

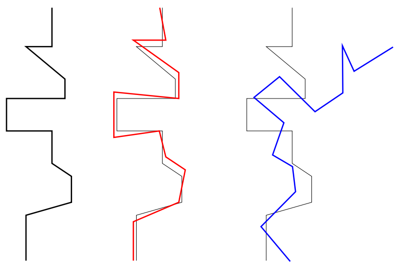


Figure 2.1.: Global vs. local match: Path from the map (black, left), globally similar shape (red, centre) and locally similar shape (blue, right). The orientation of the red and blue paths are not known, but had to be fixed for visualisation purposes only.

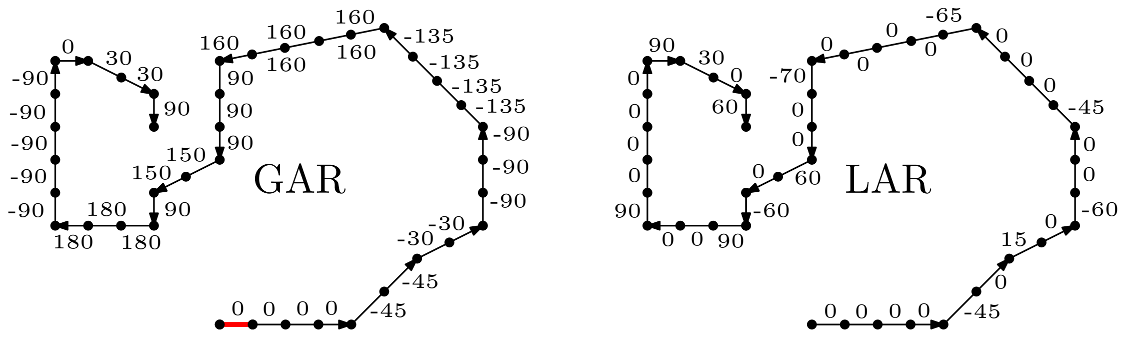


Figure 2.2.: The same trajectory represented by GAR or LAR. The black marks indicate the resampling of the shape to pieces of 1m. Note that the representations do not change if the shape were rotated.

An example for these representations can be found in Figure 2.2. Observe, that we are normally not aware of the orientation of the shape then using GAR or LAR. Hence we aim for a match in the map with translations *and* rotations being allowed. GAR and LAR differ from each other as soon as imprecisions play a role. GAR reflects more the basic intention of a curve matching, while under LAR we might declare two path shapes similar even if they do not look similar at first sight on a global scale.
Note, that for any path in the street map – where the precise coordinates of nodes are known – the representation according GAR or LAR can easily be obtained. But as outlined before we cannot expect to find an exact match for the input shape in the map. Therefore reasonable comparison methods should allow the shapes to differ to a certain extent without declaring them unequal.

2.3. Robust Comparison of Path Shapes

A naive way of comparing two path shapes S, S' represented as a sequence of angles $a_1, \cdots, a_l$ and $b_1, \cdots, b_k$ is to check whether $l = k$ and $a_i = b_i, \forall i = 1, \cdots, l$. Of course, this will rarely lead to a match even if very precise measurement devices are at hand.
A first approach to soften the equality condition is to use an *angle tolerance* t_α and declare paths with representations $a_1, a_2, \ldots, a_l$ and $b_1, b_2, \ldots, b_{l=k}$ equal if $|a_i - b_i| \leq t_\alpha$. Still this condition seems too harsh. In particular for LAR or GAR, if e.g. due to different resampling, a sharp bend shows up at the i-th position in one representation of the same path whereas at position $i+1$ in the other representation, see Figure 2.3. This problem of different resampling or different representation length can be tackled by allowing a *wobbling comparison*, see Definition 2.2.

Definition 2.2 (Wobbling Comparison). *Two path shapes $a_1, \cdots, a_l$ and $b_1, \cdots, b_k$ are declared equal for a given range parameter $w \in \mathbb{N}$, if there there exists a function $\phi : \{1, \ldots l\} \to \{1, \ldots k\}$ with $\phi(i+1) \geq \phi(i)$ for $i = 1, \ldots l-1$, $\phi(i) \in [\max(0, i-w), \min(i+w, k)]$ and $|a_i - b_{\phi(i)}| \leq t_a$ for some angle tolerance t_a.*

The existence of a function ϕ as described in Definition 2.2 can be easily checked by a left-to-right sweep over the two path representations. Note that this also takes care of systematic under- or overestimation of distances in the measurements, so the function ϕ could also be thought of a reauging of the odometer.
Angle tolerances and wobbling do not get rid of the problem that a 90° turn might show up in one representation as a sequence of nine 10° turns whereas in the other as one single 90° turn, though. This is a quite natural problem which might not even be induced by measurement errors but by different ways drivers make a 90° turn, see Figure 2.4. Similarly, unexpected obstacles

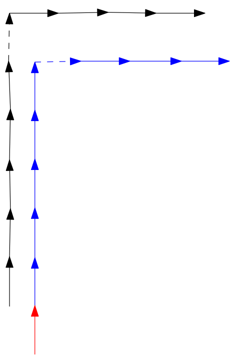


Figure 2.3.: With the red edge the paths are equal. Without it the two dashed sections get compared and the paths are declared unequal unless $t_a \geq 90$.

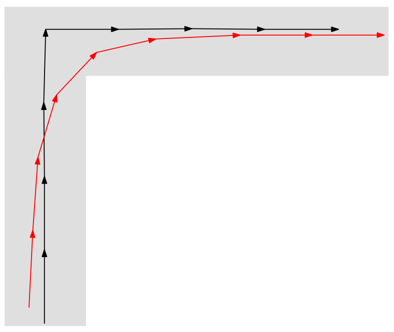


Figure 2.4.: Black: Curve derived from the map. Red: The same curve reconstructed by measurements.

like potholes might incur artefacts in the path shape that prevent a proper matching. Depending on the representation we will have to come up with different remedies.
In GAR, we want to allow small sections of the paths to differ if the global similarity is preserved. Therefore we introduce a range-based comparison (RBC).

Definition 2.3 (RBC)**.** *Two path shapes are declared equal for a given range $r \in \mathbb{N}$ and a percentage $c \in [0, 1]$, if for every section of the shape of length r at least $c \cdot r$ angles are equal to the other shape's corresponding section according to one of the previous comparison approaches (exact* or *with tolerances* or *with wobbling).*

Unfortunately, RBC only applies for GAR, since LAR is based on local deviations as main representation.
Instead of allowing $(1 - c)r$ angles in each section of length r to disagree, in case of *LAR*, we introduce a moving average over the angles, so called *sum-based comparison (SBC)*.

Definition 2.4 (SBC)**.** *Two path shapes are declared equal if they can be divided in the same sequence of (possibly overlapping) sections of length $\leq r$ and the sums of angle differences of the shapes on these section are equal (exact* or *with an angle tolerance).*

SBC is motivated by the fact, that on a local level the same directional differences should occur if we consider curves or swerving manoeuvres as units, see Figure 2.5 for concrete examples.
Of course, also SBC can be combined with *wobbling comparison* to deal with different samplings.

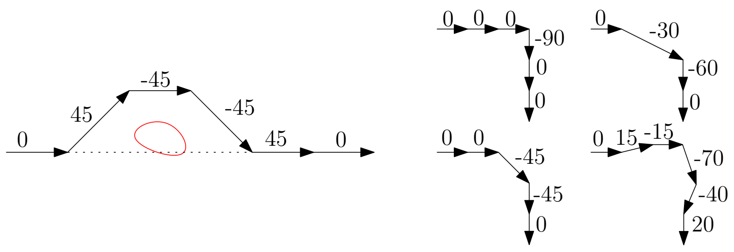


Figure 2.5.: Left: Avoiding an obstacle (red). The sum of the angle differences is 0 in conformity with the direct path (dotted). Right: Four possibilities for driving a curve, all resulting in a sum of -90.

2.4. Shape-Preserving Dijkstra

Given a path shape S (represented as GAR or LAR), the goal is now to find all vertices in the road network starting at the sequence of movements represented by the shape is possible. If there exists exactly one vertex $s \in V$ fulfilling the condition, the shape is unique and hence accurate map matching and self-localization is achievable. To check for a certain node $v \in V$ whether it is a possible start vertex, we could perform a complete Dijkstra run starting at v, backtrack all paths in the resulting search tree via predecessor labels, encode every path (according to the representation of the input shape as GAR or LAR) and then compare it to S. Iff the encoding of some path in the search tree equals S, then v is a possible start vertex. To certify that there is only one start vertex, we have to perform this approach for *all* vertices in the network. Observe that already a single complete Dijkstra run takes several seconds in larger street networks and hence is not an adequate ingredient for answering a real-time query – not to mention the total time required for performing the approach from every vertex.

Algorithm 1: Shape-Preserving Dijkstra (SPD)

```
   Input: $v \in V$ start vertex, $S = a_1, \cdots, a_l$ reference path shape, comparison model C
1  begin
2      $pred(u) = d(u) = \infty \forall u \in V$;
3      $d(v) = 0$;
4      $Q.push(0, v)$;
5      while $!Q.empty$ do
6          $u \leftarrow Q.extractMin$;
7          forall $e = (u, w) \in E$ do
8              if $d(u) + c(e) < d(w)$ then
9                  $d(w) = d(u) + c(e)$ ;
10                 $pred(w) = u$;
11             else
12                 continue;
13             if $shape(p = v, \cdots, u) + shape(e)$ is prefix of $S$ according to C then
14                 $Q.push(d(w), w)$;
15 end
```

In order to reduce the runtime, we propose a modified version of Dijkstra's algorithm called shape-preserving Dijkstra or SPD for short. The main idea is to elongate a path in the Dijkstra search tree only if its encoding is a prefix of S, see Algorithm 1. So an SPD starting at a vertex v proceeds like a normal Dijkstra, but pushes a vertex w into the priority queue only if the angle representation (LAR or GAR) of the path from v to w is equal (according to our comparison function) to a prefix $a_1, \cdots, a_i$ of $S = a_1, \cdots, a_l$. Note, that a run of SPD does not guarantee the assignment of minimal cost labels to the nodes because shortest paths might be ignored due to deviant shapes. After termination, the settled node with maximal distance indicates the longest match in the map. If this value is as least as large as the length l of S we found a complete match and v is a possible start vertex. The runtime of a SPD depends on the used comparison model. For exact and angle-tolerance based comparison we can decide for an upcoming edge whether the belonging shape fits the description of S without considering the the path section from v to u; we just have to compare $shape(e) = b_1, \cdots, b_{d(w)}$ with $a_{d(v)}, \cdots, a_l$ for the correct result. For the right and efficient encoding we store for GAR the very first absolute angle on the path from v to u for every u. For LAR we could either store the last angle on the actual path with every node or compute it on demand using the predecessor label. So the costs of an edge relaxation are proportional to the edge length in the worst case (if S has e.g. no turn changes in the respective section the check time is constant). For wobbling and range/sum-based comparison we might have to look several edges backwards to be able to compare the necessary sections. Here the range parameter w or r respectively determines the runtime.

2.5. Practicability

To argue that path shapes are a valuable stand-alone method for map matching and self-localization applications, we have to prove that the required path length for correct matching is not too long – even when considering imprecisions. For a shortest path $\pi(s,t)$ in the network, we are interested in finding the minimal i such that the prefix $a_1, a_2, \cdots, a_i$ is unique amongst the angle representations of all shortest paths in the network (if such an i exists). We call the respective sequence of angles the unique prefix of π. Note, that using fuzzy comparisons we might not be able to always find such a prefix if there are several possible sources, see Figure 2.6. Hence in that case we let our procedure return the minimal prefix that contains a unique suffix.

	GAR	avg query time (sec)	avg prefix length (m)	max prefix length (m)
$t = 0$	TAU	0.02	72	604
	MA	1.13	246	2869
	GER	86.24	169	1607
$t = 5$	TAU	0.03	125	1280
	MA	1.68	575	8439
	GER	94.75	274	4148
$t = 10$	TAU	0.05	355	4781
$r = 50m$	MA	2.36	3084	32337
$c = 0.9$	GER	134.37	684	8430
	LAR	avg query time (sec)	avg prefix length (m)	max prefix length (m)
$t = 0$	TAU	0.03	74	599
	MA	1.31	279	4480
	GER	90.24	182	1351
$t = 5$	TAU	0.03	123	1561
	MA	1.85	641	23497
	GER	95.35	287	5047
$t = 10$	TAU	0.11	222	1840
$r = 50m$	MA	3.73	1413	41036
	GER	144.45	946	78796

Table 2.1.: Unique prefix lengths in different sized road networks, with t denoting the angle tolerance and r (and c) being the parameters for RBC or SBC. Query time (in seconds) denotes the time until prefix uniqueness has been established by SPD for an $s-t$ path. Query time and the prefix length (in meters) are average values of 1000 random queries.

We used three test graphs for evaluation: The German Taunus ('TAU', small graph), the road network of Massachusetts ('MA', graph of medium size) – particularly interesting because it contains many grid-like subgraphs – and the whole of Germany ('GER') as an example of a

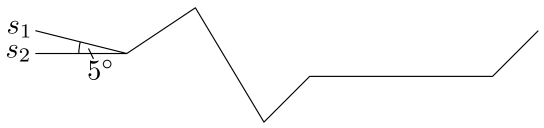


Figure 2.6: Source not identifiable for a comparison model with $t_a >= 5$.

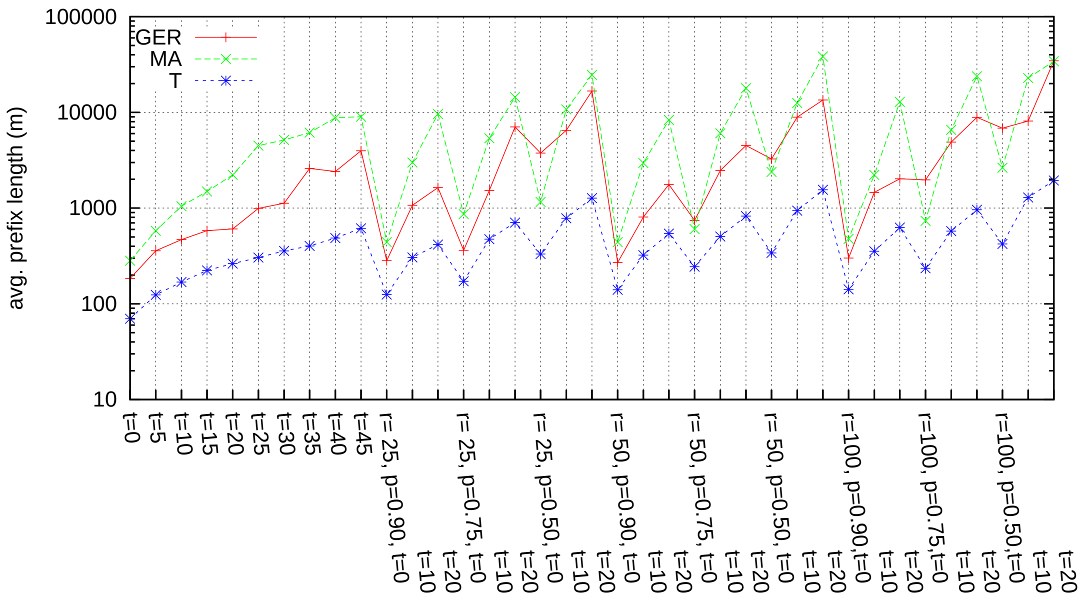


Figure 2.7.: Average prefix lengths under different comparison models for 1000 examples each.

large graph. Identifying the unique prefix for every possible (maximal) shortest path is too time-consuming, even for the German Taunus. Therefore we restricted ourselves to a large number of (s, t)-pairs chosen uniformly at random, for which we first computed the shortest path using normal Dijkstra. Then we started for every node a SPD computation and let them run until all PQs are empty.

For a choice of comparison models the average prefix lengths as well as maximum prefix lengths and timings can be found in Table 2.1 categorized according to the used path model. Moreover the plot in Figure 2.7 shows the average prefix length for a wide range of comparison parameters for *GAR*. Figure 2.8 shows for Massachusetts, that prefix lengths are actually concentrated around their mean. In Figure 2.9 one can get an idea how the number of matching paths for a reference path decreases, if the number of considered edges grows. It looks like an exponential decrease for all used comparison models, which explains the shortness of unique path prefixes. Interestingly the graph of Germany (more than 50 times larger than the one of Massachusetts) exhibits shorter unique prefixes, probably because of the less 'planned' structure of the road network due to historical reasons.

In any case, even for rather relaxed similarity measures, path shapes become unique quite early, e.g. for LAR, angle tolerance $t_\alpha = 10°$ over a range of 50 meters, the average length of the characteristic prefix of a shortest path is only 946 meters. So path shapes in principle seem to be a viable means for localization in a road network. Note that our SPD algorithm is already an algorithm to perform this localization. The running times for larger networks are prohibitive, though. Note, that starting an SPD at every single vertex of the network scales badly with the network size and therefore results in query times of over two minutes for Germany. In the next section we will present a data structure which performs this localization several orders of magnitudes faster.

Furthermore we see in Table 2.1 that using LAR or GAR does not really make that much of difference – which comes as no surprise since for short paths, LAR and GAR are essentially equivalent. We will focus on GAR in the following, while exactly the same techniques apply for LAR as well.

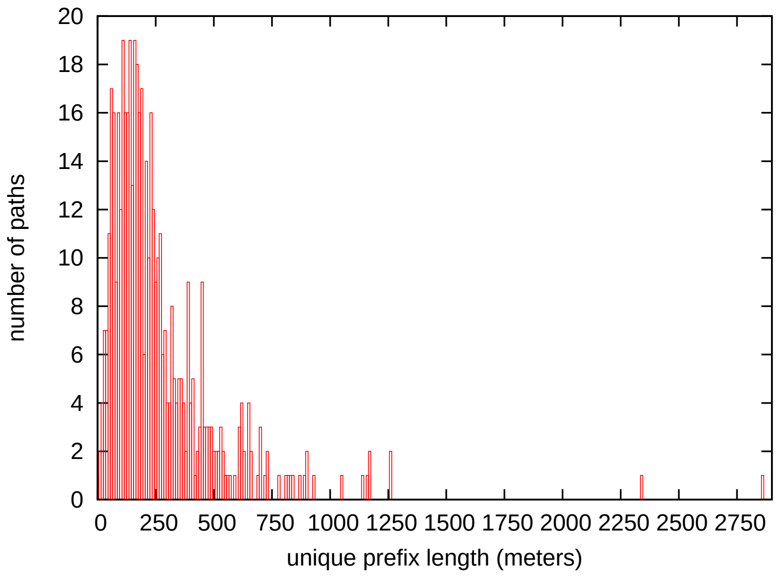


Figure 2.8.: Distribution of unique prefix lengths for MA with $t_a = 0$.

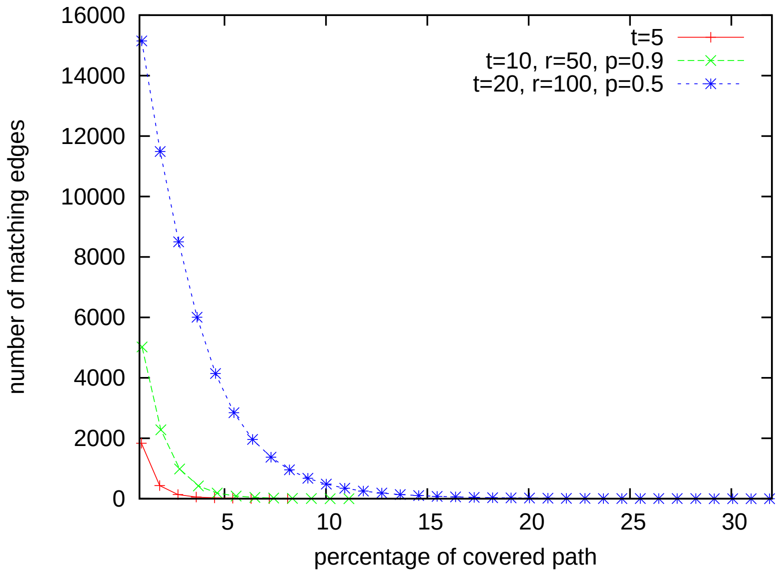


Figure 2.9.: Number of matching edges of the Taunus graph in relation to the position on the reference path for some selected models.

3. Efficient Query Answering via Generalized Suffix Trees

As seen in the last section running a SPD from every node in the network is far too time-consuming for query answering. Therefore we aim for preprocessing the graph to accelerate subsequent queries. The basic idea is to transfer our problem to the field of text search to take advantage of existing data structures and algorithms.

3.1. (Generalized) Suffix Tree

If we consider the sequence of angles defining a path as a concatenated string, classical string search algorithms can be used to identify a path in the 'text' which describes the whole road network.
To find a string pattern of size m in time $O(m)$ in a text of length n, $n \gg m$, the latter has to be preprocessed. One method to do so is constructing a (generalized) suffix tree of the text. A GST represents all suffixes of a set of strings $S_1, \cdots S_k$, fulfilling the following characteristics:

- each tree edge is labelled with a non-empty string
- there is no inner node with degree 1
- any suffix of a string corresponds to a unique path in the tree (starting from the root) with the concatenated edge labels along that path starting with this suffix

Construction. A suffix can be grafted into a GST by first identifying its longest prefix that already exists in the tree by tree traversal. If the path of this prefix ends in a node, we create a new edge and a new leaf representing the last part of the suffix (if there are remaining characters). Also, the path could end implicitly, that means the edge label contains additional characters, that are not in the suffix. Hence this edge has to be split and so its label, such that we create a new inner node that represents the longest prefix. Then we can proceed as described before. By performing this for every suffix occurring in $S_1, \cdots, S_k$ we obtain a GST of this set of strings, see Figure 3.1 for a small example. Note that there are more efficient ways of constructing GSTs, see [Ukk95], but as we will have to determine the set of strings $S_1, \ldots, S_k$ online, this more efficient construction does not apply for us.

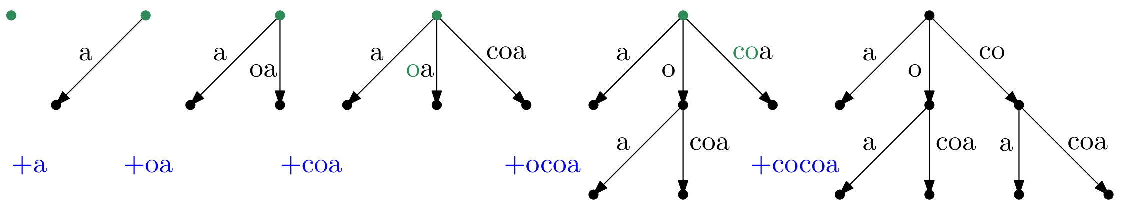


Figure 3.1.: Step-by-step construction of the suffix tree for 'cocoa'. The green color indicates the longest prefix of the next suffix to incorporate, that is already contained in the tree.

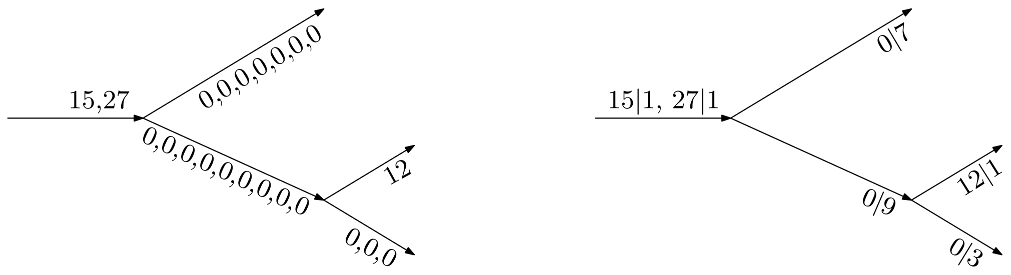


Figure 3.2.: Representing edge labels as tuples of letters and multiplicities (right image) decreases the space consumption compared to the naive variant with the 'words' fully written out (left image). In this small example, 22 numbers are necessary before compression and 12 afterwards.

Linear Time Search. With the GST being constructed, the question whether a given pattern is contained in $S_1, \dots, S_k$ can be answered in time linear in the size of this pattern, if the alphabet size is bounded, as we can associate an array with every node, which for each letter of the alphabet stores the edge (if any) whose label starts with this letter. So a query starts in the root of the GST, looking for an outgoing edge with a label, that begins with the first 'letter' of the pattern. If such an edge exists, we have to compare the respective edge label to the related pattern prefix element by element. If they are equal, we can go to the end point of the edge and repeat the search with the remaining elements of the pattern. If we always find a match, the pattern is contained in the underlying set of strings. The number of its occurrences then equals the number of leaves of the subtree beneath the last visited node. If at some point no match can be found, the pattern is not contained in $S_1, \dots, S_k$.

For a practical implementation, some tricks can be applied. For example, long straight parts of a path induce long sequences of the same 'angles' (both LAR or GAR), which can be stored in a compressed manner without affecting construction and properties of the GST, see Figure 3.2 for an illustration. While not affecting any theoretical worst-case bound, the practical performance is greatly improved.

3.2. Online Construction

Our required GST has to contain every minimal unique path prefix occurring in $G(V, E)$. But going through all possible paths and identifying in each case the unique prefix with the method described in Section 2.5 is far too time consuming.

Observation 3.1. *A prefix of length l is unique if it is not equal to any other occurring prefix with length $\leq l$ for the employed comparison model.*

Using this simple observation, we can develop an efficient approach to create the GST. We start a Dijkstra computation from every $v \in V$, stopping only when the first element in the priority queue (PQ) has a distance label greater than some selected maximal distance d_{max} (or the PQ became empty). Then this Dijkstra run is frozen, i.e. we keep the actual priority queue, as well as the distance and predecessor labels. Now for all nodes that were pushed into the PQ we compute the LAR/GAR for the respective path up to a maximal length of d_{max} starting from the source node. Note, that for each node w with $d(w) > d_{max}$ we do not have a real endpoint of the prefix in the map and therefore have to use an implicit node instead, i.e. we pretend that the respective edge is split by a dummy vertex with distance exactly d_{max}.
The representations then get grafted into our tree T. To that end, every vertex in T has a tag, whether its unique, and a pair of vertex IDs, marking the start and endpoint of the respective

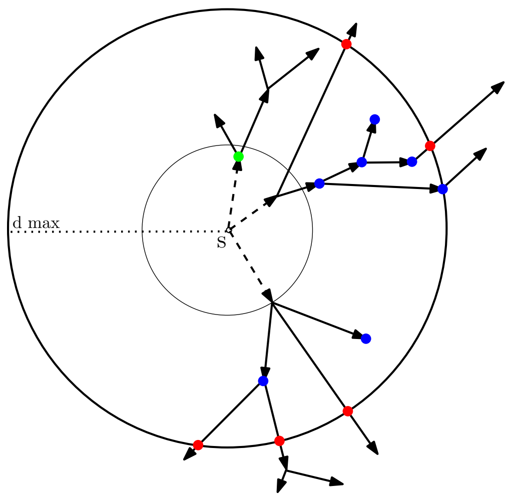


Figure 3.3.: Prefix elongation from source s. The smaller circle indicates the previous maximal distance. Blue points mark nodes, which were settled in the current round. Red points show implicit nodes, that have to be considered to assure prefix uniqueness on 'real' nodes. The green point marks the endpoint of an already unique prefix, therefore the respective subtree can be ignored.

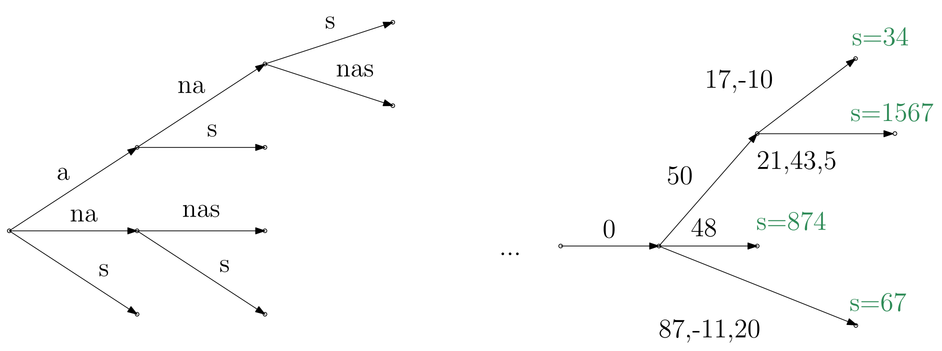


Figure 3.4.: Complete (G)ST on 'ananas' (left) and a small cutout for a GST on path shapes (right).

path (-1, if the endpoint is implicit). If a vertex is revisited with a different end point, its initial true unique tag is set to false (implicit vertices are always false tagged). If the endpoint is the same but the start point differs, we have the situation as depicted in Figure 2.6. Here the node stays unique but we add the additional source to the vertex. Having performed this for all frozen Dijkstra computations, we check in T for the newly created vertices, if they are labelled unique.

For every unique vertex the search subtree below the respective target node does not need to be considered anymore for any related source, pruning the search space of the respective Dijkstra run considerably. As long as there are non-empty PQs of some Dijkstra runs, we repeat this procedure, while increasing d_{max} in every round (see Figure 3.3).

Note, that there might be superfluous nodes and edges in T after each round, namely the subtrees beneath vertices labelled unique as well as edges and vertices referring to implicit nodes. These vertices and edges can be deleted.
In the end we can delete every auxiliary information attached to the suffix tree nodes, except the source vertices of unique path shapes (see Figure 3.4 for a small example).

At this point we have created a GST on the necessary prefixes of path shapes, which is sufficient to answer *exact* path shape queries. As queries will be based on inherently imprecise measurements, we have to think about how our GST data structure is of use for similarity searches.

3.3. Tolerating Imprecisions

So far we have learned how to construct a GST under a imprecision-intolerant comparison model. Of course, our goal is to construct and use our GST in a imprecision-tolerant manner. To that end let us first describe a *fuzzy tree traversal*, whose goal it is to determine all strings in the GST which are similar (according to one of our imprecision-tolerant comparison notions) to some given query string $a_1, \dots, a_l$. We obtain those strings by traversing the GST from the root, exploring not only the single path which matches but all paths which are 'similar' to the query string. Unfortunately, creating a GST as described in the previous section together with fuzzy tree traversal does not really solve our problem, as a typical query will then return *several* strings. Our GST creation procedure only made sure that under *exact* comparisons the path prefixes were long enough to guarantee uniqueness, for imprecision-tolerant queries, longer prefixes might be necessary.
To that end we have to modify the procedure for deciding whether a node in the GST is unique or needs to be further explored. In case of exact queries we get this information by a simple tree search, for fuzzy queries we have to employ the above described fuzzy tree traversal each time we add a path and mark all nodes inside the matching tree as non-unique if the respective target does not equal the one on the new path. This would increase construction time considerably, so we apply a more efficient approach. Additionally we attach to each node in the GST the length of the respective prefix as well as a pointer to its parent node. This allows for checking for a certain path in the tree if it is similar to any other contained path by selecting all nodes with an attached length greater or equal to the length of this path and performing a backwards tree traversal as long as the two paths are similar according to the employed comparison model. Hence we can decide *at the end of each round* for every newly created node with a unique tag whether this tag is feasible. This improves running time, as the number of unique nodes after grafting all path codes up to a certain length into the tree is considerably smaller than the number of temporary unique nodes during the prefix elongations.

3.4. Query Answering

Using the preprocessed GST, we can answer queries more efficiently. Depending on the kind of query – map matching or self-localization – we use different approaches to retrieve the result.

Map Matching. At query time, we are given a candidate path shape; we transform this path shape to the desired representation and then employ the constructed GST to identify a source vertex s for this path shape in our map by performing a (fuzzy) tree traversal until identifying a unique node. As we are not only interested in the starting point and the end point of the unique prefix, but its corresponding path in the map, we can run a single SPD from the identified starting point. Compared to the naive employment of n SPD computations this reduces the running time by orders of magnitudes.
In large networks, even this single SPD execution could be too expensive, as its running time is typically superlinear in the length of the longest path explored. In this case, a natural speed-up technique is the piece-wise reconstruction (similar to the one employed in [EFH$^+$11] for map matching): If a path is very long, we expect the unique prefix to be relatively short. That means,

if we consider the remaining path without this prefix, it still should be long enough to contain a unique prefix on its own. Again we can find this prefix by using the GST. Therefore we can reconstruct the path piecewise by identifying the paths of the unique prefixes one after the other and concatenate them. Of course, reconstructing the pieces is inherently parallelizable, reducing the query time even further on multi-core machines. If there remains a non-unique tail, we have to start a SPD in the last correctly identified vertex to reconstruct the whole path.

Self-Localization. Of course, the approach for answering map matching queries also provides the vehicle's actual position as a by-product. But to locate oneself in the street network it is not necessary to map the whole driven trajectory. Instead it is sufficient to find the minimal unique tail of the shape and identify the respective target node in the network. For that purpose it would be reasonable to create the GST by running Dijkstra computations on the *inverse* graph and encoding the resulting paths. Based on that we can perform on query time a tree traversal using the encoding of the inverted shape. As soon as a unique node is found in the GST, the actual position of the vehicle is given by the start vertex assigned to this node. Hence we can answer self-localization queries only by a single traversal of the GST *without the necessity of running Dijkstra's algorithm at all*.

4. Experimental Results

In this section we first discuss how the map preprocessing can be performed efficiently in practice. Afterwards the query times for exact and fuzzy queries are evaluated. Finally we introduce quality metrics for the map matching problem and analyze how accurate our approach is when varying the density and the precision of the given measurements. Timings were taken on the AMD Opteron 6172 with 2.1 GHz and 96 GB RAM. In the following, we restrict ourselves to the GAR model, since – as seen in Section 2.5 – for short paths they are essentially equivalent. We have conducted the same experiments with the LAR model and experienced very similar results.

4.1. Preprocessing

We build generalized suffix trees for all three test graphs. In order to do so, we need to choose an initial value for d_{max} and decide how it should be increased in every round. The larger the growth rate, the more paths have to be considered in one round. Because we cannot decide which of them already contain unique prefixes before the end of the round, there can be many superfluous encodings, tree traversals, creation of new nodes and edges as well as deletions. Choosing the growth rate of d_{max} very small, we might have rounds where no new paths are feasible at all, still we have to spend time on that. Figure 4.1 shows the preprocessing time for MA depending on an *additive* growth rate for d_{max} (added after each round). The minimum is near $150m$, so we chose this as additive growth rate. The preprocessing time can be even improved, when employing this constant growth rate until 75% of the Dijkstra runs have completed and double the growth rate in each round afterwards. In Figure 4.2 we see that the number of completed

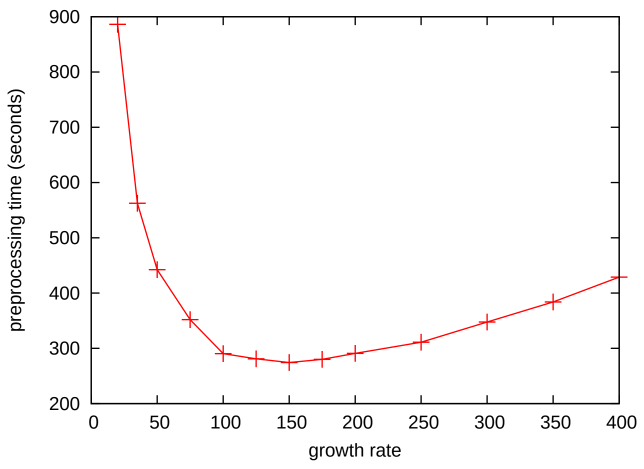


Figure 4.1.: Preprocessing time for the MA graph depending on of the growth rate of d_{max}.

Dijkstra computations (PQ ran empty) depending on the search radius follows a typical saturation curve. Up to $1250m$ there is an exponential increase, so 85% of the Dijkstra runs have completed at this point; the remaining 15% runs have finished when the search radius exceeds $3750m$. The resulting preprocessing times for all three benchmark graphs can be found in Table

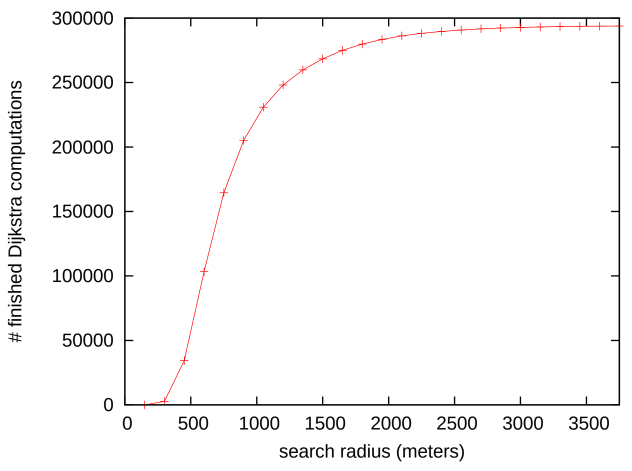


Figure 4.2.: Number of completed Dijkstra computations versus search radius for the online GST creation of MA.

4.1 along with the final size of the respective GST and the maximal search radius, that was necessary to make all prefixes unique. The effort required for preprocessing might look prohibitive at first sight (for Germany $2-3$ days on a single core and using about 70 GB of RAM on our server), but remember that this preprocessing time has to be spent only *once*. The resulting data structure is less than 2 GB in size, hence comparable to the road network representation itself, and can easily be stored even on mobile devices with microSD cards (typical size $8-32$ GB).

		TAU	MA	GER
$t_a = 0$	time (sec)	2	185	13105
	# nodes (GST)	51793	2593809	86271669
	search radius (m)	1350	5293	4680
$t_a = 5$	time (sec)	11	1701	114487
	# nodes (GST)	77784	2789902	117638920
	search radius (m)	2703	16350	9125
$t_a = 10$	time (sec)	46	3781	196580
$r = 50$	# nodes (GST)	135265	3805956	697329542
$c = 0.9$	search radius (m)	5004	42800	12750

Table 4.1.: Experimental results of the online GST creation for exact and fuzzy paths.

4.2. Query Times

We presented two approaches to answer a query, i.e. to identify a given path shape in the map: In Section 2.5 we described the naive way to tackle the problem, by starting a SPD in each vertex and waiting until all these runs have finished. Using GSTs to extract the path's source as described in Chapter 3, we can answer a query using only a single SPD, starting in that particular source. The query times of these approaches are collected in Table 4.2 for some selected comparison models. Timings contain path encoding as well as the cumulated runtime of all necessary SPDs and where appropriate the time for performing a (fuzzy) tree traversal.
Unsurprisingly, the naive approach ends up last for all considered inputs. Employing the GST for identifying the source and subsequent path exploration using a single SPD leads to a speed up of factor of 3000 for answering exact queries in the Germany graph.

		TAU	MA	GER
$t_a = 0$	naive	0.0235	1.1355	86.241
	GST+SPD	0.0002	0.0008	0.027
$t_a = 5$	naive	0.0347	1.6810	94.750
	GST+SPD	0.0010	0.0720	0.304
$t_a = 10$	naive	0.0522	2.3649	134.374
$r = 50m$	GST+SPD	0.0035	0.1318	2.233
$c = 0.9$				

Table 4.2.: Overview of query times for selected comparison models using different map matching approaches. Timings (in seconds) are averaged over 1000 random queries.

4.3. Accuracy

To evaluate the accuracy of our approach we performed two different experiments: On one hand we checked under which comparison models an exact path from the map can still be identified uniquely. On the other hand we simulated imprecise measurements by perturbing and subsampling the input data and then evaluated for which comparison models the correct path in the map still matches this input.
To measure the quality we compare our resulting path p' to the correct path p using the same metrics as in [LZZ+09]:

- $A_N(p, p')$ denotes the percentage of edges of p that are *not* matched by p'
- $A_L(p, p')$ denotes the percentage of the length of p that was *not* covered by p'

The only possible reason for an exact path not to be identified correctly is the path being too short and hence occurring multiple times in the map. This means we either get $A_N = A_L = 0$ in case of a unique match or $A_N = A_L = 1$ otherwise. For fuzzy queries our quality measures can take any value in $[0, 1]$, because apart from paths being too short, source and target vertices of a path can be ambiguous under the chosen comparison model and also deviations in the middle are possible.

For the evaluation we chose the road network of Massachusetts, because retrieving path shapes is most challenging here due to the grid-like substructures. In Figure 4.3 one can see the quality of our results for paths derived from the map, that are restricted by their lengths. As to be expected, the error rate is high for short paths but improves rapidly with growing path lengths, identifying almost 100% of the paths totally correct even for fuzzy models.

Normally the input won't be a polyline describing exactly a path in the map, but the angles might differ due to measurement uncertainty, the length of the driven route might vary slightly and depending on the density of the measurements we might miss some turn changes. If the angle and length deviation is bounded by a constant, we can use these values directly as angle tolerance and allowed wobbling range and therefore we can still guarantee to identify the path, if it is long enough. The results of the simulation of such queries can be found in Table 4.3. At first we randomly added to each angle a value in $\{-5, -4, \cdots, 4, 5\}$ (row 1:angles ± 5). The resulting path shape is typically not present in the map, hence without an angle tolerance we are not able to identify the path. Next, we perturbed the edge lengths randomly, but restricted the new total path length to differ not more than a half percent from the original one (average 1 km deviation for MA). Only a comparison model using wobbling (here with a range of 250 m) enables us to match the path in the map. Finally we simulated different measurement densities.

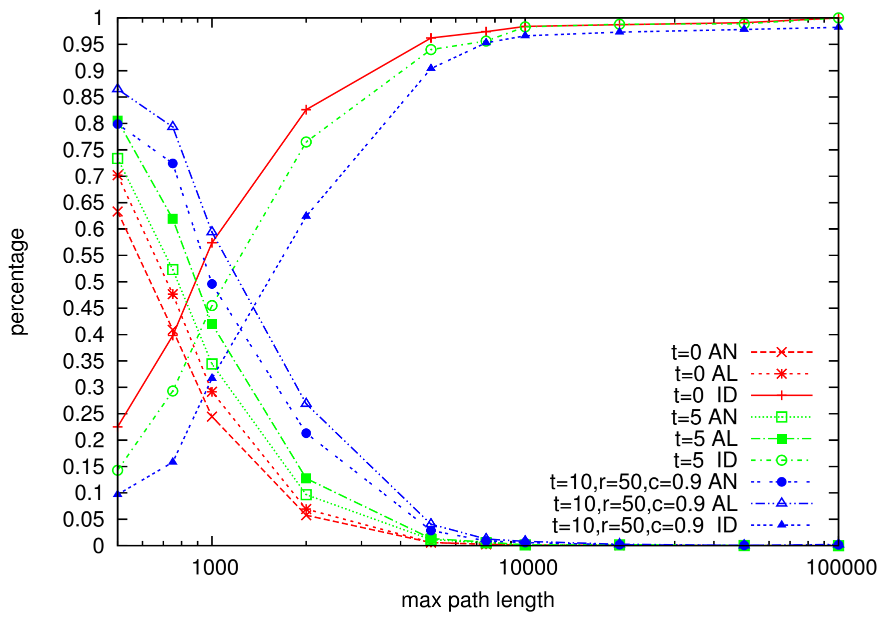


Figure 4.3.: Quality of our approach measured for some selected comparison models. *AN, AL, ID* are averaged over 1000 random queries with AN=A_N, AL=A_L and ID denoting the percentage of exactly identified paths.

Sparse measurements lead to smoother polylines, that exhibit quite big differences to the original path on small sections. Therefore only a model with range-based comparison leads to usable results.

Perturbation	$t_a = 0$	$t_a = 5$	$t_a = 10, r = 50$ $c = 0.9, w = 250$
angles ± 5	0.989/0.991	0.000/0.000	0.000/0.000
length $\pm 0.5\%$	0.881/0.931	0.864/0.888	0.005/0.001
density 1 m	0.000/0.000	0.000/0.000	0.001/0.000
density 2 m	0.966/0.973	0.942/0.940	0.002/0.003
density 5 m	0.991/0.995	0.977/0.982	0.014/0.017
density 10 m	0.993/0.998	0.992/0.995	0.058/0.061

Table 4.3.: A_N/A_L for several input perturbations and comparison models (averaged over 1000 examples).

5. Discussion and Extensions

We have presented a novel approach for self-localization and map matching which is based on relative movement patters instead of absolute positional information. Our experiments revealed the soundness of the approach for real-world application. While achieving a precision comparable to GPS, the main advantage is the autonomy of the used devices that gather the movement data. This not only makes the data retrieval more robust but also enhances the privacy of the driver significantly as no communication with third parties is necessary at all.

Natural future directions of research are reduction of the preprocessing times, consideration of *all* possible paths instead of shortest only (here Dijkstra-based preprocessing will not work anymore, and the practicability has to be validated in this scenario), and incorporation of other information sources. If additional information apart from the relative movement pattern is available, the problem gets easier, of course; for example, many mobile devices nowadays have an *electronic compass* built-in which can be used to enrich the path shape information.

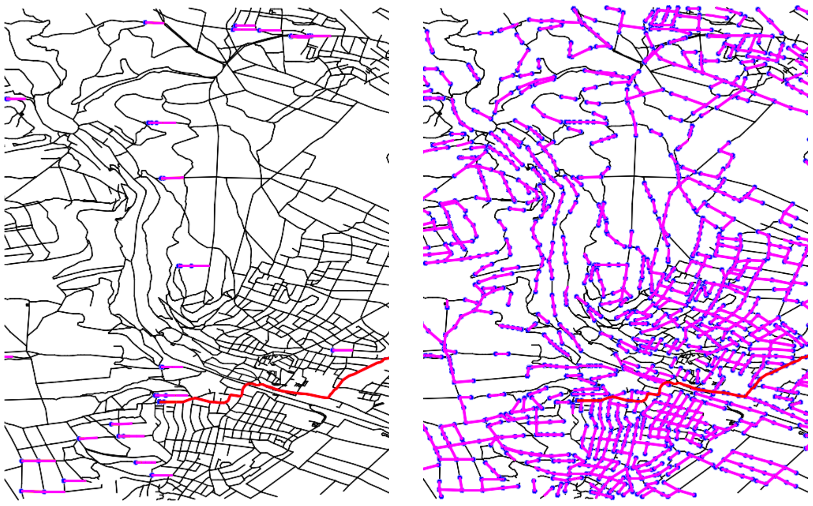


Figure 5.1.: Reference path (red) and its matches under the comparison model $t_a = 5$, left with compass information, right without (GAR).

In this case, we would naturally encode the path as sequence of (absolute) directions; the number of paths with the same shape representation shrinks drastically compared to the models considered so far, see Figure 5.1. Unsurprisingly, this results in much faster preprocessing and query times. Our approach can also be combined with imprecise but *absolute location information* in a natural way by excluding possible source/target vertices in the GST traversal while answering queries. Car manufacturers have been using techniques to incorporate relative movement patterns for their self-localization, they are only meant as a backup for a precise localization method like GPS during short periods of time (e.g. while driving through a tunnel), though. We can also make use of very imprecise location information like 'we are currently in the state of Bavaria'.
Furthermore, *height information* can also be acquired fully autonomously (at least about the *change* of height via a barometer) and incorporated into a description of a path shape. While probably not of much use in the midwest of the US, the height profiles of routes in the Alpes might be characteristic even without any (horizontal) directional information.

Part III.

Route Planning for Electric Vehicles

”The less energy you need, the more distance you need.”

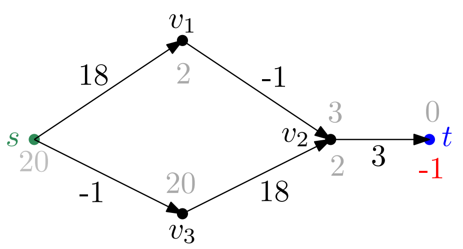
(golden rule of route-planning for electric vehicles)

6. Motivation

In recent years E-mobility has emerged as an important means to reduce the consumption of fossil fuels. Electric Vehicles (EVs) are battery-powered and the necessary electricity can be produced from regenerative sources like wind, hydropower, or solar energy. Furthermore EVs typically exhibit lower emissions to their immediate environment in terms of combustion gases or noise levels. To accelerate the transition to E-Mobility several governments offer reduced taxes for electric vehicles compared to fuel driven cars, and provide federal funding for the development of green technologies. Nevertheless Electric Vehicles still wait for their great breakthrough. One reason might be the restricted cruising range of current EVs (up to 150km) due to the limited battery capacity. In use, there is the possibility to recuperate energy during deceleration phases or when going downhill, but recharging is still necessary on longer tours. Here further inconveniences become noticeable: On one hand loading stations (LSs) are still sparse in most countries – certainly not as widespread as gas stations – therefore one cannot just plan a trip and rely on the presence of nearby LSs when the battery charge level drops. On the other hand reloading takes several hours hence it should be avoided if the objective includes travel time.
So a wide range of new routing problems arise that were not as relevant before: Which is the most energy-efficient route? Which destinations can be reached before the battery runs empty? Where should I recharge to minimize the total travel time? The answer to these and other related questions play a key role in enabling E-Mobility. We will present algorithmic methods and techniques which help to solve such fundamental problems efficiently.

6.1. Related Work

The problem of finding the most energy-efficient route for an EV was introduced in [AHLS10]. The authors emphasize the differences to conventional shortest path computations: The recuperation capability of the EV leads to partly negative energy consumption, therefore in a graph with edge costs reflecting energy consumption Dijkstra's algorithm cannot be applied directly. Moreover an EV must never run out of energy and while recuperating the maximal battery capacity cannot be exceeded (see an illustration in Figure 6.1). These so called battery constraints prohibit the usage of plain Bellman-Ford's algorithm as well. In [AHLS10] this problem was modelled as an instance of the constrained shortest path problem (CSP) with hard and soft constraints. More specifically a route is declared infeasible, if the battery charge level becomes negative at some point (hard constraint) and a route is less preferred if at some point recuperation cannot take place because the battery's maximal capacity is already reached (soft constraint). These constraints were handled by modelling 'absorption functions', reflecting how much energy is consumed along a certain path. The goal is then to find a path from s to t with minimal total absorption, while guaranteeing that for every prefix of the path the absorption is smaller than or equal to the battery's maximal capacity. Therefore the authors call this problem also the *prefix-bounded shortest path problem* or PBSP for short. They present an extension of Bellman-Ford's algorithm which can deal with the absorption functions and compute energy-optimal paths on that basis. The resulting runtime of their approach is $\mathcal{O}(nm)$, which is polynomial but nevertheless leads to query times of several seconds already in graphs with less than a million nodes.



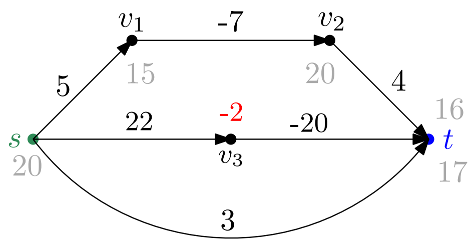


Figure 6.1.: Graph examples, where route planning for Electric Vehicles differs from conventional route planning. We assume that the battery's capacity is 20 units of energy (e.g. kWh). Battery loads are displayed in gray at each vertex. In the left image the upper path is valid, because all battery loads are ≥ 0. The lower path (although only the first two edge costs are switched) is infeasible, because it would result in a negative battery load at the target. In the right image the effect of reloading only up to the maximal capacity plays a role as well. While the upper path has only summed costs of 2, the lower path with costs of 3 leads to a higher battery load (of 18 compared to 17) at the target. The path in the middle also has only summed costs of 2 but is infeasible.

Hence this approach is not sufficient to enable real-time navigation of EVs in large street networks.

6.2. Outline

In Chapter 7 we will introduce graph preprocessing techniques that allow for answering energy-efficient path queries in time $\mathcal{O}(n \log n + m)$ which is a drastic improvement to the previous results. The key idea is to model the battery constraints as edge cost functions that allow for straightforward application of the Bellman-Ford algorithm to compute the most energy-efficient path. On top of that we generalize Johnson's shifting technique that serves as a tool to get rid of negative edge costs. This allows to answer queries in the resulting graph using Dijkstra's algorithm. While the theoretical speed-up is also reflected as reduced practical run times, they are still too large for instant query times. Therefore we additionally employ Contraction Hierarchies to the resulting graph. This is possible because our edge cost functions do not suffer from unlimited complexity growth due to chaining like e.g. many time-dependent edge cost functions. These approaches have been published at AAAI 2011 [EFS11]. Next, we deal with the efficient computation of the cruising range of an electric vehicle. If the target should not be part of the actual cruising range, recharging is necessary to arrive there. As reloading is very time-consuming, we will present an algorithm which minimizes the necessary number of recharging events. The respective preprocessing techniques are given in Chapter 8 and were published at AAAI 2012 [SF12]. Finally in Chapter 9 we study multi-criteria objectives including metrics like time, distance, reloading effort and energy-consumption. Again we present suitable preprocessing techniques to allow for efficient query answering. The respective algorithms and results have been published at ICAPS 2012 [Sto12b] and IWCTS 2012 [Sto12a]. Each chapter concludes with experiments on real-world data which underline the soundness of our approaches for practical purposes. We close the second part of this theses with a brief discussion of the results in Chapter 10.

7. Energy-Efficient Routes for Electric Vehicles

Computing the most energy-efficient route rather than the shortest or quickest path helps maximizing the cruising range of the EV and might save time intensive reloading. Moreover solving this task efficiently will provide us with a tool for tackling more complex optimization criteria, e.g. finding the energy-optimal path not exceeding a given distance constraint.

Formalized as a graph problem we are faced with the following setting: We are given a graph $G(V, E)$ with a possibly negative cost function $c : E \to \mathbb{R}$ representing energy consumption. Of course, G is not allowed to contain a negative or zero cost cycle, otherwise the EV would be a perpetual motion machine of the first degree. Moreover the maximal battery capacity $M \in \mathbb{R}^+$ is known; we also assume $c(e) \leq M$ for all edges $e \in E$ and that G is strongly connected, i.e. driving from anywhere to anywhere is possible (potentially with recharging). On query time a source node $s \in V$, a target node $t \in V$, and some $I \in \mathbb{R}^+, I \leq M$ indicating the initial battery charge level are provided. The goal is to determine a path $p = s, v_1, \cdots, v_k, t$ and a charge level $b(v_i)$ for each of these vertices such that

1. $b(s) = I$ (we have not used any energy at the start)
2. $b(v_i) \leq M, i = 1, \ldots, k$ (no overcharging of the battery)
3. $b(v_i) \geq 0, i = 1, \ldots, k$ (no running out of energy)
4. $b(t)$ is maximal over all such possible paths from s to t

Our approach to solve this problem consists of two steps. First we model the battery constraints – both overcharging as well as running out of energy – as cost functions obeying the FIFO property; hence a regular Bellman-Ford algorithm can be applied to solve the problem in $\mathcal{O}(nm)$; this simplifies and matches the result in [AHLS10]. Then by employing a generalization of Johnson's potential shifting technique to the (partly) negative edge cost functions we make Dijkstra applicable, which results in a drastically improved query time of $\mathcal{O}(n \log n + m)$. Finally we will show that the specific construction of our edge cost functions allows for the adaptation of the speed-up technique Contraction Hierarchies (CH) – resulting in ultra short query times. One important ingredient here is to bound the complexity growth of the function descriptions. This improvement is directly reflected in our experimental results on large, real-world problem instances.

7.1. Modelling Battery Constraints as Edge Cost Functions

While in [AHLS10] the authors handled the battery constraints explicitly by a check procedure embedded into the Bellman-Ford algorithm, our goal is to be able to decide for an edge individually if the actual battery load is sufficient to use the edge or not. We achieve this by modelling the constraints directly as edge costs, which results in edge cost functions dependent on the battery charge level.

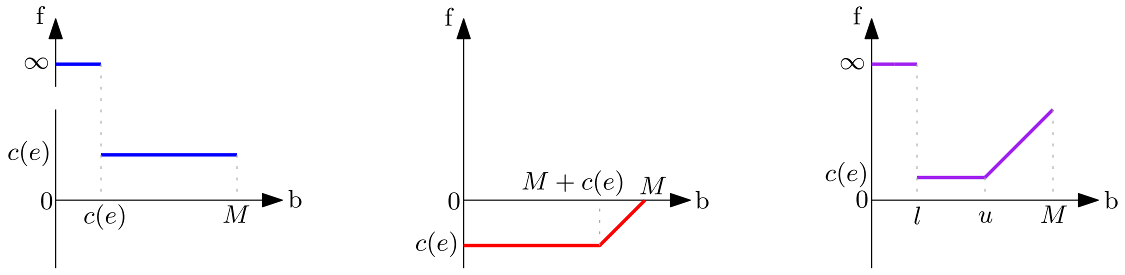


Figure 7.1.: Examples for energy-consumption functions f dependent on the EV's battery load b: for positive consumption (left), for negative consumption (middle) and for both together (right).

Never Running out of Energy: Let us look at the constraint that the EV never runs out of energy, that is, when being at a node v and considering travelling along edge $e = (v, w)$ and $c(e)$ exceeds the current charge level of the EV (i.e. $c(e) > b(v)$), we are not allowed to take this road segment. Note that this concerns only edges with $c(e) > 0$. To prohibit the use of such an edge in case the energy level does not suffice to get across it we set the respective edge costs to ∞, resulting in the following charge level dependent function $f_e^+ : \mathbb{R} \to \mathbb{R}$:

$$f_e^+(b(v)) = \begin{cases} \infty & b(v) < c(e) \\ c(e) & otherwise \end{cases}$$

No Overcharging: The constraint that the battery charge never exceeds the maximum charge level M can similarly be modelled in a cost function. Note that this only concerns edges $e = (u, v)$ with $c(e) < 0$. When arriving with a battery level $b(v)$ such that $b(v) - c(e) > M$, we want the energy level at w to be M only. This can be realized replacing the original cost with the following function:

$$f_e^-(b(v)) = \begin{cases} c(e) & b(v) < M + c(e) \\ b(v) - M & otherwise \end{cases}$$

Unified Cost Function Representation: Dealing with two different types of edge cost functions f^+/f^- (depending on the edge types) is somewhat inconvenient. Therefore we unify our cost function representation by defining a new function $f_e : [0, M] \to \mathbb{R}$ to describe the cost of an *arbitrary* edge $e = (v, w)$ (be it a downhill or uphill edge) depending on the actual battery charge $b(v)$ at node v:

$$f_e(b(v)) = \begin{cases} \infty & b(v) < l \\ c(e) & b(v) \in [l, u] \\ b(v) - u + c(e) & b(v) > u \end{cases}$$

with $l,\ u,\ |c(e)| \in [0, M]$ being parameters depending on e. For an edge e with $c(e) \geq 0$ we have $l = c(e)$, $u = M$, for an edge e with $c(e) < 0$ we have $l = 0, u = M + c(e)$. So it is easy to see that this function class subsumes the two previous definitions of edge cost functions, see Figure 7.1 for an illustration. We will now prove that f obeys the FIFO property. In this context the FIFO property essentially says that there is no benefit of starting with a lower initial battery level.

Lemma 7.1. *The edge cost function f_e as defined above fulfils the FIFO-property, i.e. for $e = (v, w)$ yields $\forall b(v) \leq b'(v)$ we have $b(w) = b(v) - f(b(v)) \leq b'(v) - f(b'(v)) = b'(w)$.*

Proof. First assume that $b(v) < l$, hence $b(w) = b(v) - \infty = -\infty$. Therefore any $b'(w)$ will fulfil the inequality. Next assume $b'(v) > u$; we derive $b'(w) = b'(v) - (b'(v) - u + c(e)) = u - c(e)$. Observe that this is a constant, hence for $b(v) > u$ equality would hold. For $b(v) \leq u$ we can bound the battery load $b(w) \leq b(v) - c(e) \leq u - c(e)$, therefore also this scenario is covered. It remains the case $b(v), b'(v) \in [l, u]$. Observing that $b(v) - c(e) \leq b'(v) - c(e)$ completes the proof. □

Having encoded the battery constraints into cost functions on the edges obeying the FIFO property we can use plain Bellman-Ford; the only difference is that when a node v is pulled from the queue, the edge cost functions on its outgoing edges are evaluated at $b(v)$. If all functions on the edge costs were positive, even straightforward Dijkstra could be applied, see [Dre68]. Unfortunately, this is not the case here (yet). The edges that lead to a gain in energy bear cost functions which are partly negative, which we alleviate later on.

7.2. Generalizing Johnson's Shifting Technique

At this point we are dealing with a graph $G(V, E)$ and for each edge $e \in E$ we have a cost function $f_e : \mathbb{R} \to \mathbb{R}$ obeying the FIFO property. For a subset of the edges f_e might bear negative function values, but no negative cycles are contained in the graph, because intuitively an EV going in circles without ever running out of power does not make any sense.

For graphs with constant, possibly negative edge costs (but no negative cycles!), the shifting idea of Johnson (see Section 1.3.1) allows for transformation of the edge costs into non-negative ones while preserving the structure of shortest paths. Generalizing this shifting technique to our edge cost *functions* requires the node potentials to become functions, so $\phi : V \to (\mathbb{R} \to \mathbb{R})$. While for constant edge costs standard Bellman-Ford can be used to determine potentials, we now have to employ Bellman-Ford to perform a profile search [DW09], which as a result not only yields for any $v \in V$ a *cost value* but a *function* describing the costs dependent on the initial charge level at the dummy source x. This is relatively complicated as well as time- and space-consuming. If the edge cost functions satisfy an additional condition (which is naturally fulfilled in our scenario), we can get away without this inconvenience:
Consider the graph which on each edge e, instead of the function f_e has constant cost $\underline{c}(e) := \min_b f_e(b)$. Assume that the graph with these edge costs does not contain any negative cycle; we can derive a potential function $\phi : V \to \mathbb{R}$ in the same manner as before for constant edge costs. Let us convince ourselves of the use of this potential function for our graph with edge cost functions. We define as new edge cost functions $f'_e(b(v))$ for every $e = (v, w)$:

$$f'_e(b(v)) := f_e(b(v) - \phi(v)) + \phi(v) - \phi(w)$$

We have to show that the resulting cost functions are both non-negative and optimal path preserving.
For the first aspect observe that $\phi(v) - \phi(w) \geq -\underline{c}(e)$ by construction of the node's potentials. Therefore it also yields $\phi(v) - \phi(w) \geq -f_e(\chi)$ for any χ and hence:

$$f'_e(b(v)) = f_e(\chi) + \phi(v) - \phi(w) \geq f_e(\chi) - f_e(\chi) = 0$$

It remains to show that this transformation does not change the structure of shortest paths. For that we inductively proof that the cost under the new edge cost function f' equals the cost under the old edge cost function f plus the target's potential $\phi(v_k)$:

Lemma 7.2. *Let $f(p)$ be the original cost of a path $p = e_1 \dots e_k$ with $e_i = (v_{i-1}, v_i)$, $v_0 = s$ with initial value $b_s = b$, and $f'(p)$ the new cost of the same path with initial value $b'_s = b + \phi(s)$. Then this yields $b'(v_k) = b(v_k) + \phi(v_k) \quad \forall k \geq 0$.*

Proof. This is certainly true for $k = 0$. For $k > 0$ observe, that $b'(v_{k+1}) = b'(v_k) - f'_{e_{k+1}}(b'(v_k))$. According to the induction hypothesis we can replace $b'(v_k)$ with $b(v_k) + \phi(v_k)$. We can then transform the equation as follows:

$$\begin{aligned} b'(v_{k+1}) &= b'(v_k) - f'_{e_{k+1}}(b'(v_k)) \\ &= b(v_k) + \phi(v_k) - f'_{e_{k+1}}(b(v_k) + \phi(v_k)) \\ &= b(v_k) + \phi(v_k) - f_{e_{k+1}}(b(v_k) + \phi(v_k) - \phi(v_k)) - \phi(v_k) + \phi(v_{k+1}) \\ &= b(v_k) - f_{e_{k+1}}(b(v_k)) + \phi(v_{k+1}) \\ &= b(v_{k+1}) + \phi(v_{k+1}) \end{aligned}$$

□

Hence our shifting procedure does not affect the structure of optimal paths – paths that are optimal under the old edge cost functions are also optimal under the new ones since the potential-induced cost of a path does not depend on the path itself.
Finally observe that in our case, the Graph (V, E) with edge costs $\underline{c}(e)$ has no negative cycle. This is easy to see since we have $\underline{c}(e) = c(e)$, that is, we have the original edge costs. By assumption this graph did not have any negative cycles.
The transformed cost function f'_e for an edge $e = (v, w)$ and $b = b_v$ can then be given explicitly as:

$$\begin{aligned} f'_e(b) &= f_e(b - \phi(v)) + \phi(v) - \phi(w) \\ &= \begin{cases} \infty & b < l + \phi(v) \\ c(e) + \phi(v - \phi(w) & b \in [l + \phi(v), u + \phi(v)] \\ b - u + c(e) - \phi(w) & b > u + \phi(w) \end{cases} = \begin{cases} \infty & b < l' \\ c'(e) & b \in [l', u'] \\ b - u' + c'(e) & b > u' \end{cases} \end{aligned}$$

with $l' = l + \phi(v), u' = u + \phi(v)$ and $c'(e) = c(e) + \phi(v) - \phi(w)$. Therefore f'_e stays in the same function class given above to describe f. As $f'_e \geq 0$ for all $e \in E$ and the FIFO property stays obviously unaffected (as it was proven for the whole function class), we can employ Dijkstra's algorithm once we have determined the potential function ϕ. So subsequent s-t queries can be answered in time $\mathcal{O}(n \log n + m)$ [FT87]. In Figure 7.2 the whole transformation process is depicted on the example of a small energy consumption graph.

A Realistic Cost Model

In the application scenario of energy-efficient routing, the typical case where a road segment $e = (v, w)$ bears negative cost (meaning a *surplus* of energy when travelling along this segment) is that the elevation of the target node w is lower than the one of the source node v, so we are going 'downhill'. It is natural to model the cost of an edge $e = (v, w)$ as $c(e) := fixed + \alpha(\eta(w) - \eta(v))$, where $fixed$ denotes a height-independent part (e.g. depending on the horizontal distance only), $\eta(w)$, $\eta(v)$ the elevations of w and v respectively, and α, a weight factor which determines how important the height-dependent part is. Clearly, with this simple model our problem gets considerably easier as the height-dependent part of the cost of a path only depends on the heights of the first and the last node in a path, all other height-dependent parts cancel out, unless overcharging takes place, which can be dealt with quite easily, though.

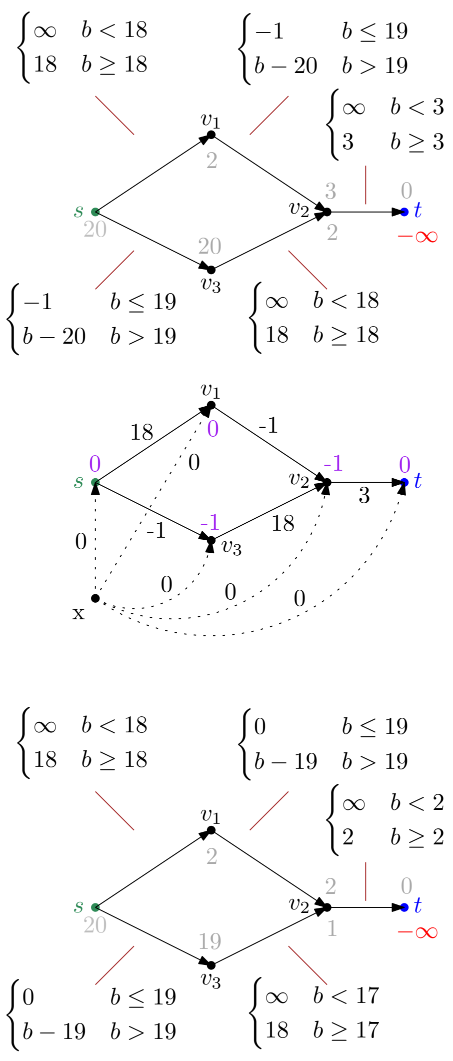


Figure 7.2.: Generalized Johnson Transformation on the example of the same graph as used in Figure 6.1. Again battery loads are assigned to the vertices in gray. In the upper image the cost functions before the transformation are displayed. Clearly some of them are partly negative. In the middle the graph with constant costs (minima of the cost functions) is given, augmented with a dummy node x which is directly connected to all other nodes via zero cost edges. The values in purple are the shortest path distances from x, which are obtained by a run of the Bellman-Ford algorithm and serve as potentials for the transformation. In the image at the bottom, the transformed cost functions are presented. The initial battery load stays the same as in the first image, because the potential of the source node s is zero. The battery loads in this image are the same as the ones in the upper image minus the potential of the respective node. Therefore the optimal path remains the same, and (in)feasibility is not affected by the transformation. Because now all cost functions are non-negative a plain Dijkstra run in this graph will reveal the optimal path in time $\mathcal{O}(n\log(n)+m)$.

It gets more interesting (and realistic) when we introduce the natural condition that the energy for going uphill ($\eta(w)-\eta(v)>0$) is not fully recuperated when going downhill ($\eta(w)-\eta(v)\leq 0$): So we have $c(e) = fixed + \eta(w)-\eta(v)$ for $\eta(w)-\eta(v)>0$ and $c(e) = fixed+\alpha(\eta(w)-\eta(v))$ for $\eta(w)-\eta(v)\leq 0$ with some $0<\alpha<1$. We drop α in the uphill case as this could easily be compensated for by scaling up the $fixed$ part. Hence α can be interpreted as the degree of efficiency of the EV.
Plugging in η as potential function ϕ we observe that for $\eta(w)-\eta(v)>0$ we obtain:

$$\begin{aligned} c'(e) &= fixed + \eta(w) - \eta(v) + \eta(v) - \eta(w) \\ &= fixed \\ &\geq 0 \end{aligned}$$

Similarly for $\eta(w)-\eta(v)\leq 0$, we get:

$$\begin{aligned} c'(e) &= fixed + \alpha(\eta(w) - \eta(v)) + \eta(v) - \eta(w) \\ &= fixed + (1-\alpha)(\eta(v) - \eta(w)) \\ &\geq 0 \end{aligned}$$

So the transformed costs will be non-negative and according to Lemma 7.2 the optimal path will be preserved under this new cost(function)s. Therefore the potential function ϕ which we had to painfully construct before can be determined with no effort if the underlying model is known. So in most practical cases we do not need to compute ϕ using Bellman-Ford but simply use the underlying elevation function on the nodes of the network.

7.3. Speeding-up Queries with Contraction Hierarchies

As already outlined in Section 1.3.3 the approach of applying Contraction Hierarchies (CH) to speed up optimal path computation in principle also works if the edge costs are functions. But in practice chaining edge cost functions – which is necessary for shortcut insertion – typically leads to drastically increased complexity and space consumption of the edge cost representations. Hence computing the CH might be impossible or the speed-up for query answering will be reduced. In the following we will exhibit some nice characteristics of our edge cost function class that allow for the *space-efficient* application of CH in our scenario.

Descriptive Complexity. For general edge cost functions the descriptive complexity might increase with every contraction step. For example consider two polynomials of degree k, chaining them leads to a polynomial with degree k^2 and so for $k \geq 2$ the description complexity increases dramatically. Similarly, chaining two step functions with different interval boundaries in the worst case leads to a combined function with about twice the number of steps.
In the following we show, that the cost function of a path in our edge cost model has bounded descriptive complexity and hence repeated node contraction does not lead to unbounded growth in function complexity.

Theorem 7.3. *The descriptive complexity of a cost function f_p of a path p on up- and downhill edges is bounded.*

Proof. Consider an arbitrary path $p = e_1 \dots e_k$ with $e_i = (v_{i-1}, v_i)$, $v_0 = s, v_k = t$ in the graph $G(V, E)$ with edge costs as defined before. Let l be the minimal battery charge a vehicle has to have at node s to reach t along p, i.e. none of the edge costs f_{e_i} become ∞ for that starting charge at v_{i-1}. Because of the FIFO property we get $\forall b(s) < l : f_p(b(s)) = \infty$. Amongst all charges at s larger or equal to l let u be the first charge where overcharging along p kicks in (it is possible that $u = l$). Again, due to the FIFO property all initial charges lower than u do not cause overcharging, hence $\forall b(s) \in [l, u] : f_p(b(s)) = \sum_{i=1}^{k} c(e_i)$. For battery charges greater than u there is at least one node v_{i^*} where the battery is fully loaded. Therefore $b = b_t$ is the same $\forall b(s) > u$ and so the path costs on this interval can be expressed by a linear function, more precisely as the difference of the initial battery power and b. So in summary we need l, u and $c = \sum_{i=1}^{k} c(e_i)$ to describe the cost function of an arbitrary path in G and hence the descriptive complexity of chained edge cost functions is bounded. $\square$

Calculation of Chained Edge Costs. When creating a shortcut between neighbours v and v'' of a node v' to be contracted, we need to combine the edge cost functions of edges e and e' that form a shortest path $vev'e'v''$ from v to v'' via v'. The newly created shortcut edge edge $\hat{e}$ gets as cost function $c_{\hat{e}}(b(v)) := f_e(b(v)) + c_{e'}(b_{v'}) = f_e(b(v)) + c_{e'}(b(v) - f_e(b(v)))$. Three cases need to be distinguished as in Table 7.1.
From this description it is obvious that the cost function for the short cut edge can be found in constant time.

	$l_{\hat{e}}$	$u_{\hat{e}}$	$const_{\hat{e}}$
$u_e < l_{e'} + c(e)$	M	M	∞
$u_e \in [l_{e'} + c(e), u_{e'} + c(e)]$	$\max(l_{e'} + c(e), l_e)$	$\max(l_{\hat{e}}, u_e)$	$c(e) + const_{e'}$
$u_e > u_{e'} + c(e)$	$\max(l_{e'} + c(e), l_e)$	$\max(l_{\hat{e}}, u_{e'} + c(e))$	$c(e) + const_{e'}$

Table 7.1.: Case distinction for chaining edge cost functions.

No Space Overhead. In practice it might not be enough to have a path cost function with bounded descriptive complexity. Consider for example a cost function linear in $b(v)$ with slope larger than one on every edge. When chaining this type of cost function along some path, the coefficient of the (still linear) chained cost function grows exponentially in the length of the path. In the worst case this might lead to coefficients which need $\Omega(n)$ space or we have to deal with rounding errors. Fortunately, in our case parameters only get added or subtracted, as the slope of our linear cost function in the interval $]u, M]$ is always 1, see Table 7.1. Therefore, irrespectively how many edge cost functions are chained, we never require more space than the original edge costs, even better, the parameters also never exceed the range $[-M, M]$, as if they do we can simply set them to the interval boundary without changing the function. So we are able to chain any number of edges and use the same data types to store the resulting parameters as for the original edge costs.

Creation of Shortcuts When contracting a node v and considering edges $e_1 = (w, v)$ and $e_2 = (v, z)$, a shortcut (w, z) with the chained cost function of e_1 and e_2 has to be added, if there exists any battery level at w for which the path $p = e_1, e_2$ is the energy-optimal one to reach z. Naively this requires computing shortest paths from w to z at all possible battery charge levels $b(w) \in [0, M]$ and comparing them with the cost function of the path e_1, e_2; in fact it is possible to do exactly this using a so-called *profile search* [DW09], which is relatively time- and space-consuming, though. Instead, we discretize the range of possible battery charges and compute shortest paths for these charge levels. For each of these paths (containing at least one node different from $w,\ v,\ z$) we consider its cost function (ranging over all possible charge levels); if for any charge level at least one of these cost functions does not exceed the cost function of e_1, e_2 the shortcut is *not* necessary. This can easily be checked by inspecting the cost functions at all breakpoints. This does not compromise correctness, it might lead to the addition of some unnecessary shortcuts, though. Note that this construction of the CH sometimes leads to multiple edges between a pair of nodes since our allowed function class does not allow for construction of shortcuts bearing the *minimum* of a set of cost functions. Our experimental evaluation shows, though, that this only happens rarely and still allows for a drastic speed-up.

Query Answering. For constant edge costs, queries in a CH-graph are answered in a bidirectional manner. Unfortunately, using edge cost functions the backward search cannot be performed because the necessary function arguments are unknown. So to answer a query from some node s to t with given initial battery level I in the created CH-graph we run a breadth first search (BFS) from t identifying and marking all edges on downward paths to t. Then we start a Dijkstra run in s which considers upwards and marked edges (upward edges only if the previous edge on the actual path was not a marked one). As soon as Dijkstra's algorithm settles the target node t, the optimal path can be backtracked via the predecessor labels.

7.4. Experimental Results

We present results for a large subset of our test graphs. The CH was computed on the AMD Opteron 6172 with 2.1 GHz and 96 GB RAM, while queries were answered using the Intel i3-

2310M processor with 2.1 GHz and 8 GB RAM. The edge costs were modelled according to Section 7.2 as linear combination of distance and height difference. Here, the power efficiency (α) of the EV is assumed to be 0.25, i.e. when driving downhill the EV can regenerate a quarter of the potential energy induced by the height difference of the path.
We first evaluate the preprocessing phase by comparing it to a conventional CH on euclidean distances in the same graphs, see Table 7.2. For creating the energy CH-graph we used five initial battery charge levels during each witness search, namely $0, 0.25M, 0.5M, 0.75M$ and M. We observe that both preprocessing time and number of added shortcuts increase when using the battery-dependent energy cost functions. The greater preprocessing time results from the increased number of Dijkstra runs per witness search, having to evaluate cost functions instead of constant values and a higher number of shortcuts and hence witness searches. However the effect of additional shortcuts is not drastic and we receive a rather sparse CH-graph containing only about twice the number of edges compared to the original one (although multiple edges between two nodes are possible).

	dist		energy	
	prep. time (s)	edges	prep. time (s)	edges
TAU	0.11	47061	0.16	47162
SH	0.86	379170	1.08	380477
WIN	6.96	1953330	10.48	1960667
BW	13.23	3910827	21.85	3924349
SG	145.10	23236361	780.11	23358690
CAL	313.80	44430253	1551.25	44876498
GER	457.57	56857615	6597.83	57729012
JAP	625.63	108841056	14329.87	115726398

Table 7.2.: CH preprocessing time (in seconds) and total number of edges in the resulting graph for constant costs (distances) and the energy-consumption functions.

On that basis we compared query times in the CH-graph to Bellman-Ford's algorithm in the original graph and Dijkstra's algorithm in the transformed graph (with non-negative edge costs only), see Table 7.3. The initial battery charge L was chosen such that the underlying search space was not restricted by insufficient initial energy. For lower battery charges, queries are even faster, of course. Nevertheless we can see a dramatic speed-up due to our preprocessing techniques, with an impact proportional to the graph size. Replacing Bellman-Ford's algorithm with the Dijkstra approach in the transformed graph already provides us with two to five orders of magnitudes faster query answering. Applying CH on top, we can reduce the runtime by

	BF		Dijkstra			CH-Dijkstra		
	time (ms)	polls	time (ms)	polls	speed-up	time (ms)	polls	speed-up
TAU	$1.4 \cdot 10^{1}$	$2.2 \cdot 10^{5}$	0.99	5457	14	0.06	72	15
SH	$6.4 \cdot 10^{2}$	$9.3 \cdot 10^{6}$	14.63	45785	43	0.12	89	121
WIN	$5.0 \cdot 10^{4}$	$3.7 \cdot 10^{8}$	118.51	278938	426	0.80	223	147
BW	$1.3 \cdot 10^{5}$	$8.2 \cdot 10^{8}$	215.312	489573	611	1.66	301	129
SG	$3.6 \cdot 10^{6}$	$3.4 \cdot 10^{10}$	1500.59	2806930	2397	21.52	1504	71
CAL	$1.0 \cdot 10^{7}$	$3.2 \cdot 10^{11}$	2562.16	6274860	4056	33.23	1358	66
GER	$2.7 \cdot 10^{7}$	$7.1 \cdot 10^{11}$	5752.01	9506489	4732	38.86	7709	148
JAP	$1.0 \cdot 10^{8}$	$2.2 \cdot 10^{12}$	6492.58	14431809	16082	44.93	10024	144

Table 7.3.: Query times (in ms) and polls averaged over 1000 random queries for the three proposed ways to compute energy-optimal paths.

another factor of 100 for larger graphs, resulting in a total speed-up of about two million for the Japan graph. So finally we achieve query times of only a few centiseconds for graphs with millions of nodes and edges.

As expected for real data the percentage of edges with negative costs is very small even for graphs with large height differences. Still, taking into account energy consumption and recuperation due to the height differences results in quite different paths, see Figure 7.3. Note that the paths in Figure 7.3 do not differ much in length but considerably in terms of energy consumption. Choosing the right path might make the difference between being able to reach the target or to run out of energy in the middle of the trip. The EV chooses a path that avoids driving uphill as far as possible, leading to a higher final battery charge see Figure 7.4. Energy can also get lost when driving downhill, see Figures 7.6 and 7.5. If the initial battery charge is more than 96.5875% we always end up with a final charge level of 98.5%. That means, starting with a full battery leads to a positive total energy consumption due to the overloading constraint, while we could gain energy on the very same path using a smaller initial battery charge.
In general we observe that routing vehicles under battery constraints and height-induced edge cost functions leads to considerably different optimal path structures, still queries can be answered very efficiently with our approach.

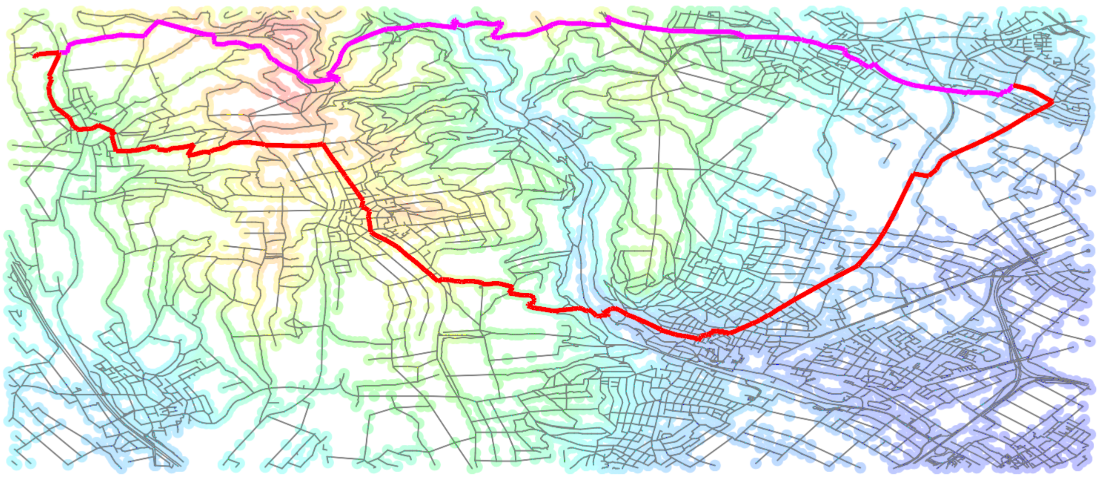


Figure 7.3.: Shortest (pink) and energy-optimal path (red) for the same source and target. The node colors indicate elevation ranging from 99m as deep blue to 412m as red. The energy-optimal path obviously makes a bit of a detour, but exhibits less hard climbs.

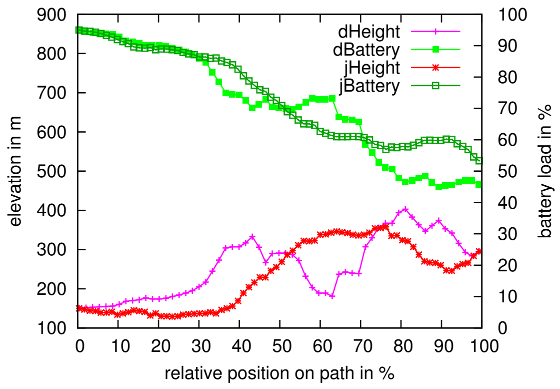


Figure 7.4.: *dHeight* and *jHeight* are the elevation profiles for the Dijkstra and Johnson paths shown in fig. 7.3. *dBattery* and *jBattery* are the corresponding battery charges. Although the euclidean length of the Dijkstra path is the shorter one, the energy consumption is higher, resulting in a lower final battery charge.

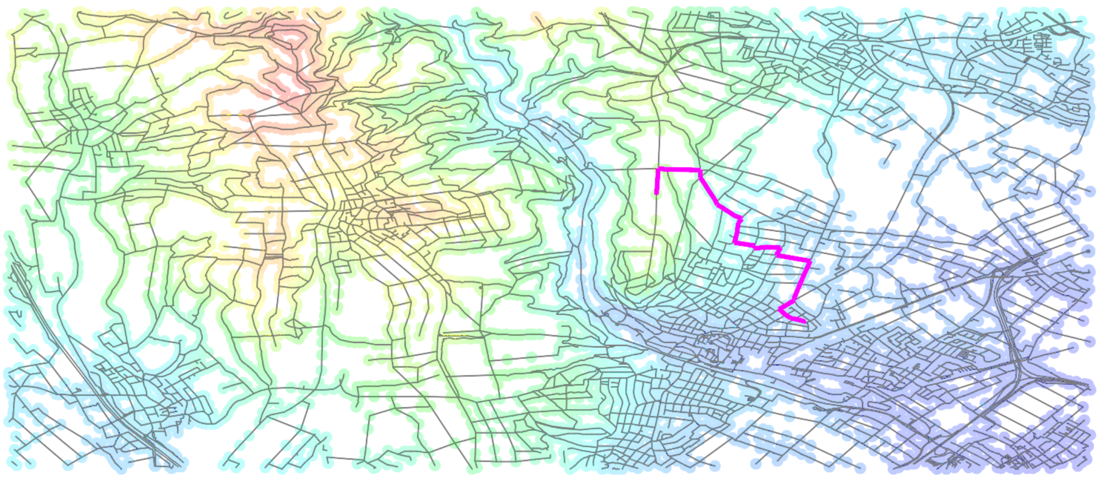


Figure 7.5.: The energy-optimal path (violet) in this image starts at 279m and ends at 149m if followed in eastern direction. The downhill nature of the path allows us to study the influence of initial battery charge on battery overloading and net energy cost.

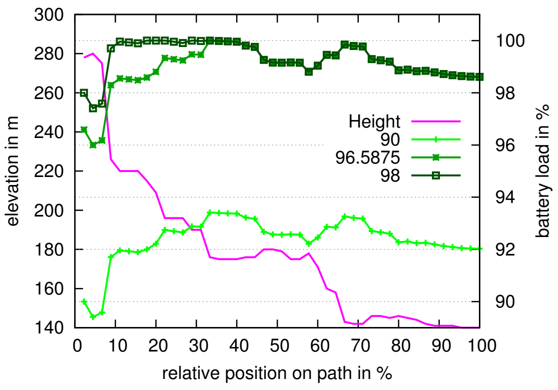


Figure 7.6.: *Height* is the elevation profile of the path, shown in fig. 7.5, starting at 279m and ending at 149m. 90, 96.5875 and 98 are the battery charge on each path node where the respective number is the initial battery charge (ie. 90%). 96.5875% marks the highest possible battery charge where no overloading occurs.

8. Reachability and Connectivity

In spite of all energy optimization, the available battery load might simply not suffice to reach a desired target. Hence the battery must be recharged at loading stations (LS) on the way to complete the trip. This leads to a variety of questions for the EV driver, e.g.:
Which destinations are in my actual cruising range? Can I get from A to B and back without recharging inbetween? Can I get from A to B (and back) when recharging is allowed? How can the number of recharging events be minimized?

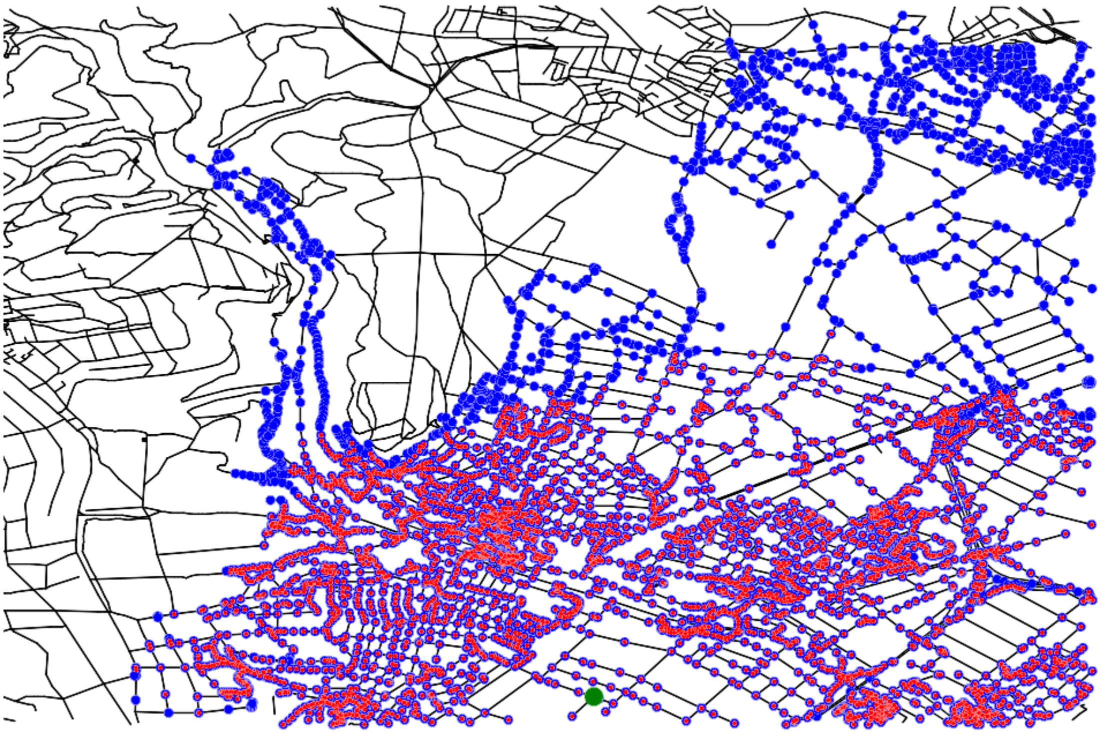


Figure 8.1.: Ev-reachable nodes (blue) and strongly ev-connected nodes (red) for a given source node (green).

For example, in the pilot project E-Tour funded by the German Ministry of Economics and Technology, tourists can use EVs to explore the Bavarian Alps. Tourists stranding during their trip with a EV due to a depleted battery is tantamount to a severe setback in the acceptance of E-mobility technology. In the long run, insights into the structure of the reachability regions of EVs will allow for a systematic and cost-efficient building of a network of charging stations and play a role in battery design.

In graph theory, a node v is *reachable* from a node u in a given (directed) graph $G(V, E)$, if there exists a (directed) path from u to v. The vertices u and v are called *connected* if either u is reachable from v or vice versa and *strongly connected* if both holds. In the context of route-planning for EVs we have to redefine the terms reachability and connectivity to account for the additional battery constraints. A path from u to v in our graph does not imply that we can travel from u to v, as the battery charge status in node u might not allow for driving along the path without running out of energy in between. Therefore in our context of EV route planning we call v *ev-reachable* from u if there exists an ev-feasible path from u to v, i.e. it obeys the battery constraints. Similarly, we call v *strongly ev-connected* to u if there exists a round tour visiting v, that starts and ends in u and obeys the battery constraints. In Figure 8.1 gives an impression, how the sets of ev-reachable and strongly ev-connected nodes look like in practice. They are influenced by the height profile of the underlying terrain. As driving downhill can recuperate some energy, the EV is able to get further in the middle of the map, where the streets follow a small valley. Note that for visualization purposes we have chosen an artificially small battery capacity in this figure (allowing to drive about 5km on average).

In this chapter we extend the techniques developed in [AHLS10] and [EFS11] to answer the above mentioned questions of EV route planning. To that end, we show how to compute the set of ev-reachable nodes and how to determine the minimal battery load necessary to reach a given target. Latter requires deliberate modelling of another type of edge cost functions than proposed in the previous section. Moreover we take loading stations into account and propose graph preprocessing techniques that allow for the efficient computation of ev-reachable and ev-connected sets at query time even in the presence of LSs. Furthermore we present an algorithm for energy-aware route planning minimizing the number of visited LSs during a trip from A to B.

8.1. EV-Reachable and EV-Connected Node Sets

Again we are given a street network $G(V, E)$ and a cost function $c : E \to \mathbb{R}$ representing the energy consumption of the edges. We assume G to be free of any negative cycles as well as knowledge of the battery capacity M of the EV. A path from s to t is called energy-optimal, if the final battery load in t is maximized by this path compared to all other paths. In the last chapter we described how such paths can be found by a Dijkstra run after an $\mathcal{O}(nm)$ preprocessing phase. During this Dijkstra run we assign labels to the nodes representing the battery charge status. Initially these labels are $b(v) = -\infty \; \forall v \in V \setminus s$ and $b(s) = I$ with $0 \leq I \leq M$. Using a max-priority-queue (PQ) for the temporary labels, Dijkstra's algorithm then settles the nodes in *decreasing* order of battery charge status. The label of a settled node equals the maximal possible battery charge status we can reach this node with when starting in s.

8.1.1. EV-Reachable Node Sets

Computing the set of ev-reachable nodes requires to check for all nodes, if there exists a feasible path from the source node. Hence we can compute the set of ev-reachable nodes as $R(s) := \{v \in V | \; b_s(v) \geq 0\}$ with $b_s(v)$ being the final battery label, that was assigned to v by a Dijkstra computation starting in s with a fully charged battery ($b(s) = M$) and running until the PQ becomes empty. Note, that any node that gets pushed into the PQ during the Dijkstra run already has a feasible battery label at this point in time. Therefore the set of pushed nodes equals the set of ev-reachable nodes R and hence our algorithm has an output-sensitive runtime of $\mathcal{O}(|R| \log |R| + |E_R|)$ with E_R being the set of outgoing edges of the nodes in R. Clearly, this is $O(n \log n + m)$ but much smaller if the reachable nodes are only a small portion of the whole network (as usually the case).

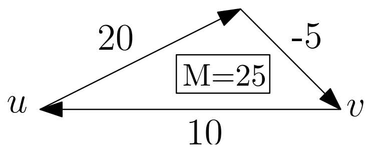


Figure 8.2.: Strong ev-connectivity is not an equivalence relation: The roundtour is feasible when starting fully charged in u, but not feasible when starting in v (with a full battery charge of $M = 25$).

8.1.2. Strongly EV-Connected Node Sets

The set of strongly ev-connected nodes is a subset of the ev-reachable node set, containing only the nodes, that also allow for returning to s without running out of energy. Note, that in contrast to the conventional definition of strong connectivity, strong ev-connectivity is not an equivalence relation anymore as reflexivity might be compromised due to the existence of edges with negative costs, see the example in Figure 8.2. To compute the strongly ev-connected nodes $C(s)$ for a source s, we could check for each node $v \in R(s)$ if a Dijkstra run from v with initial battery load $I = b(v)$ yields a feasible path back to s. The running time $\mathcal{O}(n^2 \log n + nm)$ of this approach is prohibitive, though.
To improve runtime, we do not want to decide individually for a given battery charge status on a node v if there exists a feasible path back to s, but instead compute for all nodes the minimal charge status $b_{min}(v)$, that is sufficient to complete the round tour. This leads to the following formal definition of the strongly ev-connected node set: $C(s) := \{v \in V |\ b_s(v) \geq b_{min}(v)\}$.
Based on that we define a new edge cost function on the reverse graph, that allows for computing b_{min} for all nodes by starting a single computation in s. If the original edge $e = (v, w)$ has non-negative costs we have to add the costs of the edge to $b_{min}(w)$ to obtain the minimal necessary battery load in v. If the resulting value $b_{min}(v) = b_{min}(w) + c(e)$ exceeds M, we cannot use this edge. Therefore we define the cost function f_e^+ as follows:

$$f_e^+(b(w)) = \begin{cases} c(e) & b(w) \leq M - c(e) \\ \infty & otherwise \end{cases}$$

Considering an edge $e = (v, w)$ with negative costs we subtract the absolute value of the costs from $b_{min}(w)$ to obtain $b_{min}(v)$, therefore $b_{min}(v) = b_{min}(w) + c(e)$. This edge can always be traversed, but as $b_{min}(v)$ better not be negative, we have to set it to 0 if $b_{min}(w) + c(e) < 0$. This can be modelled with the following definition of c_e^-:

$$f_e^-(b(w)) = \begin{cases} -b(w) & b(w) < -c(e) \\ c(e) & otherwise \end{cases}$$

So different from the definition of c_e^+ we have to deal here with a partly negative edge cost function, prohibiting the application of Dijkstra's algorithm at this point. It is easy to see that the two cost functions can be expressed uniformly as follows with appropriate choice of l and u:

$$f_e(b(w)) = \begin{cases} -b(w) + l + c(e) & b(w) < l \\ c(e) & b(w) \in [l, u] \\ \infty & b(w) > u \end{cases}$$

To apply Dijkstra or the Bellman-Ford algorithm [Bel58], [For62] on a graph with edge cost functions in general, these functions have to fulfil the FIFO-property.

Lemma 8.1. *The cost function f_e fulfils the FIFO-property.*

Proof. Let $f(x) = f_e(x)$. If $x < l$ we have $x + f(x) = 0$, which is the smallest possible battery charge status and therefore the inequality is fulfilled. Otherwise, let y be larger than u. Then we get $y + f(y) = \infty$, which of course is greater or equal to any possible term on the left side. Otherwise $x, y \in [l, u]$, but then $f(x) = f(y) = c(e)$ and hence the inequality is also fulfilled. □

As shown in the last chapter, Johnson's shifting technique applies for FIFO edge cost functions if the graph with modified edge costs $\underline{c}(e) = \min_x f_e(x)$ does not contain negative cycles. This condition is fulfilled in our case, as $\min_x f_e(x) = c(e)$ and the original graph with these constant edge costs was assumed to be free of negative cycles. So also in this scenario we are able to transform the cost functions within a $\mathcal{O}(nm)$ preprocessing step into non-negative ones while maintaining all optimal path structures.
So at this point we can compute the minimal necessary battery charge status for every node v by running a single Dijkstra on the reverse graph starting in s with $b_{min}(s) = 0$ and $b_{min}(v) = \infty\ \forall v \in V \backslash s$ using a min-priority-queue. It remains then to check $\forall v \in R(s)$ if $b(v) \geq b_{min}(v)$, because the FIFO property assures that in this case a feasible path exists from v to s. So all in all we need for each query two Dijkstra computations with a runtime of $\mathcal{O}(n \log n + m)$ respectively and the verification procedure, which is linear in the number of nodes. Therefore the resulting runtime is $\mathcal{O}(n \log n + m)$. Looking more closely it turns out that in the second Dijkstra run on the reverse graph it makes no sense to push any node v with $b(v) = -\infty$, because they neither can be part of the strongly ev-connected nodes on their own nor influence $b_{min}(v)$ of any v that is element of $C(s)$. Hence our runtime depends again on the size of the set of ev-reachable nodes R and can be expressed as $\mathcal{O}(|R| \log |R| + |E_R|)$ with this time E_R being the whole set of adjacent edges to nodes in R.

8.2. Considering Loading Stations

With an increasing number of EVs on the streets the density of loading stations will grow as well. A loading station is a node $l \in V$, that leads to a battery load $b(l) = M$, whenever it is visited. There are two types of LSs, either the EV is recharged via a power connection or the battery system gets completely switched. The latter is of course less time consuming but also more expensive and not as far spread as the first alternative.
The presence of LSs can augment the set of reachable and connected nodes and might affect the most energy-efficient path. In this section we develop algorithms which take into account LSs. We assume that the number of LSs is not too large ($O(\sqrt{n})$), which seems a realistic assumption to make (for Germany which has about 14k gas stations this corresponds to several thousand LSs).

8.2.1. Augmented Reachable Node Sets

We are given a source node $s \in V$ and additionally a set of LSs $L \subseteq V$. We want to compute the sets of nodes, which are ev-reachable from s directly or over a feasible path, that visits one or more LSs. The following approaches will work for any initial battery load in s, but as we want to compute the maximal ev-reachable set, we assume a fully loaded battery at the start and hence set again $b(s) = M$. We refer to the resulting set of nodes as $R^L(s)$.
Naively we can compute this augmented ev-reachable node set incrementally: At first we set $R_0 = \emptyset$ and $R_1 = R(s)$ and then incrementally $R_i = R_{i-1} \cup \bigcup_{l \in L \cap R_{i-1} \backslash R_{i-2}} R(l)$. We can stop as soon as $|R_i| = |R_{i-1}|$, which is then the desired result. If we have to compute $R(l)\ \forall l \in R$.

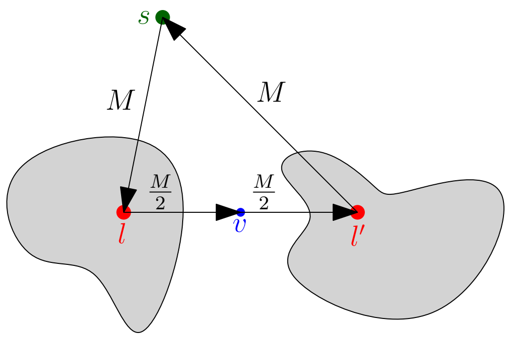


Figure 8.3.: Node v is neither in $C(l)$ nor in $C(l')$ (gray areas), but belongs to $C^L(s)$ as the round tour s, l, v, l', s is feasible.

on demand we end up with a runtime of $\mathcal{O}(|L|n \log n + |L|m)$ for this naive approach. Of course, the set $R(l)$ is invariant under the choice of s and therefore we could precompute it for all LSs, taking time $\mathcal{O}(|L|n \log n + |L|m)$ and space $\mathcal{O}(|L|n)$. Subsequent queries need then a runtime of $\mathcal{O}(n \log n + m + |L|n)$, because computing $R(s)$ requires time $\mathcal{O}(n \log n + m)$ and as each node in the graph might be contained in $\mathcal{O}(|L|)$ reachable sets, we still need time $\mathcal{O}(|L|n)$ to compute their union.
The following method will also take $\mathcal{O}(|L|n \log n + |L|m)$ preprocessing time, but allows for a query time of $\mathcal{O}(n \log n + m)$, using only linear additional space.
We make use of an auxiliary graph $Q(L, F)$ which is constructed as follows: For every $l \in L$ we compute $R(l)$; there is an edge $(l, l') \in F$ if $l' \in R(l)$. Q obviously has space consumption $O(|L| + |F|) = O(n)$ under the assumption of a not too dense set of LSs.
Answering a query starts again with computing $R(s)$ for the given source s conventionally, storing all contained LSs along the way in a list. Afterwards we start a BFS in Q on the set of these LSs in order to extract all indirectly ev-reachable LSs. Having the complete set of ev-reachable LSs $L' \subseteq L$, we can start a single Dijkstra run on this set by setting the battery labels $b(l) = M\ \forall l \in L'$ and $b(v) = -\infty\ \forall v \in V \setminus L'$ and pushing all elements of L' into the PQ. The result of this Dijkstra computation is the assignment of a battery label to each node v, that denotes the highest possible charge status that we can get in v, starting at *any* LS $l \in L'$. Therefore it only remains to build the union of the nodes, that were visited during this Dijkstra run, and $R(s)$ to obtain the final set $R^L(s)$.
The runtime for a single query consists of two Dijkstra runs for computing $R(s)$ and $R(L')$, requiring time $\mathcal{O}(n \log n + m)$, a BFS computation with a runtime of $\mathcal{O}(m + n)$ and taking the union of two sets, requiring at most time $\mathcal{O}(n)$. Therefore the overall runtime is $\mathcal{O}(n \log n + m)$ and hence – different from the naive approaches – independent of the number of LSs in the street network.

8.2.2. Augmented Connected Node Sets

Again we are given a source node s with $b(s) = M$ and a set of LSs $L \subseteq V$. Now we want to determine all nodes $v \in V$, for which exists a feasible round tour from s to v and back with an arbitrary number of LSs on the way.
Note, that different from computing $R^L(s)$ the augmented strongly ev-connected node set $C^L(s)$ is not equal to the union of the strongly ev-connected node sets of s and all directly or indirectly connected LSs, but a superset as illustrated in Figure 8.3. Because of that the naive incremental approach now has to maintain two sets R and R' with R containing the ev-reachable nodes and R' the ones, that s can be reached from. So we have $R_0 = \emptyset, R_1 = R(s)$. The initialization of R' is $R'_0 = \emptyset$ and $R'_1 = R^{-1}(s) := \{v \in V | b_{s\ min}(v) < \infty\}$. The augmentation of the sets consists

again of checking if in the last round new LSs were added and if so take the union with $R(l)$ or $R^{-1}(l)$ for all such LSs l. Moreover we have to remember for every node in R the maximal possible battery label and for R' the minimal necessary battery label. If both sets have their final size, we can check for all nodes in their intersection if the maximal battery label exceeds the minimal necessary battery charge. The set of nodes, for which this condition is fulfilled equals $C^L(s)$. Again the runtime and/or the space consumption scales badly with the number of LSs in the network as the theoretical runtime is similar to the one of computing $R^L(s)$.
Fortunately our auxiliary graph $Q(L, F)$ can again help speed up the computation for this scenario as follows: For a given source s we compute $R(s)$ and $R^{-1}(s)$ conventionally, constructing $L_1 = L \cap R(s)$ and $L_2 = L \cap R^{-1}(s)$ along the way. Then we mark all LS nodes in Q green, that are visited by a BFS run starting on L_1. Furthermore we mark all nodes red, that are visited in the reverse of Q starting on L_2. The set of nodes marked green *and* red L_{gr} equals the set of strongly ev-connected LSs for s. Knowing this set, we can compute $R(L_{gr} \cup s)$ and $R^{-1}(L_{gr} \cup s)$ each with a single Dijkstra run in the graph with the respective edge cost functions, taking time $\mathcal{O}(n \log n + m)$ as described in the previous subsection. Afterwards we also have to check for the nodes in $R(L_{gr} \cup s) \cap R^{-1}(L_{gr} \cup s)$, if the battery label $b()$ assigned during the computation of $R(L_{gr} \cup s)$ exceeds $b_{min}()$, that is determined by the Dijkstra run for $R^{-1}(L_{gr} \cup s)$. This needs only time linear in the size of the intersection, hence the total runtime remains $\mathcal{O}(n \log n + m)$. As a result we obtain the desired set of strongly ev-connected nodes $C^L(s)$.

8.2.3. One-to-one Queries

If we want to compute the optimal path from s to t with $s, t \in V$ in the presence of LSs, the notion of 'optimal' has to be defined first. The path maximizing the final battery charge in t could be declared optimal. But this might lead to long detours over many LSs and a high total energy consumption, though. So instead we aim for a feasible path requiring the minimal number of LSs on the way.
Again our auxiliary graph $Q(V, F)$ (augmented with uniform edge costs) can be used to answer such a s-t-query. We first compute all the LSs s can reach directly – let's call this set L_s – and all the LSs t can be reached from – L_t – as before in time $\mathcal{O}(n \log n + m)$. Then we augment Q by a dummy source l_s and a dummy target l_t as well as zero-weight edges $(l_s, l) \forall l \in L_s$ and $(l, l_t) \forall l \in L_t$. In the augmented graph we start a Dijkstra from l_s to l_t. Let $l_s, l_1, v_2, \dots v_{k-1} l_k, l_t$ be the resulting shortest path in the augmented graph. We output a feasible path from s to l_1 concatenated by the path from l_1 to l_k concatenated by a feasible path from l_k to t as final result. The two feasible paths from s to l_1 and l_k to t have been computed implicitly during the construction of L_s and L_t, respectively. It should be obvious that the approach returns the path with the minimal necessary number of recharging events and the overall running time is $O(n \log n + m)$.

8.3. Experimental Results

Our algorithms were evaluated on two benchmark graphs: The Taunus and Southern Germany – both rather hilly regions within Germany, that contain relatively many edges of negative cost, one of the main challenges for an efficient algorithmic solution (Johnson's shifting technique has to be employed because of that). Timings were again taken on a single core of the Intel i3-2310M processor with 2.1 GHz and 8 GB RAM.
In Table 8.1 we show our query times for computing reachable and connected node sets without considering LSs. The maximal battery charge status M was chosen, such that the cruising range was about 125 km, matching the real cruising range of current EVs. We can reach about 1/5 of

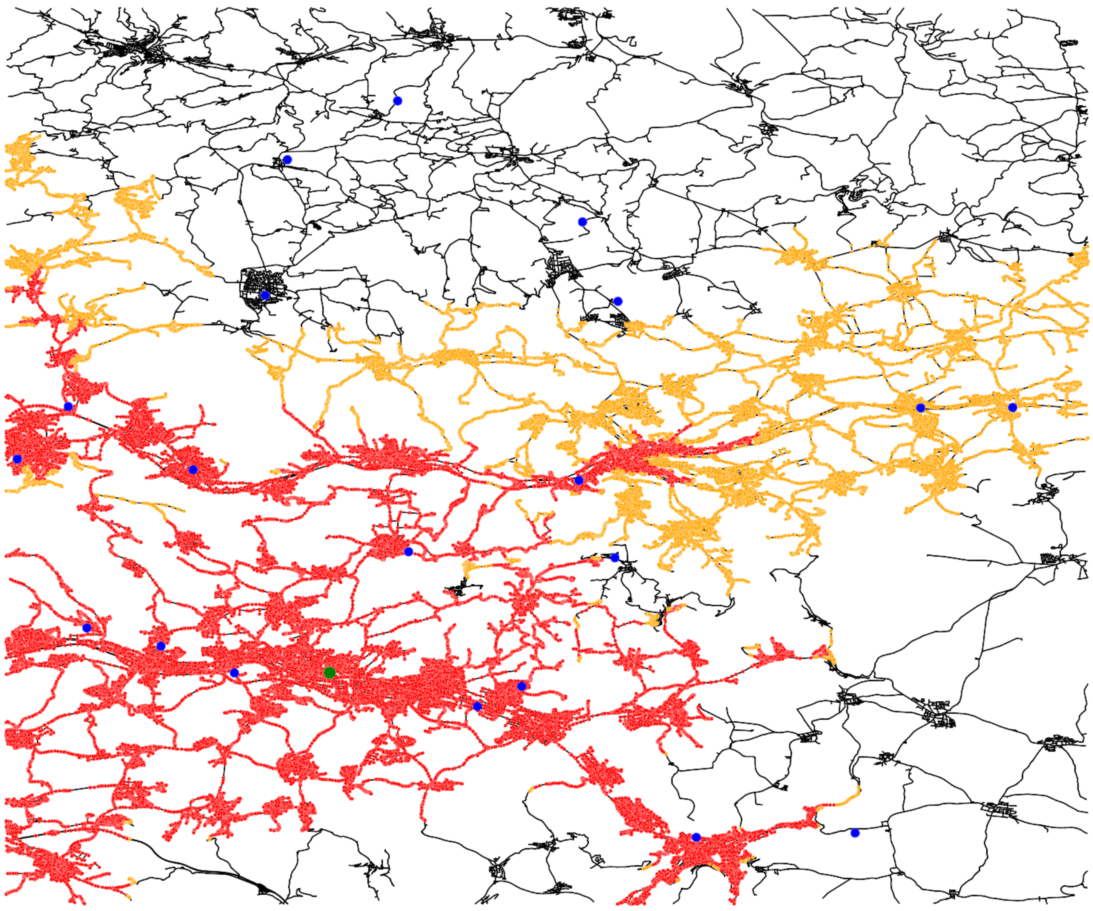


Figure 8.4.: Augmented reachable node set for a source node (green). LSs are marked blue. Red: nodes directly ev-reachable; Orange: nodes indirectly ev-reachable.

the nodes in Southern Germany on average, when starting fully charged at a randomly chosen source. Our approach yields query times below a second for computing connected node sets. This is about factor 10^5 better than the naive way of checking for each node in $R(s)$ if there exists a feasible path back to s, although we speed-up these computations by using contraction Hierarchies as described in [EFS11]. Next we implemented the naive approaches as well as our

	TAU		SG	
	time(sec)	% nodes	time(sec)	% nodes
R(s)	0.00431	98.3	0.41872	19.0
C(s) naive	4.64245	98.3	96279.3	13.9
C(s)	0.00924	98.3	0.92853	13.9

Table 8.1.: Query times for computing ev-reachable and strongly ev-connected node sets; percentage of covered nodes; averaged over 1000 random queries.

new strategy for computing augmented sets of ev-reachable and strongly ev-connected nodes in the presence of loading stations. In Figure 8.4 all directly and indirectly ev-reachable nodes for a random selected source and twenty randomly placed LSs are depicted. We can see in the lower part of the picture, that the ev-reachable LSs enlarge the number of ev-reachable nodes only slightly, while in the upper part the number increases significantly, resulting in a third of all ev-reachable nodes being only reachable via LSs. This is again due to the structure of the

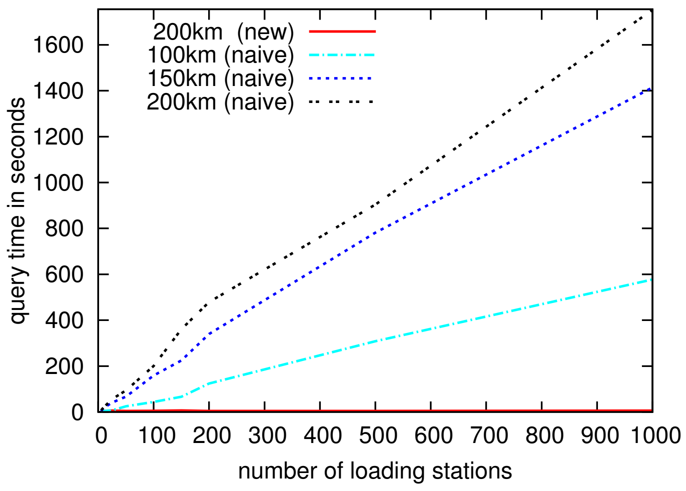


Figure 8.5.: Query times for computing strongly ev-connected node sets in the presence of LSs. Each plot line corresponds to a specific battery capacity, allowing to travel the given range on average.

underlying terrain. If a LS is close to the given source and the path from the source to the LS has no uphill character, we arrive at the LS almost fully charged anyway. The same holds true for LSs that are further away, but the path allows to recuperate most of the used energy. On the other hand, if a LS can only be reached on a very energy-consuming path, we expect its ev-reachable nodes to be really different to the ones of the source, especially if the LS allows to cross a peak.

The query times for the two approaches evaluated on the map of Southern Germany are shown in Figure 8.5. The timings are averaged over 10 rounds of randomly placing the given number of LS in the map and 100 subsequent queries for each. The query times for the naive strategy – computing $C(l)\forall l \in L$ on demand – grows dramatically with the number of LS as well as the maximal battery charge status and hence is only practicable for a very small number of LSs and small cruising ranges. In contrast to that the runtime for the BFS based approach only grows slightly with increasing battery capacity, namely from 1.14-4.06 for 100 km up to 4.46-6.21 seconds for 200 km.

Finally we measured the runtime of one-to-one queries making use of LSs in the map of Southern Germany. The results can be found in Table 8.2. Again the LSs were placed randomly in the street network. We computed the path, that required the minimal number of stops at loading stations. As expected the total query times are very similar to that for computing the set of reachable nodes, therefore we also achieve practicable runtimes below one second for answering one-to-one queries. In Figure 8.6 one can see two examples of such paths. In both cases the target would not have been ev-reachable from the source without LSs. In the upper image, only one visit of a LS is necessary, in the lower image two recharging events occur. The paths are piecewise energy-optimal, hence avoiding going uphill as far as possible, following the valleys in the area. In both examples the paths reveal only small detours to visit the LSs and hence these routes seem to be useful in practice.

cruising range	# LS	no path (%)	direct (%)	indirect (%)	query time (sec)
80km	200	60.2	8.4	31.4	0.17
100km	100	38.0	19.4	42.6	0.32
100km	200	20.9	19.3	59.8	0.32
125km	50	10.8	35.6	53.6	0.47
125km	100	8,8	37.6	53.6	0.45
150km	50	8.0	52.8	39.2	0.53
150km	100	5.9	51.7	42.4	0.51

Table 8.2.: Query times for one-to-one queries dependent on the number of LSs and the possible cruising range. Moreover we recorded the percentages of paths, where the target is not ev-reachable from the source (no path), where the target is ev-reachable from the source without having to visit a LS (direct) and where LSs are necessary to receive a feasible path (indirect). All values are averaged over 1000 random queries.

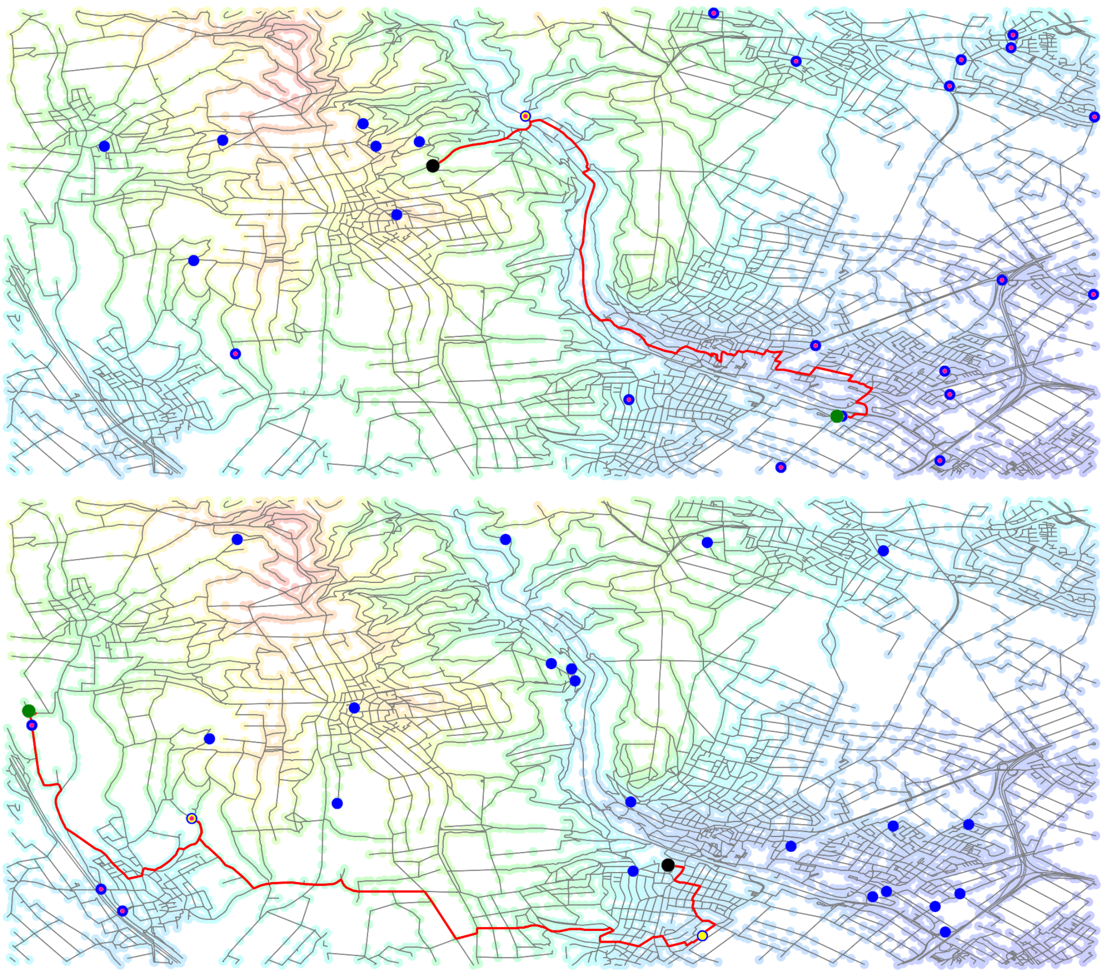


Figure 8.6.: Example paths (red), where recharging is necessary to reach the target(black) from the source(green). LSs are indicated by blue marks. All LSs, that are ev-reachable from the source are marked pink and those actually selected for recharging are also coloured yellow. The desaturated node colours indicate elevation, ranging from 99 meters (deep blue) up to 412 meters (red).

9. Constrained EV-Paths

It is unlikely that people are willing to accept considerably longer travel times just to save a few kWh of energy. The techniques presented so far do not guarantee anything in terms of travel time or distance. Of course, we already dealt with constraints, namely the battery constraints. But note that only one metric – energy consumption – was involved up to now. So the constraints and the objective were both concerned with the same metric, while incorporating travel time or distances we introduce a second metric to the problem. Minimizing one metric while putting a constraint on the other is an instance of the constrained shortest path problem (CSP), which is NP-hard in general. Therefore we can no longer hope for a polytime algorithm like for the previously studied problems. Nevertheless efficient query answering might be possible in practice. Our basic idea is transferring speed-up techniques originally developed for the conventional shortest path problem in order to reduce query time (and space consumption) for CSP queries.

In recent years several speed-up techniques, developed originally for the one-to-one shortest path problem, found their way into more sophisticated applications. For example, it was shown that SHARC [BD09] is also very useful to identify pareto paths [DW09], contraction hierarchy can be used for route planning scenarios with flexible objective functions [GKS10] and both can be applied to networks with time-dependent edge costs [Del08], [BDSV09]. All these papers, except [DW09], are concerned with routing problems, which can be solved efficiently in *polynomial* time, and query answering is always based on Dijkstra's algorithm (or a modification thereof).
One method to solve CSP exactly – the *label setting algorithm* – is based on the same concept as Dijkstra's algorithm. Like in Dijkstra's algorithm, labels get stored in a priority queue and every time we extract the label with minimal key value and perform an edge relaxation step. In addition, both algorithms allow for storing predecessors along with the labels and hence the optimal path can be found efficiently by backtracking. Moreover, like Dijkstra's algorithm, the label setting can be performed in a bidirectional manner. Due to these similarities, Dijkstra-based speed-up techniques like CH and arc-flags seem to be applicable to the label setting algorithm. As these preprocessing methods extract a sparse subgraph for query answering without compromising optimality, they promise significantly improved run time as well as space consumption during a label setting computation.

Note that for conventional CSP instances both involved metrics are represented as constant costs along the edges. Using our energy consumption edge cost functions as one of the metrics complicates the setting. Therefore we will first study a slightly simplified model, where we minimize travel time or distance while setting an upper bound on the positive height difference of the path. As seen in the last sections energy consumption correlates strongly with the number of meters driven uphill, therefore solutions for such queries might be also of practical use, especially if the maximal battery capacity or the actual load level are unknown. Based on that we highlight the necessary modifications to make our techniques work when the energy consumption functions are applied. Moreover we will also take loading stations into account and present methods to regulate or minimize the number of necessary recharging events if additional constraints on distance or time play a role.

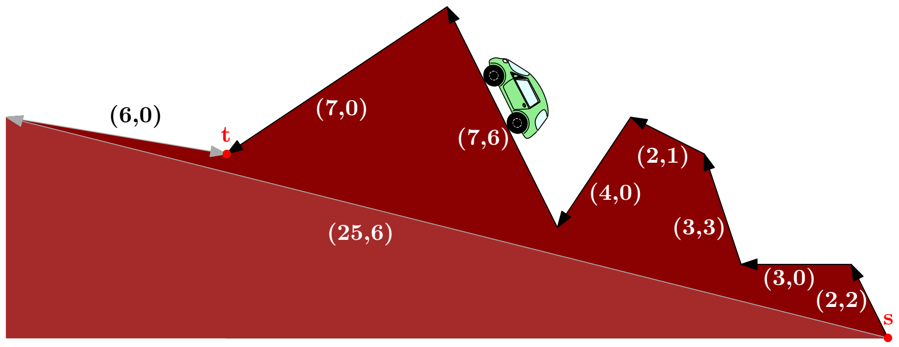


Figure 9.1.: Example of a CSP-instance. Tuples contain the length and the positive height difference of an edge. While the upper path (black) from s to t is shorter, the lower path (gray) bears less climbs. Hence it depends on the choice of the resource bound, which of them is declared optimal.

9.1. Avoiding Hard Climbs

Large detours are not desired when planning a trip with the EV. On the other hand, one is certainly willing to ride a few extra kilometres if this saves hard climbs and therefore reduces energy consumption (see Figure 9.1 for a small example). This gives rise to two natural optimization problems:

1. *Find the route from A to B with the least (positive) height difference (summed over all segments) which has length at most D.*
2. *Find the route from A to B which is shortest among all paths which have height difference of at most H.*

In practice one might choose the distance limit D for example as 1.2 times the shortest path distance, or the height difference limit H as 1.5 times the path of minimal height difference (both of which can be easily computed using standard shortest path algorithms).
More formally, we are given a digraph $G(V, E)$, a cost function $c : E \to \mathbb{R}_0^+$ and a resource consumption $r : E \to \mathbb{R}_0^+$ on the edges. For our application, distances equal the Euclidean distance between two nodes; to retrieve the positive height difference we are also given an elevation function $h : V \to \mathbb{Z}$. Based on that the resource consumption of an edge $e = (v, w)$ can be defined as $r(e) = \max(h(w) - h(v), 0)$.

In a query we are given the source and destination node, s and t, as well as the budget or resource bound R. The goal is to determine the minimal cost path from s to t, whose resource consumption does not exceed R. With $c(p) = \sum_{e \in p} c(e)$ and $r(p) = \sum_{e \in p} r(e)$ we refer to the cost and resource consumption of a path p. We say, that a v-w-path p dominates another v-w-path p' if $c(p) \leq c(p')$ and $r(p) \leq r(p')$ and at least once the inequality is strict. We call a v-w-path p pareto-optimal, if there exists no dominating path for p. The set of all pareto-optimal paths between a source and a target node equals the set of all optimal CSP solutions (for any possible resource bound). Note, that their number might be exponential in the input size.
Exact algorithms for CSP – like the label setting (LS) approach or dynamic programming (DP) – have to (partly) explore all of these paths to find the optimal one. Therefore obtaining an exact solution is normally both time- and space-intensive. If exact algorithms are prohibitive for practical use, approximation algorithms – like binary search [AMO93] – are often used instead. The binary search algorithm provides approximate CSP solutions by restricting the search to a subset of all pareto-optimal paths, namely the ones forming the so called *lower convex hull (LCH)*. The lower convex hull can be illustrated as follows: Let $c(p)$ be the costs of a path p and

$r(p)$ the respective resource value. The tuple $(c(p), r(p))$ can be represented as line segment $\lambda c(p) + (1 - \lambda)r(p)$ with $\lambda \in [0, 1]$. A v-w-path p is part of the lower hull if there exists a $\lambda \in [0, 1]$ for which $\lambda c(p) + (1 - \lambda)r(p)$ is minimal among all v-w-paths. Paths on the lower convex hull have the advantage of being computable for fixed λ with a conventional Dijkstra run in $G(V, E)$ with edge costs $\lambda c(e) + (1 - \lambda)r(e)$ (we refer to the respective graph also as G^λ). In the binary search algorithm λ-values are chosen consecutively to iterate over the slopes of the line segments forming the lower convex hull. Note, that not only the number of *all* pareto-optimal paths might be exponential but also the number of paths on the lower convex hull. Nevertheless the binary search algorithm is guaranteed to find an approximate solution for discrete costs and resource consumptions in polytime by halving the interval of possible slopes in every round.
In this section we show that Dijkstra-based speed-up techniques can also be used to reduce the query time and space consumption for instances of CSP *without compromising the optimality of the solution*. We will describe in detail how CH can be modified to maintain all pareto-optimal solutions in a given street network. We propose several approaches to decide whether a shortcut is necessary, allowing to trade reduced preprocessing time for graph sparseness. Here a crucial ingredient for accelerating the CH-graph construction is a witness search procedure that operates in a binary search like manner on paths on the lower convex hull only. Apart from answering queries in the CH-graph with the label setting algorithm, we also outline how the dynamic programming approach can take advantage of CH. Additionally, we introduce a modified version of arc-flags that allows to speed up CSP queries alone or in combination with CH.

9.1.1. Pruning with Contraction Hierarchy

To reduce the runtime, we want to exclude as many nodes and edges as possible a priori and hence thin out the search space for the labelling algorithm. The simple pruning technique 1.3.4 might be useful here, as all nodes v with any path $s, \cdots, v, \cdots, t$ having a resource consumption greater than R get removed. Note that the impact of this technique is low if either R is very large or many edges with a small resource consumption exist. Unfortunately, latter is the case in our application because every downhill or flat edge will receive a positive height difference of 0. Therefore we aim for more effective pruning strategies. In the following we will show how the speed-up techniques CH and arc-flags can be adapted to our scenario with resource constraints and study their performance in pruning nodes and edges for our CSP-queries.

Node Ordering

A crucial ingredient to construct a CH-graph is fixing the order in which the nodes are contracted. In our case nodes seem more important if they belong to cost-optimal paths with low total resource consumption. Therefore we also use a weighted edge difference, but break ties by first contracting nodes whose incoming edges have a high resource consumption.

Witness Search for Constrained Shortest Paths

Like every subpath of a shortest path has to be a shortest path itself, every subpath of a pareto-optimal path has to be pareto-optimal. Hence we have to maintain all pareto-optimal paths on a local level to guarantee correct query answering in the CH-graph. Therefore, while contracting a node v, we can avoid adding a shortcut between two of its neighbours u and w only if we find an alternative path from u to w which dominates the reference path $p = uvw$.

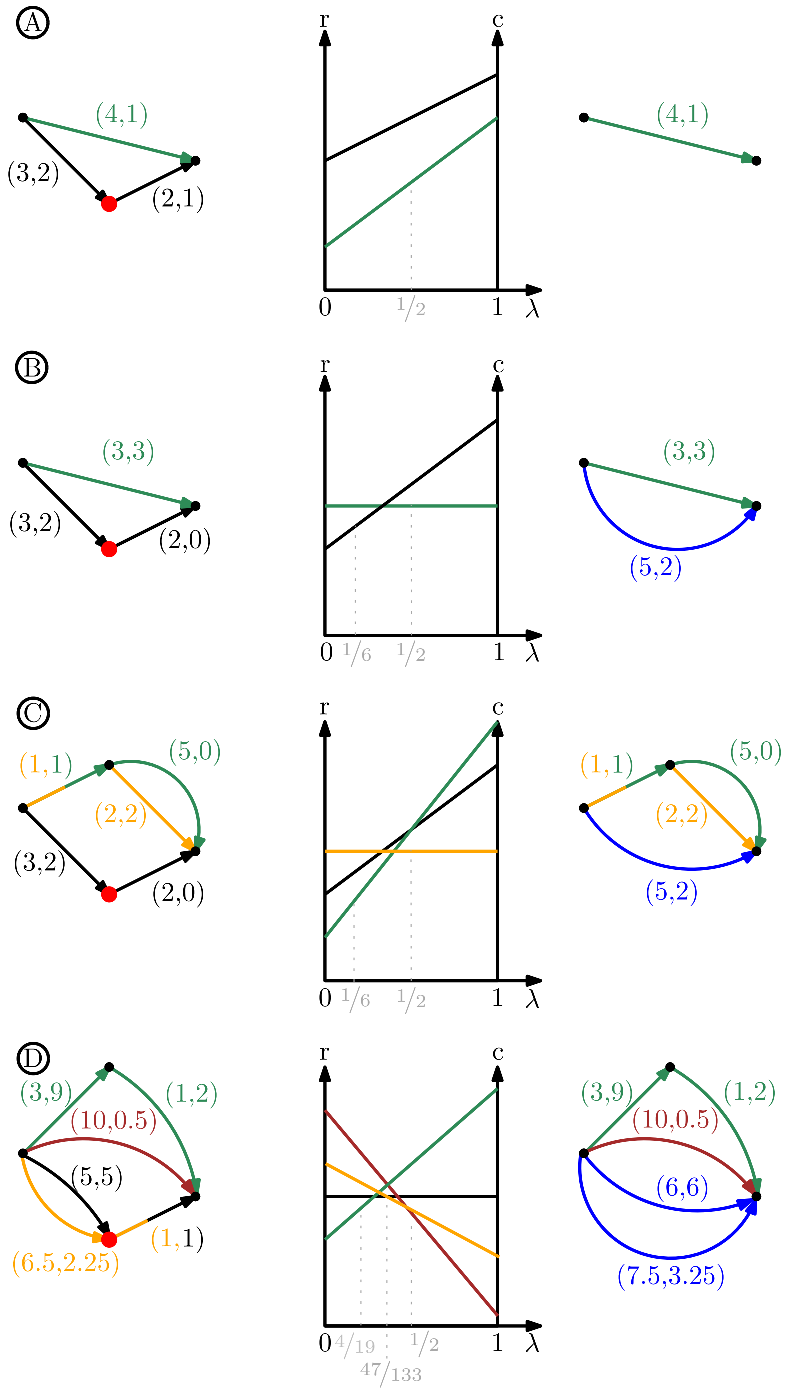


Figure 9.2.: Examples for contracting a node (large red mark). The resulting graph is given on the right. In (A) the reference path (black) is dominated by the green path. The latter describes the complete LCH, therefore a single Dijkstra run in G^λ with $\lambda = 0.5$ is sufficient to make sure that the respective shortcut for the black path can be omitted. In (B) the green path does not dominate the black one. Instead, the reference path is a part of the LCH, as the second Dijkstra run with $\lambda = 0.1\overline{6}$ will reveal; hence the shortcut must be inserted (indicated by the blue edge on the right side). In (C) the shortcut is needed as well, but the exploration of the LCH will be inconclusive as the reference path is neither a part of the LCH nor dominated by the ones that are. In (D) the black path is again a part of the LCH, but – as indicated by the gray lines – three check points are not enough to detect it. Hence our checker needs more support points for a conclusive result.

A straightforward way to find such a dominating witness is starting a label setting computation (LSC) in u with a resource bound $R = r(p)$. As soon as w pops out of the PQ the first time, we stop the LSC and check if the label's cost is below $c(p)$. If this is the case, p is dominated by another path and the shortcut can be omitted. Otherwise (if the label equals $c(p)$) the path p is pareto-optimal and hence the shortcut $sc = (u, w)$ with $c(sc) = c(u, v) + c(v, w)$ and $r(sc) = r(u, v) + r(v, w)$ is needed for sure. Of course, we can already abort the LSC if a label is assigned to w that dominates the label of the reference path. Note that for conventional shortest paths there can be at most one shortcut (v, w) for any pair of nodes $v, w \in V$ in G'. But in our scenario there might be as many shortcuts as there are pareto-optimal paths in G between v and w.
Unfortunately, LSC might be too time-consuming to apply to every pair of neighbouring nodes in every contraction, even if we apply simple pruning first. Because of that, we now propose a procedure which can help to avoid some of these computations:

The basic idea is to first restrict ourselves to paths on the lower convex hull (LCH) of all paths. If p is a part of the LCH, p has to be pareto-optimal and therefore no dominating path exists. Hence we have to insert the shortcut $sc = (u, w)$. On the other hand if there is a dominating path p', it is likely (not required!), that $p' \in LCH$. To get candidate paths on the LCH we use the weighting parameter $\lambda \in [0, 1]$ and determine optimal paths in G^λ where the edge costs equal $\lambda c(e) + (1 - \lambda)r(e)$. In G^λ we can identify an optimal path between two nodes using plain Dijkstras algorithm. If we extract this path and evaluate it in G we can easily check if this path dominates p or is equal to p. Otherwise we can retry using another value of λ.

In practice we want to restrict ourselves to a small set of support points $\lambda_1, \cdots, \lambda_t$. Of course, we could choose these values randomly or sample the interval $[0, 1]$ uniformly, but in both cases many λ_i might lead to the same resulting path. We can do better if we search for the support points systematically.

The basic idea of our approach is somewhat similar to the binary search algorithm, because in every step we will at least halve the interval for the considered parameter. However in contrast to conventional binary search, we do not want to compute a certain path on the lower convex hull but want to check if our reference path $p = u, v, w$ is part of the LCH or not. To that end we start with $\lambda = 0.5$ and compute the respective optimal path $p_{0.5}$. If $p = p_{0.5}$ we know that p is contained in the LCH and therefore the shortcut is necessary. If instead $p_{0.5}$ dominates p, we found a witness and so the shortcut can be omitted for sure. In both of these cases the witness search is already conclusive and we are done. If both cases do not apply, there must exist an intersection point S of the line segments representing p and $p_{0.5}$ for some $\lambda \in [0, 1]$. This λ-value splits the interval $I = [0, 1]$ in two continuous subintervals I_1, I_2; with w.l.o.g. $p_{0.5}$ being below p in I_1 and vice versa in I_2. For any $\lambda \in I_1$ we already know that p will not be the minimum cost path, as $p_{0.5}$ dominates p in this interval. Hence it is sufficient to continue the witness search in I_2 only. Observe that I_2 cannot cover more than the half of I, because at $\lambda = 0.5$ the path $p_{0.5}$ is below π for sure. Now we can repeat this process by selecting a new λ support point as the center of the interval I_2 and determining a new witness candidate path. This procedure always maintains an interval in which the reference path is still 'alive', i.e. is below all previously identified paths. If this interval runs empty we know that the reference path is not part of the LCH, but it still might be pareto-optimal. Hence the result is inconclusive in that case. But if we identify p as part of the LCH or find a single dominating path during the search process, we can definitely decide whether the shortcut is needed or not.

Observing that for our application the resource consumption of an edge is always bounded in the costs (because one cannot drive more meters uphill than total), we can prove a polynomial runtime of $\mathcal{O}(\log(nC)(n \log(n) + m))$ with $C = \max_{e \in E} c(e)$ for this part of the witness search, see lemma 9.1.

Lemma 9.1. *The witness search restricted to paths on the lower convex hull needs time* $\mathcal{O}(\log(nC)(n\log(n)+m))$.

Proof. Let p_0 be the reference path, p_i be the last found path on the LCH so far, and λ_i the respective support point. Consider the triangle spanned by the intersection point of p_0 and p_i and their intersection points with the line $\lambda = 1$, i.e. $(1, c(p_0))$ and $(1, c(p_i))$. Furthermore let l be a line passing through the point $(1, {}^{(c(p_i)-c(p_0))}/_2)$ and the intersection point of p_0 and $\lambda = \lambda_{i+1} = {}^{1-lambda_i}/_2$. Obviously this line is parallel to and below the line segment representing p_i. Therefore l is in particular below the line segment for p_i evaluated at λ_i. Now observe that any l' with higher costs than l or a smaller function value at λ_{i+1} would have an even higher slope than l and so would be below the line segment of p_i at λ_i as well. Hence for a new found path p_{i+1} which is optimal for λ_{i+1} it must yield $c(p_{i+1}) \leq {}^{(c(p_i)-c(p_0))}/_2$, because otherwise p_{i+1} would already have been identified as p_i in the previous step. Therefore with every iteration the potential cost interval is halved. As the maximal path costs can be described as nC and paths have integer costs, there are at most $\log(nC)$ search steps possible of which each requires a single Dijkstra run in G^λ, and the computation of the next λ-value (if necessary), which can be performed in constant time. □

Moreover, we can fix a maximal value for t, restricting the number of possible Dijkstra computations a priori. Figure 9.2 shows some examples that illustrate the connection between the witness search and the check procedure.
Of course, the Dijkstra computations in G^λ can also be sped up by simple pruning. To that end we just have to store and update the cost and resource consumption for a node simultaneously with the transformed cost label. Observe, that the simple pruning is independent of the choice of λ and hence has only to be done once for every pair of nodes.

If the checker does not provide a conclusive result, we can either start the LSC on top, or add the shortcut $sc(u, w)$ without further investigations in order to reduce the preprocessing time. Of course, without the LSC we might add some superfluous shortcuts. Note that this does not compromise correctness, but can lead to worse query times and increased space consumption.

Observe that in the resulting CH-graph G' – consisting of all original nodes and edges as well as all shortcuts – we can now answer *both* kinds of queries mentioned in the introduction more efficiently, i.e., we can either restrict the distance and ask for the path with the minimal positive height difference or we could also restrict the latter and compute the shortest path fulfilling the height constraint. Note, that this is a direct consequence of omitting shortcuts only if a dominating path can be found.

Answering Queries

To answer an s-t-query in the constructed CH-graph, we only have to consider edges that lie in the upward graph $G^\uparrow$ induced by s or in the downward graph $G^\downarrow$ induced by t *and* are adjacent to a feasible node according to the simple pruning with resource labels. Therefore we can also speed up the required Dijkstra computations for the simple pruning by considering only edges in $G^\updownarrow = G^\uparrow \cup G^\downarrow$. If t does not receive a feasible resource label in this step, there exists no path from s to t fulfilling the resource constraint and we are done. Note that, as a nice side effect of simple pruning in $G^\updownarrow$, all nodes that do not lie on any path from s to t receive a resource label of ∞ automatically. Hence they will not be considered anymore, pruning the search space additionally. We also run a (bidirectional) Dijkstra computation in $G^\updownarrow$ from s to t, now considering

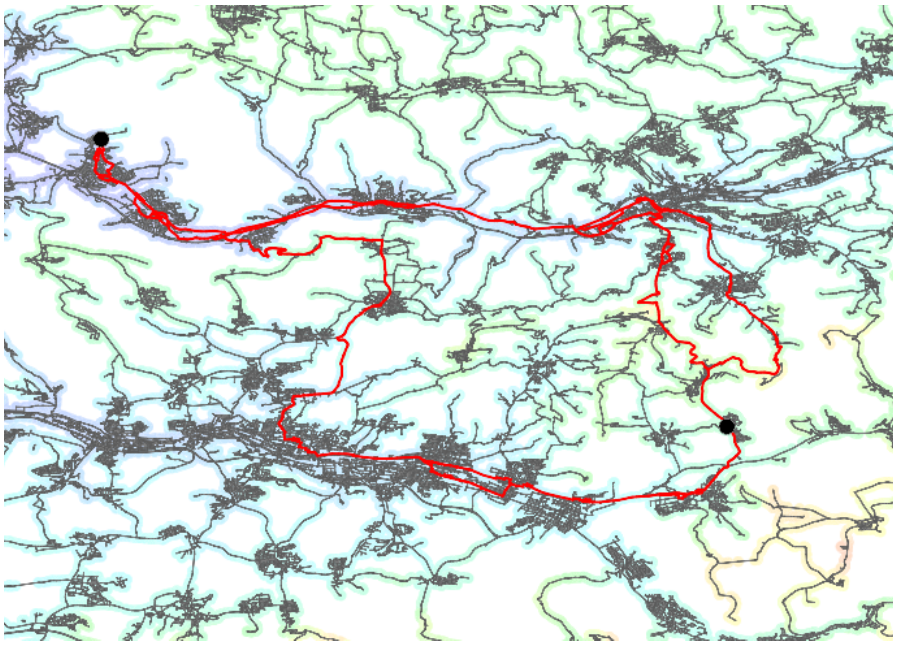


Figure 9.3.: Pareto-optimal paths (red) between source (top left) and destination node (bottom right). Node colours indicate elevation (from blue-243 m up to red-786m). While there are 87 different pareto-optimal solutions, the total number of used edges is rather small.

the edge costs. If the resulting path is feasible wrt to its resource consumption, we found the optimal solution straight away. Otherwise further computations are required:
The *bidirectional* version of the *LS* can be easily modified to take advantage of CH. The only difference is that the forward run from s considers only upward edges, while the backward run from t considers only downward edges. Of course we do not push any nodes into the PQ, that were declared infeasible in the simple pruning step.

For combination with other speed-up techniques it might be desirable to answer queries in a *unidirectional* manner. For that purpose we start a breadth-first search (BFS) at t to mark all downward incoming edges recursively, omitting edges that are adjacent to infeasible nodes. Afterwards we can start a conventional LSC at s that runs on upward and marked edges. The first time t is popped out of the PQ, the optimal label is found.

Up to now we only outlined how CH can be useful to answer queries with LS more efficiently. But we can also use this preprocessing to reduce space consumption and runtime for the *dynamic programming* approach. To that end we perform a BFS not only from t on downward edges, but also from s to mark all possible upward edges. Afterwards we extract the subgraph by keeping only marked edges and their adjacent nodes (if they are feasible).

Extension: Pruning with Arc-Flags

The basic assumption to make arc-flags work is that the set of edges which connect any points of two far away regions is not too large. For CSP the number of edges on pareto-optimal paths between two regions is surprisingly small (see Figure 9.3), but of course larger than the number of edges on shortest paths only. In order to further reduce the number of relevant edges for a query, we introduce the concept of resource-dependent arc-flags. To that end we divide the range of possible bounds R into intervals $[0, R_1], (R_1, R_2], \cdots (R_k, \infty]$. Now we split the edge flag for

a certain partition P_i in k flags. Such an interval flag for $(R_{i-1}, R_i]$ is true iff a pareto-optimal path to a node $t \in P_i$ starts starting with this edge has a resource consumption of at most R_i. Using again only boundary nodes to determine arc-flags, we set all interval flags for the partition that contain the edge to true.
Answering a query changes slightly: now we consider only edges $e = (v, w)$ if the resource consumption of the actual label assigned to v lies in an interval flagged with 'true' for the target's partition.
Also, arc-flags and CH can work together; combining them promises an even smaller search space and therefore better query times. Computing the CH first, and assigning arc-flags to all edges including shortcuts afterwards is very time-consuming, because adding shortcuts also increases the number of nodes that are located on partition boundaries. Moreover, the edge flags of shortcuts can be easily derived if the edge flags for the original graph are already known. Normally, a shortcut that skips two edges e_1, e_2 can only have a true edge flag for partition P_i if both respective edge flags of e_1 and e_2 are true. Therefore, conventional arc-flags for shortcuts can be derived by applying the bitwise and-operator to the vectors of edge flags for e_1, e_2. Using resource-dependent arc-flags we set the shortcut's interval flag for $(R_{i-1}, R_i]$ to true if the according edge flag of e_1 is true and the flag of e_2 for the interval containing $R_i - r(e_1)$ is true as well.

9.1.2. Experimental Results

In this section we evaluate the impact of the introduced speed-up techniques on real-world instances. We used five test graphs (TAU,SH,WIN,BW,SG), which are all subnetworks of the street graph of Germany. Distances and elevations were used with a precision of $1m$. The average path lengths and the average positive height differences for all test graphs can be found in Table 9.1. Preprocessing times were measured on a single core of the Intel Core i3-2310M CPU with 2.10GHz and 8 GB RAM.

	TAU	SH	WIN	BW	SG
shortest path					
avg length	5618	27606	52402	79599	184634
avg height diff	189	715	1164	1677	4038
minimal height difference path					
avg length	6925	30938	70657	98333	287461
avg height diff	116	553	766	1179	2116

Table 9.1.: Characteristics of the test graphs: Average path length as well as height difference for shortest paths and for paths with minimal height difference. All values are given in meters and are averaged over 1000 random queries.

We preprocessed all the test graphs using contraction hierarchy – once in the conventional way, considering only costs and aiming to maintain all shortest paths (SP), and once using the new variant for CSP. For the latter we used the described check procedure (considering first paths on the lower convex hull) with three support points. Based on that we could avoid about 62% of the local label setting computations. We restricted ourselves to contracting 99.5% of the nodes in each graph, in order to achieve reasonable preprocessing times for the CSP-CH (4 seconds for the 10k graph, about 2h for the SG graph). Edges between uncontracted nodes were declared downward edges. The total number of edges in the final CH-graphs can be found in Table 9.2 for all five graphs. Surprisingly, the total number of edges in the CSP-CH graph is only about $3 - 10\%$ larger than the number in the SP-CH graph. Hence we also have only about twice the number of original edges in the CH-graph. This might partly result from the fact that the positive

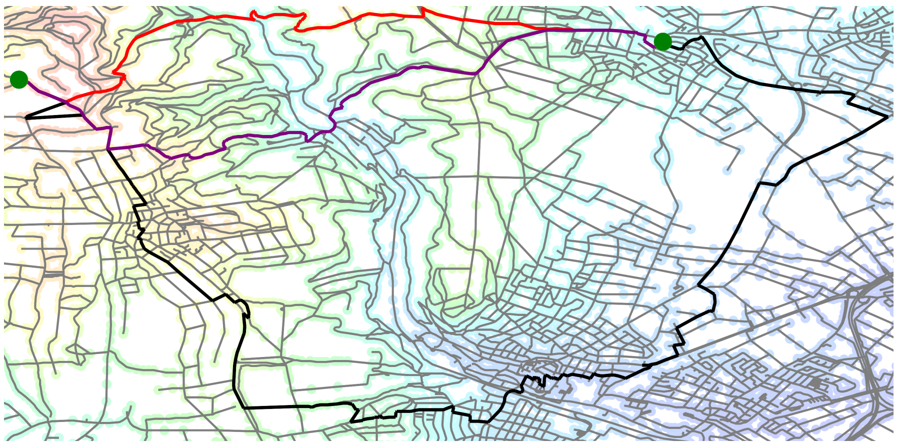


Figure 9.4.: Example of a shortest path (top, red, distance 7.5km, height diff. 517m) and the respective path with minimal height difference (bottom, black, distance 19.1 km, height diff. 324m), which makes a very large detour. The shortest path under the constraint of having a height difference smaller than $1.5 \cdot 324$m $= 486$m is a fair compromise (middle, purple, distance 7.7km, height diff. 410m).

height difference is bounded in the path length, as one cannot achieve to gain 10 meters in height without riding for at least 10 meters at the same time. Moreover, the large number of edges with zero resource consumption is beneficial here, as finding a witness for a path without any resource consumption simply reduces to finding a path with smaller costs, which is equivalent to the conventional scenario. We measured query times and polls for the unidirectional label

		edges		
graph	nodes	original	SP-CH	CSP-CH
TAU	11220	24119	50641	54383
SH	100242	213096	407034	419210
WIN	500011	1074458	2090628	2160438
BW	999591	2131490	4150204	4276478
SG	5588146	11711088	23970043	26586530

Table 9.2.: Number of nodes and edges for the used test graphs. 'Original' describes the number of edges in G before applying CH. The last two columns show how conventional CH for shortest paths (SP-CH) and CH for CSP augments the set of edges.

setting algorithm. At first we picked a source and a target vertex s, t randomly, and computed the path with the minimal positive height difference H. Then we started a LSC with a resource bound of $1.5H$ and aimed for the shortest path fulfilling this constraint (see Figure 9.4 for a an example in the TAU graph).

The results can be found in Table 9.3. In all test graphs the number of edges in the subgraph of the CH-graph induced by s and t is at least one order of magnitude smaller than in the original graph. With the only exception of the SG graph, applying CH only reduced the number of edges far more than the simple pruning approach. Nevertheless, in all cases the queries in the CH-graph could be answered with significantly less poll operations than in the feasible subgraph based on simple pruning. This is also reflected in the runtime. While pruning with resource labels only halves the runtime needed to answer a query in a completely unpreprocessed graph, the respective CH-query can be answered two orders of magnitude faster. The combination of both techniques leads to query times below 1 second in graphs with a size up to $5 \cdot 10^5$ nodes. The reason why for the SG graph we achieve only a speed-up of 20 using CH (50 for the com-

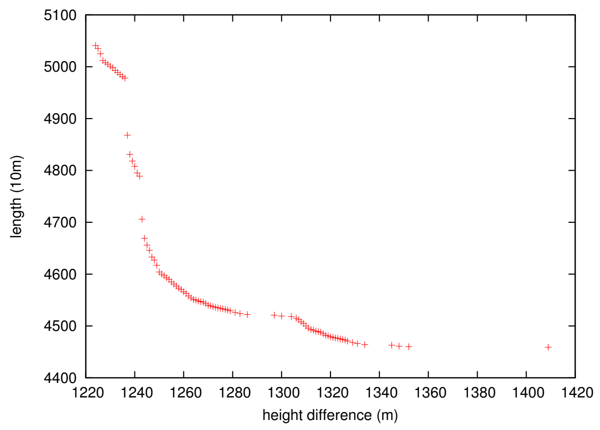


Figure 9.5.: All 91 pareto-optimal labels (consisting of path length and corresponding height difference) assigned to a single node during the label setting algorithm in the SH graph.

bination of CH and simple pruning) might be the incomplete contraction of the nodes during preprocessing, because all of the edges between uncontracted nodes have to remain in the induced subgraph for answering a query. Moreover the runtime of a LSC not only depends on the number of polls, but also on the time required to check the pareto-optimality of a label and prune dominated ones previously assigned to the respective node. The costs of these operations increase, of course, with the number of labels that are already assigned to a node. As one can see in Figure 9.5 for the SH graph this number might be very large in our scenario.
Nevertheless, the reduction in the graph size was significant and allowed us to answer all queries using only 8GB of RAM. In contrast, using the naive approach, part of the queries in BW and SG failed (these were excluded from timings). Even after switching to a server with 96GB RAM we ran out of memory for some queries in the SG graph. With our new techniques, the search space reduces remarkably, and therefore enables us to handle such queries on desktop computers or even laptops.

As outlined in the previous section, we can also take advantage of the CH-graph when answering queries using dynamic programming. Here we first have to extract the feasible subgraph to build a smaller dynamic programming table. The numbers of nodes that remain in this subgraph (and therefore determine the dynamic programming table's width) are given in Table 9.4. For all considered graphs the nodes are less than 1% of the original nodes, leading to an overall reduction in table size of at least two orders of magnitude. Therefore, the average CH-table size for the SG graph is now comparable to the original one for the SH graph. Accordingly, we could again answer all queries using only 8GB of RAM, while no queries in the BW and SG graph could be answered at all in the original setting. The query times, given also in Table 9.4, show a speed-up by a factor of $10 - 30$, though the runtime for larger graphs is prohibitive for practical use, taking about one hour for a query in the SG graph. But note that path lengths were computed with a very high precision of one meter, and as the value of the optimal solution determines the second dimension of the dynamic table, this leads to a large number of table rows.

Finally we evaluated (resource-dependent) arc-flags in the CSP setting. Due to very long preprocessing time and high space consumption, we restricted ourselves to measurements on the TAU and SH graphs.
First we divided the TAU graph into 64 partitions (using a uniform 8x8 grid). We achieved a speed-up similar to CH (without simple pruning), but used about ten times the preprocessing time. Secondly, we applied resource-dependent arc-flags to the SH graph, using 12 partitions

graph	subgraph edges naive	naive+p	CH	CH+p
TAU	$2.4 \cdot 10^4$	$7.5 \cdot 10^3$	$2.8 \cdot 10^3$	$1.2 \cdot 10^3$
SH	$2.1 \cdot 10^5$	$1.0 \cdot 10^5$	$5.2 \cdot 10^3$	$2.9 \cdot 10^3$
WIN	$1.1 \cdot 10^6$	$4.6 \cdot 10^5$	$3.6 \cdot 10^4$	$1.2 \cdot 10^4$
BW	$2.1 \cdot 10^6$	$5.3 \cdot 10^5$	$8.1 \cdot 10^4$	$3.4 \cdot 10^4$
SG	$1.2 \cdot 10^7$	$7.0 \cdot 10^5$	$1.8 \cdot 10^6$	$1.6 \cdot 10^5$

graph	number of polls naive	naive+p	CH	CH+p
TAU	$3.6 \cdot 10^4$	$1.2 \cdot 10^4$	$4.9 \cdot 10^2$	$2.3 \cdot 10^2$
SH	$1.4 \cdot 10^6$	$6.5 \cdot 10^5$	$8.1 \cdot 10^3$	$4.9 \cdot 10^3$
WIN	$8.9 \cdot 10^6$	$6.7 \cdot 10^6$	$8.9 \cdot 10^4$	$7.4 \cdot 10^4$
BW	$2.9 \cdot 10^7$	$6.8 \cdot 10^6$	$3.4 \cdot 10^5$	$9.1 \cdot 10^4$
SG	$9.2 \cdot 10^7$	$3.7 \cdot 10^7$	$1.7 \cdot 10^6$	$8.7 \cdot 10^5$

graph	query time (s) naive	naive+p	CH	CH+p
TAU	0.0233	0.0117	0.0015	0.0002
SH	7.5942	2.7301	0.1294	0.0168
WIN	123.0791	105.7814	1.0438	0.9895
BW	265.6990	117.6859	5.6712	2.5879
SG	2369.3361	1131.1603	124.2088	64.2364

Table 9.3.: Experimental results for answering queries with the label setting algorithm in the original graph (naive) and the CH-graph, in combination with simple pruning ('+p') and without. All values are means based on 1000 random queries. For all measurements standard deviations range between 1.2 and 1.6 times the average.

	nodes	table size		query time (s)	
graph	CH	naive	CH	naive	CH
TAU	100	$6.5 \cdot 10^7$	$5.6 \cdot 10^5$	13.7	1.9
SH	412	$2.9 \cdot 10^9$	$2.9 \cdot 10^6$	106.2	11.2
WIN	2508	$3.1 \cdot 10^{10}$	$1.5 \cdot 10^8$	3729.1	104.4
BW	4724	$(8.5 \cdot 10^{10})$	$4.2 \cdot 10^8$	-	498.9
SG	21486	$(1.3 \cdot 10^{12})$	$4.6 \cdot 10^9$	-	3573.5

Table 9.4.: Experimental results for the dynamic programming approach. The second column gives the number of nodes in $G^{\updownarrow}$, and hence equals the first dimension of the dynamic table. Query times for the naive approach could not be measured for the graphs BW and SG, because we ran out of memory. Values in brackets are estimates based on the optimal solutions returned by the CH-variant and the number of nodes in the respective graphs. All other values are averaged over 100 random queries.

and three intervals with $R_1 = 400$ and $R_2 = 800$. The preprocessing took about 4 hours and considered 2576 boundary nodes. For high resource consumption bounds the number of polls was reduced by an order of magnitude in comparison to the naive approach. The combination of CH and arc-flags led to an equal number of polls as CH plus simple pruning. In addition with the help of all three of these techniques, the number of polls and the subgraph size could be halved once more.

9.2. Short and Energy-Efficient Paths

Now we want to focus on the scenario where the battery capacity as well as the edge cost functions representing energy consumption are available. So we are faced with instances of CSP where either the costs or the resource consumption are functions instead of constant values. Still the goal is to find paths that are energy-efficient and short or fast at the same time. Naturally optimization problems that arise in this context are

1. Computing the shortest or quickest feasible (in terms of energy consumption) path.
2. Computing the energy-optimal path with bounded distance/travel time.
3. Computing the shortest/quickest feasible path with only k recharging events allowed.
4. Computing the feasible path with a minimal number of recharging events and bounded distance.

For the last three problems the EV driver can input a parameter (bound on distance/travel time or number of recharging events) according to his own preferences. For example, an environment-aware user might accept a path that is 1.5 times as long as the shortest path to save energy, while another one is only willing to accept a path which is not more than 1.2 times this distance.

For the first two problems we will show that again a label setting computation in combination with CH results in practical query times. This will be the basis to solve problems 3. and 4. for which we propose additional preprocessing techniques that involve the construction of auxiliary graphs.

9.2.1. Energy-optimal Paths with Bounded Distance

To prohibit arbitrarily long detours on a path from s to t , we now search for the energy-optimal path which is no longer than D times the shortest path, where D is an input parameter revealed at query time.
So we are faced with an instance of CSP, where the costs are given by a function representing the energy consumption, the resource usage is equivalent to the edge length, and the resource bound is given by D times the length of $\pi(s,t)$. Hence a LSC will assign tuples $(c(p), d(p))$ to the nodes consisting of summed costs and distances. Note that the summed costs equal the difference between initial and actual battery charge status, therefore we could also assign tuples $(b, d(p))$, where the summed costs are replaced with the actual battery load. Here of course the tuple dominates $(b', d'(p))$, if $b \geq b'$ and $d(p) \leq d'(p)$.
Simple pruning can be performed as usual in this case (see Section 1.3.4) by two Dijkstra computations starting at s and t respectively and considering only the distance values.
To speed up the LSC via CH, we first have to clarify how a witness is characterized in this scenario. Of course we can only omit a shortcut (u, w) for a path $p = u, v, w$ if for *any* initial battery load $b(u) = L \in [0, M]$ there exists an alternative path which is not longer than p and leads to at least the same final battery load in w as p. Observe that equivalent to the CH construction for unconstrained energy-optimal paths (Section 7.3), there is no necessity that this is *the same path* for every L. Therefore a witness can be described as set of paths $q_1, \cdots, q_t$ with $minf(b(q_1), \cdots, b(q_k)) \geq b(p)$ and $d(q_i) \leq d(p)\ \forall i = 1, \cdots, k$. To perform the witness search efficiently we choose a set of initial battery loads $L_1, \cdots$ like described in Section 7.3, but start a LSC with resource bound $d(p)$ instead of a simple Dijkstra computation for every $b(u) = L_i$.

9.2.2. Shortest/Quickest Feasible Paths

The energy-optimal path might not always be the desired route. Many owners of EVs can recharge their EV in their garage; therefore when driving home after work, the goal is rather to get there as fast as possible than to maximize the battery load at the destination. Accordingly

we would like to compute the quickest or shortest path that is feasible.
In this scenario the costs are constant values while the resource consumption is described by the energy consumption function. Hence the simple pruning technique has to be adapted. Here we first compute $R(s)$ and $R^{-1}(t)$ and store the respective battery loads b and b_{min} for all contained nodes. All nodes $v \in V$ with $b(v) < b_{min}(v)$ are not part of any feasible path from s to t and hence can be ignored completely (this is especially true for $v \notin R(s) \cap R^{-1}(t)$).
Observe that the CH-graph can be obtained just as described in the last section (or reused), because the definition for a witness proposed there is still valid in this scenario.

Incorporating Loading Stations

The presence of loading stations might affect the shortest feasible route or might be the reason for the existence of a feasible path at all. To incorporate loading stations efficiently, we make the following observations: Firstly an optimal path visits at most $|L|$ (all) loading stations once, because any cycle starting and ending at a loading stations would increase the distance, but could not lead to a higher battery charge status at that vertex. Secondly the pairwise shortest feasible paths between loading stations are not influenced by the choice of s and t; therefore those paths can be preprocessed. These two observations give rise to the construction of an auxiliary graph similar to the one proposed in Section 8.2.1. Again we create a node for every loading station $l \in L$ and edges (l, l'), if there exists a feasible path from l to l' (without the usage of any further loading stations). In contrast to before, we now weigh the edges with the shortest feasible path distance. On query time we have to add the nodes s and t and suitable edges. For this purpose we first compute $R(s)$ and $R^{-1}(t)$ and intersect these sets with L to derive the directly (inverse) reachable loading stations. For every $l \in R(s) \cap L$ we have to compute the shortest feasible path from s to l. Of course, this can be achieved simultaneously for all of the involved loading stations via a single LSC. For t the same is true in the reversed graph using the functions proposed in Section 8.1.2 as resources. After adding the edges $(s, l), l \in R(s) \cap L$ and $(l, t), l \in R^{-1}(t)$ to the auxiliary graph, a single Dijkstra run from v_s to v_t returns the desired sequence of loading stations on the shortest feasible path from s to t.

To benefit from applying CH when computing $R(s)$ and $R^{-1}(t)$ we leave all loading station nodes uncontracted. Therefore they all lie in the upwards graph induced by s and the downwards graph induced by t respectively, i.e. exploring these graphs is sufficient to determine the set of (inverse) reachable loading stations. Moreover this speeds up the computation of the auxiliary graph, because the shortest feasible distances from one LS to all the others can be computed in the remaining (much smaller) graph with a plain label setting computation.

Naturally such a route might contain a large number of recharging events, because a fully loaded battery might allow for driving a much shorter/quicker tour than a heavily discharged one. In case of minimizing travel time, we can find a more reasonable path by considering also the time needed for the recharging event. These 'penalty times' have to be added to all edges in the auxiliary graph which end in a loading station. Then again a single run of Dijkstra's algorithm leads to the optimal sequence of loading stations, see Figure 9.6 for a small example.

9.2.3. Controlling the Number of Recharging Events

In the last section we showed how one can incorporate penalty times for recharging events in order to keep their number small on quickest feasible paths. But it is difficult to estimate the penalty times reasonably, as they actually include possibly waiting times, time to install the new battery (if it is completely switched, otherwise time to recharge), time to pay and so on.

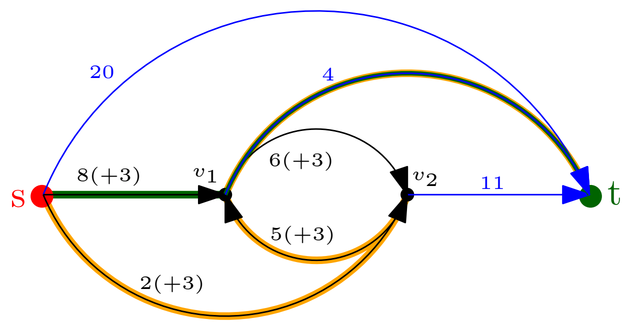


Figure 9.6.: Example for computing the quickest feasible path in the presence of loading stations. If no penalty times for visiting a loading station are assigned, the optimal path (orange, $c = 11$) includes two recharging events. If penalty times are added (values in brackets, here 3 units for each loading station), the new optimal path (green, $c = 12 + 3$) requires recharging only once.

Moreover if distance is minimized instead of travel time, the optimal path still might include a lot of recharging events. Therefore we will now present two ways to control the number of recharging events without accepting long detours.

Limiting the Number of Recharging Events

While the natural limit of recharging events is $|L|$, a driver is normally only willing to accept a few recharging events, e.g. one for small tours and three for longer tours. Therefore we would like to compute the shortest feasible path that requires at most k recharging events. The first step is of course to check if there is *any* path fulfilling the constraint. Therefore we have to compute the minimal number of necessary recharging events on a path from s to t. An efficient way to do so was presented in Section 8.2.1. So we first compute $R(s)$ and $R^{-1}(t)$ and perform a single BFS run in the augmented auxiliary graph, that represents ev-reachability between loading stations. If the number of recharging events revealed by the BFS run is smaller or equal to k, the constrained query is valid. It remains to find the shortest path among all paths from s to t which requires at most k recharging events. For that purpose we turn the auxiliary graph proposed in the last section into a *layered graph* as depicted in Figure 9.7. The main and invariant part of the graph consists of $|L|$ layers each containing a vertex v_l for every loading station $l \in L$. We call $i(v_l) \in \{1, \cdots, |L|\}$ the layer index of v_l. Between subsequent layers j and $j+1$, there exist all edges $(v_l, v_{l'})$ $l \neq l'$ with $i(v_l) = j$ and $i(v_{l'}) = j+1$. Edges are weighted according to the shortest feasible path distance that can be achieved from l to l' in G without using any other loading station. To answer an s-t-query we have to augment the layered graph as well. We insert v_s into the layered graph at layer 0 and connect it to the first layer by adding edges $(v_s, v_l) \forall l \in R(s) \cap L, i(v_l) = 1$ with the precomputed distances. Then we insert v_t at layer $|L|+1$. To enable paths with an arbitrary number of loading stations, we have to connect *every* layer to t. Hence we add edges (v_l, v_t) to the layered graph with $l \in R^{-1}(t)$ and $i(v_l) = 1, \cdots, |L|$. For a given bound k on the number of recharging events, we only connect the layers $1, \cdots, k$ to t or ignore all layers with an index of $k+1$ or higher.

Minimizing the Number of Recharging Events under a Distance Constraint

A user could also be interested to specify a limit on the path length and ask for the path with a minimal number of recharging events and the length not exceeding the given limit. For this purpose we could use the same layered graph as for finding the shortest feasible path regarding loading stations, but now let the distances be the resource consumption and the actual costs are 1 for all edges not adjacent to t and 0 otherwise, see Figure 9.8 (left) for a small example. A

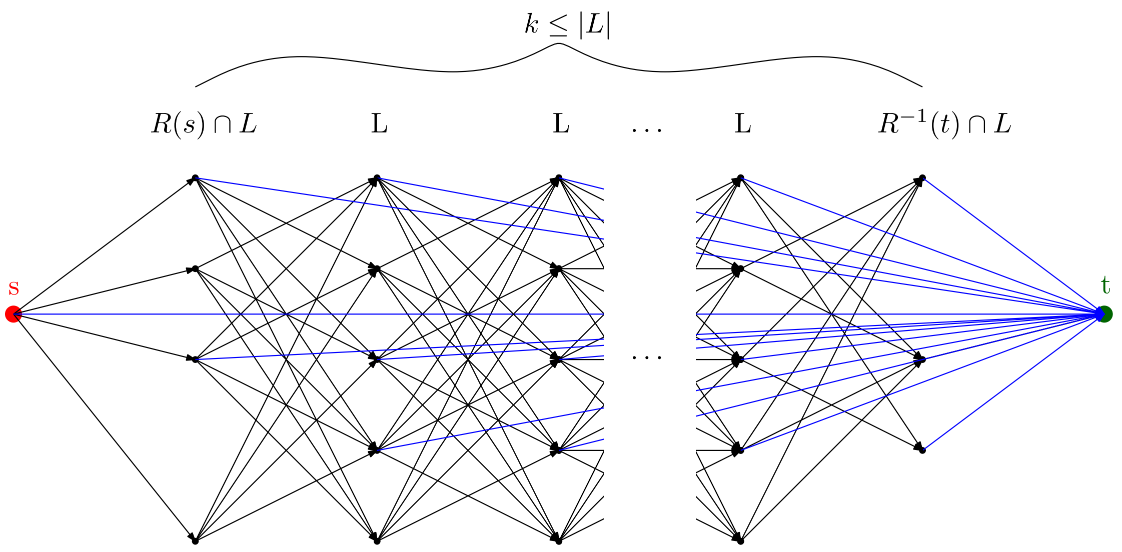
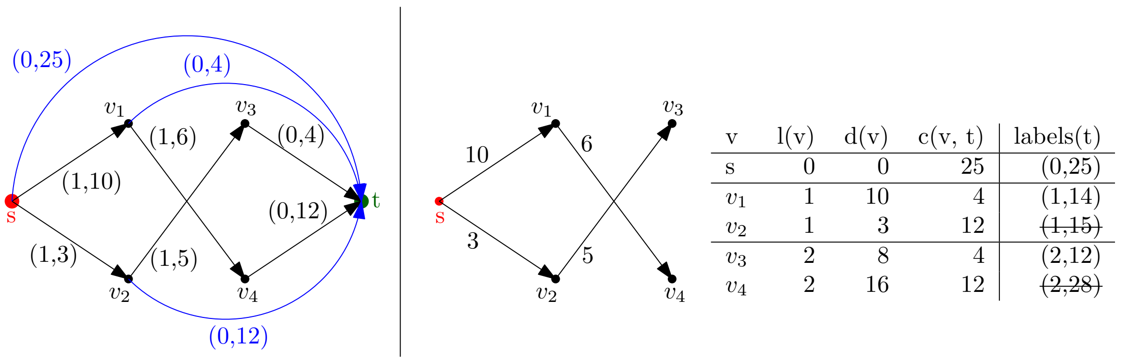


Figure 9.7.: Layered graph to find the shortest feasible path from s to t visiting with at most k loading stations. Each node has an edge to every node in the next layer (black) – apart from edges that indicate the visit of the same loading station consecutively. Nodes in $R^{-1}(t) \cap L$ have an additional edge to t (blue). Edges are weighted according to the shortest feasible path distance in G neglecting loading stations (∞ if no feasible path exists).



v	l(v)	d(v)	c(v, t)	labels(t)
s	0	0	25	(0,25)
v_1	1	10	4	(1,14)
v_2	1	3	12	~~(1,15)~~
v_3	2	8	4	(2,12)
v_4	2	16	12	~~(2,28)~~

Figure 9.8.: Example for finding the path with minimal number of recharging events under a distance constraint. On the left the augmented layered graph is given on which a label setting computation will output the optimal path. On the right an alternative approach is presented. First the shortest path distances of all nodes $v \in \{s, v_1, v_2, v_3, v_4\}$ are computed in the diminished layered graph. Then all possible pareto-optimal labels for t are derived by parsing through the nodes v in layer order and combine their layer index with the minimal distance from s to t over v. Dominated tuples get pruned (crossed out in the table).

LSC from s to t in the layered graph with the given distance limit answers the query. But as the layered graph is a DAG and the resource consumption for all nodes except t is equivalent to the node's layer index, the LSC will assign only one pareto-optimal tuple to every node except t, namely the index combined with the shortest path distance from s to the node in the layered graph. Accordingly there will be at most $|L| + 1$ tuples assigned to t as there are only $|L| + 1$ different cost values possible. The labels for $v \in V \setminus \{t\}$ could also be obtained by computing the shortest path distances for all nodes in the original layered graph without considering edges adjacent to t. As this graph is also a DAG the distance labels are computed layerwise. For the first layer this requires time $\mathcal{O}(|L|)$ as there are only $|L|$ ingoing edges, for every sequent layer we need time $\mathcal{O}(|L|^2)$. As soon as the distance label of a node v is settled, we relax the edge (v, t) (if it exists). If we assign a distance label to t that does not exceed the limit, we are done and the minimal number of recharging events k_{min} equals the layer index $i(v)$. Accordingly the runtime of this approach is in $\mathcal{O}(k_{min}|L|^2)$. Of course, we could also retrieve the set of all pareto-optimal solutions for t by storing the tuples $(i(v), d(v) + c(v, t))$ in a list and pruning dominated ones (see Figure 9.8, right). Because the nodes get parsed in layer order we only have to compare a new tuple to the tail of the list to decide whether one of them needs to be pruned. Hence we need $|L|$ comparisons and the overall runtime is $\mathcal{O}(|L|^3)$.

Observation 9.2. *If for all nodes v in layer $j \geq 1$ the tuple $(j, d(v) + c(v, t))$ is not pareto-optimal for t, then for all nodes w in a layer with an index $k \geq j$ the tuple $(k, d(w) + c(w, t))$ is not pareto-optimal for t either.*

According to Observation 9.2 we can abort the search for pareto-optimal labels for t as soon as we derive no pareto-optimal solution from a certain layer. While this does not change the theoretical runtime, it might save a lot of time in practice.

9.2.4. Experimental Results

We implemented the described approaches and evaluated their performance on three test graphs (WIN, BW, SG), considering varying cruising ranges (CR) and number of loading stations if applicable. All timings (t) were taken on a single core of the Intel i3-2310M processor with 2.1 GHz and 8 GB RAM.

First we computed quickest feasible paths. Without any preprocessing the LSC takes several

	CR	LSC		CH		CH-LSC	
	(km)	polls	t(ms)	t(s)	edges	polls	t(ms)
WIN	25	$1.7{\cdot}10^4$	13	357	$2.1{\cdot}10^6$	148	3
	250	$9.1{\cdot}10^5$	433	42	$1.9{\cdot}10^6$	1316	3
BW	25	$3.7{\cdot}10^4$	15	373	$4.1{\cdot}10^6$	290	4
	250	$1.2{\cdot}10^6$	681	111	$3.9{\cdot}10^6$	1576	3
SG	25	$1.2{\cdot}10^6$	932	8278	$3.1{\cdot}10^7$	435	17
	250	$5.2{\cdot}10^6$	3900	1935	$2.7{\cdot}10^7$	8157	11

Table 9.5.: Experimental results for CH-construction and computation of quickest feasible paths (averaged over 1000 random queries).

seconds and about 5M priority queue polls in Southern German (see the third and fourth column of Table 9.5).The CH only adds about 1-1.6 times the number of original edges as shortcuts to the graph (see the total number of original + shortcut edges in column 6) which is comparable to the normal case for shortest paths. Note that a small cruising range leads to more shortcut insertions, because a larger diversity of paths is optimal for different battery charge levels. Therefore – and because the runtime for pure LSC correlates with the CR – the speed-up for the 25km range is only about 4-55, while for 250km we get up to a factor of 354. Incorporating LSs increases the runtime, because more of the cruising range has to be explored to find all (inverse) reachable LS from s (from t) and the Dijkstra run on the auxiliary (reach-)graph takes additional time, see Table 9.6. Nevertheless we still end up with practical runtimes of about a second on maximum.

Moreover we observed that the average number of reloading events seems to be acceptable for the considered scenarios, while the maximum number indicates that our approach also outputs some unreasonable routes (see the last column of Table 9.6). Adding reloading penalties reduces both values of course, but even when choosing them deliberately there are still optimal routes no user would be willing to take. Therefore we performed experiments with a limit of $k = 2$

	#	CR	reach-graph		query	reachable	avg/max
	LS	(km)	t(s)	edges	t(s)		# reload
WIN	10	50	2	38	0.027	0.83	0.54/2
	10	125	6	72	0.069	0.98	0.00/0
BW	10	50	3	26	0.038	0.54	0.41/3
	10	125	8	65	0.092	0.95	0.17/1
	100	50	22	2360	0.035	1.00	1.46/4
SG	10	125	58	32	0.747	0.58	0.55/2
	10	150	88	34	0.963	0.80	0.30/2
	100	50	69	679	0.128	0.58	3.20/9
	100	125	618	3114	0.834	1.00	1.12/5
	100	150	985	4528	1.150	1.00	0.86/4
	1000	100	3101	249785	0.539	0.96	1.54/5

Table 9.6.: Experimental results for computing quickest feasible paths regarding loading stations (averaged over 10 randomly chosen sets of LS and 100 subsequent random queries each). 'reachable' denotes the ratio of targets to which a feasible path from the source (via LSs) existed.

and $k = 3$ on the number of recharging events in Southern Germany. Unsurprisingly we got comparable query times, because the layered graph is only about k times as large as the original reach-graph. Comparing the pure shortest feasible routes to the ones with the recharging limit, we saw that the ratio of reachable targets remains almost unchanged (reduction by only 4% for $k = 2$) and the average travel time (without reloading time) increases slightly by about $3 - 10\%$. So taking reloading effort into account leads to more practical routes in some cases.
Similar results were obtained for minimizing the number of recharging events under a distance constraint: Setting the bound to 1.05 times the shortest feasible path distance we could already save 1-2 recharging events on average, resulting in much more useful EV routes.

10. Discussion and Extensions

We developed algorithmic solutions for computing routes for EVs considering a variety of reasonable optimality criteria. We could reduce both the theoretical and the practical runtime for computing the most energy-efficient route – obeying the battery constraints and making use of energy recuperation. Using this as a basic building block we could solve more complex and also customizable route planning problems within reasonable time. Answering such kinds of route planning queries for EVs precisely and efficiently is an important step towards a better acceptance of E-Mobility.

Nevertheless this work gives rise to a wide range of new theoretical and practical problems. For example we always assume that recharging at a loading stations is performed until the battery is fully loaded. In practice this condition might be softened, as partial reloading is possible – even if it might be suboptimal for the life span of the battery. Moreover the calculation of the energy-consumption along an edge is a simplification, because there are a lot of external factors which might have a significant impact but are hard to estimate a priori (like weather conditions, additional weight in the trunk, traffic situation, etc.). As we rely in all our approaches on preprocessing, incorporating such parameters would require the existence of fast updating procedures for the precomputed auxiliary data.

Furthermore the locations where an EV can be recharged are still rather sparse in most countries. As loading stations are essential to allow for driving longer tours with the EV, their number has to increase in the next decades to make EVs competitive to conventional cars. Ideally the set of loading stations in a network should allow at least for travelling from an arbitrary source to an arbitrary target and back. Therefore an interesting combinatorial is to find the minimal set of loading stations that meets this or a related criterion.

Part IV.

Epilogue

Conclusions

In this thesis we have considered two major challenges in the context of vehicle navigation: How to acquire one's own position in a fully autonomous manner and how to compute energy-efficient paths.

Our new map matching and self-localization approach is based on the acquisition and retrieval of relative movement patters which we call *path shapes*. While directional information so far has been used only as a backup for short periods of time when e.g. GPS is unavailable, we have shown that it is fully sufficient for self-localization and map matching on its own. We assumed for our approaches that users travel on at least piecewise shortest paths and proposed a variant of Dijkstra's algorithm that takes the shapes of the paths into account. A natural extension would be to consider *all* (non self-intersecting) paths. The shape and comparison models would still be valid in this scenario, but the map matching has to be performed in another way, e.g. using an adaption of breadth-first search instead of Dijkstra's algorithm. Moreover if the navigation system is not built-in by the car manufacturer, the ESP and odometer data might not be available for path shape retrieval. As smartphones are nowadays equipped with navigation software, they are often used as external navigation systems. While not providing precise distance measurements, smartphones can deliver *absolute* directional information via built-in electronic compasses. One challenge is to check whether path shapes based on this data are also sufficient for self-localization.

Computing energy-efficient paths, we took care of the partly negative edge costs (due to energy recuperation) by a generalization of Johnson's shifting technique from constant costs to edge cost functions. Moreover we showed that with the help of contraction hierarchies and the construction of auxiliary graphs, queries can be answered several orders of magnitudes faster – even if taking into account reloading decisions and more complex optimality criteria like finding the shortest path on which the Electric Vehicle does not run out of energy. Our route planning algorithms as well as the approaches for computing the cruising range help to improve the quality and safety of Electric Vehicle navigation. Our proposed preprocessing techniques might be adaptable to a wide range of application domains, where negative cost functions or resource constraints play a role. Future work includes the consideration of partial reloading and dynamic changes of the energy consumption functions (e.g. due strong wind). Furthermore approaches for navigating hybrid cars, which have the ability to switch between battery or fuel powered operation, are of great interest. Our algorithms might serve as basic building blocks to find an energy-efficient path and the best mode of power supply for every section of the path.

Zusammenfassung

Der Großteil moderner Autos ist mit einem eingebauten Navigationssystem ausgestattet. Diese erlauben dem Nutzer durch genaue Fahranweisungen ein beliebiges vorgegebenes Ziel möglichst rasch oder kostengünstig zu erreichen. Darüber hinaus kann auch die aktuelle Verkehrslage berücksichtigt und somit gezielte Stauumfahrung gewährleistet werden. Um diese Aufgaben erfüllen zu können muss zum einen stets die aktuelle Position des Fahrzeugs im Straßennetzwerk bekannt sein und darüber hinaus die optimale Route von dieser Position zum Ziel berechnet werden können. Beide Teilprobleme müssen fortlaufend neu gelöst werden, da das Navigationssystem eventuelle Abweichungen von der vorgegebenen Route sofort erkennen und die entsprechenden Pfadvorgaben zeitnah abändern muss. Daher spielt die Effizienz der Positionsbestimmung und Pfadberechnung eine wichtige Rolle für sicheres und zielgerichtetes Navigieren. In dieser Arbeit wurden beide Probleme untersucht und Algorithmen sowie Beschleunigungstechniken vorgestellt, die die Beantwortung von Anfragen in Straßennetzwerken mit Millionen von Knoten und Kanten in Echtzeit ermöglichen.

Bisher wurde die Position eines Fahrzeuges meist mit Hilfe von GPS Signalen bestimmt. GPS basiert auf einem System von Satelliten, welche frequent ihre Position und die aktuelle Zeit übermitteln. Ein GPS-Empfänger, der Signale von mindestens vier Satelliten gleichzeitig erhält, kann auf dieser Basis den Längen- und Breitengrad nahezu exakt berechnen. Leider kann es beispielsweise durch Hochhäuser, dichte Bewaldung, Wolken oder Tunnel zu Signalblockaden oder Reflektionen kommen, die die Präzision stark einschränken. Daher kann GPS allein nicht überall eine akkurate Positionsbestimmung garantieren. In dieser Arbeit wurde ein neues Konzept – genannt *Path Shapes* – zur Lokalisierung von Fahrzeugen in Straßennetzwerken vorgestellt. Hierbei werden im Gegensatz zu konventionellen Methoden keine absoluten Positionen berechnet, sondern die Form der bisher gefahrenen Trajektorie als Anhaltspunkt benutzt. Diese stellt eine *relative* Abfolge von Bewegungen dar, beispielsweise *"100m geradeaus, Abbiegen um 45° nach links, 2km geradeaus, $\cdots$ "*. Um solch einer Form den korrekten Pfad im Straßennetzwerk zuzuordnen wurden mehrere Methoden vorgestellt: Zur Überprüfung ob von einem gegebenen Startpunkt die gegebene Sequenz von relativen Bewegungen möglich ist ohne das Straßennetz zu verlassen, wurde eine Variante von Dijkstra's Algorithmus beschrieben, welche nur Pfade exploriert deren Präfix eine Form ähnlich der der Referenz aufweisen. Ist der Startpunkt jedoch unbekannt, so muss dieser Ansatz für jeden Knoten im Netzwerk durchgeführt werden, was für praktische Zwecke deutlich zu zeitaufwändig ist. Daher wurde eine Vorberechnungstechnik vorgestellt, die aus der Textsuche entlehnt ist. Kodiert man Pfade als Wörter, so kann man mit Hilfe eines sogenannten Suffixbaumes die Beantwortung von Anfragen in Zeit linear in der Länge der Trajektorie (in Metern) garantieren – zumindest für exakte Eingaben. Da es sich beim Input jedoch um physikalische Messungen handelt, müssen bestimmte Formabweichungen toleriert werden um eine Zuordnung zu einem Pfad in der Karte zu ermöglichen. Zu diesem Zweck wurden robuste Vergleichsmethoden eingeführt und die Vorberechnung des Suffixbaumes entsprechend angepasst. Die notwendige Länge einer Trajektorie um eindeutig im Netzwerk identifizierbar zu sein, stellte sich dabei in Experimenten als überraschend kurz heraus, sodass z.B. im Straßengraph von Deutschland bereits ein Pfad kürzer als ein Kilometer (auch für sehr große Toleranzen) im Schnitt ausreicht, um die entsprechende Form in der Karte zu finden. Für realistische Toleranzen konnte dadurch eine sehr effiziente Lokalisierung

gewährleistet werden, deren Präzision mit der von GPS vergleichbar ist. Ein klarer Vorteil von Path Shapes ist die Autonomie der Datenextraktion; das Odometer misst die gefahrene Distanz, während das ESP die relativen Lenkwinkel verzeichnet. Dadurch ist keine Kommunikation mit externen Geräten mehr nötig um die relative Pfadform zu bestimmen. Darüber hinaus kann so eine gleichbleibend hohe Datenqualität gewährleistet werden, da Umweltfaktoren und Bebauung keinen Einfluss auf die Messgeräte haben. Somit kann die aktuelle Position eines Fahrzeugs im Straßennetzwerk jederzeit sehr akkurat bestimmt werden.

Im Bereich der Berechnung optimaler Pfade von der aktuellen Position zum Ziel konzentrierte sich diese Arbeit auf die besonderen Ansprüche batteriebetriebener Elektroautos, deren Navigation sich maßgeblich von der herkömmlicher Fahrzeuge unterscheidet: Bei der Routenplanung für normale Autos ist das Ziel zumeist Distanz, Fahrzeit oder Benzinkosten zu minimieren. Da in den meisten Ländern ein ausreichend dichtes Tankstellennetz existiert um sicherzustellen, dass rechtzeitiges Auftanken nahezu immer möglich ist, wird der Benzinverbrauch und die aktulle Tankfüllung des Wagens normalerweise außer Acht gelassen. Im Gegensatz dazu sind Ladestationen für Elektroautos noch immer rar, darüber hinaus dauert das komplette Aufladen der Batterie mehrere Stunden. Das Ziel der Routenplanung für Elektrofahrzeuge sollte also sein, den Energieverbrauch zu minimieren um die Notwendigkeit des Aufladens zu umgehen. Außerdem ist dies auch aus ökologischer und ökonomischer Sicht eine sinnvolle Zielsetzung. Leider lassen sich die für konventionelle kürzeste Wege Berechnung entwickelten Algorithmen nicht direkt auf dieses neue Problem übertragen: Zum einen können moderne Elektroautos mittels regenerativen Bremsens einen gewissen Anteil an verbrauchter Energie zurückgewinnen. Dies führt zu teilweise negativen Kantenkosten, was die Anwendbarkeit von Dijkstra's Algorithmus unterbindet. Zum anderen darf die Batterie weder einen negativen Ladestand aufweisen, noch darf die Maximalkapazität der Batterie überschritten werden. Der bisher verwendete Ansatz löste dieses Problem mit einer abgewandelten Version des Bellman-Ford Algorithmus' mit einer theoretischen Laufzeit von $\mathcal{O}(nm)$ mit n Anzahl der Knoten und m Anzahl der Kanten im Netzwerk. Mit dieser Vorgehensweise konnten allerdings keine praktikablen Laufzeiten für größere Straßennetzwerke erzielt werden.
In dieser Arbeit wurde zunächst beschrieben, wie man die Batteriebedingungen als Kantenkostenfunktionen modelliert. Die enstehenden Energieverbrauchsfunktionen in Abhängigkeit vom Ladestand der Batterie weisen die sogenante FIFO-Eigenschaft auf, welche die direkte Anwendung von Bellman-Ford's Algorithmus ermöglicht. Darüber hinaus konnten wir diese Funktionen so umwandeln, dass sämtliche Kosten nicht negativ sind, aber die Struktur optimaler Pfade erhalten bleibt. Die Basis hierfür ist die Verallgemeinerung der Transformationstechnik von Johnson für konstante Kantenkosten auf Funktionen. Im resultierenden Graphen ohne negative Kosten kann dann Dijkstra's Algorithmus zur Ermittlung energieoptimaler Pfade verwendet werden. Da in sehr großen Graphen Dijkstra's Algorithmus noch immer mehrere Sekunden benötigt, wurde außerdem noch eine Variante der Beschleunigungstechnik Contraction Hierarchies beschrieben. Die Anwendung von Contraction Hierarchies bedarf des Einfügens neuer Kanten, deren Kosten sich aus der Verkettung von Kantenkosten entlang eines Pfades ergeben. Im Allgemeinen führt dies zu einer starken Zunahme der Funktionskomplexität und somit auch zu erhöhtem Speicherbedarf und längerer Evaluationszeit. Für unsere Energiefunktionen konnten wir jedoch beweisen, dass die Beschreibungskomplexität einer beliebigen Verkettung durch eine Konstante beschränkt ist, wodurch obige Problem ausbleiben. Die Kombination all dieser Zutaten resultiert in Berechnungszeiten im Bereich von Millisekunden.
Da trotz aller Minimierung des Energieverbrauchs das Aufladen der Batterie zwingend notwendig sein kann um von A nach B zu kommen, wurden außerdem geeignete Algorithmen präsentiert, welche die Route mit der minimalen Anzahl an Neuaufladungen berechnen. Erneut konnte durch

gezielte Vorberechnungen das Finden des optimalen Pfades in unter einer Sekunde gewährleistet werden. Die Basisidee war hier, dass optimale Pfade zwischen Ladestationen invariant unter der Wahl von Start- und Zielknoten sind und diese deshalb vorab berechnet und in einem Hilfsgraphen gespeichert werden können.
Für praktische Routenplanung ist der Energieverbrauch jedoch nicht das einzige Kriterium; gern würde man außerdem den Umweg beschränken, den man bereit ist in Kauf zu nehmen um Energie zu sparen oder den kürzesten oder schnellsten Weg finden, der die Batteriebedingungen erfüllt. Dererlei Probleme sind Instanzen des Constrained Shortest Path Problems, welches im Allgemeinen NP-hart ist. Nichtsdestotrotz konnten wir durch Adaption der bereits vorher verwendeten Vorberechnungstechniken – Contraction Hierarchies und Hilfsgraphen basierend auf den Ladestationen – praktikable Berechnungszeiten erreichen. Die hier vorgestellten Techniken scheinen durchaus geeignet um auch andere Instanzen des Constraint Shortest Path Problems effizienter bearbeiten zu können.

Insgesamt wurden in dieser Arbeit die algorithmischen Grundlagen für ein Navigationssystem beschrieben, welches auf der einen Seite autonome und präzise Selbstlokalisierung im Straßennetz verwirklicht und zum anderen für eine Vielzahl möglicher Optimalitätskriterien in Bezug auf Elektrofahrzeuge die beste Route zum gewünschten Ziel effizient berechnet.

Bibliography

[AAN82] V. Aggarwal, Y. Aneja, and K. Nair. Minimal spanning tree subject to a side constraint. In *32nd ACM Symposium on Theory of Computing (STOC)*, pages 286–295, 1982. 14

[AAN83] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair. Shortest chain subject to side constraints. *Networks*, 13(2):295–302, 1983. 14

[ACH$^+$90] E. M. Arkin, L. Chew, D. Huttenlocher, K. Kedem, and J.S.B. Mitchell. An efficiently computable metric for comparing polygonal shapes. In *1st Symposium on Discrete Algorithms (SODA)*, pages 129–137, 1990. 22

[ACRC09] M. Azizyan, I. Constandache, and R. Roy Choudhury. Surroundsense: mobile phone localization via ambience fingerprinting. In *15th annual international conference on Mobile computing and networking (MobiCom)*, pages 261–272, New York, NY, USA, 2009. ACM. 21

[AERW03] H. Alt, A. Efrat, G. Rote, and C. Wenk. Matching planar maps. *Journal of Algorithms*, 49:262–283, 2003. 22

[AHLS10] A. Artmeier, J. Haselmayr, M. Leucker, and M. Sachenbacher. The shortest path problem revisited: Optimal routing for electric vehicles. In *33rd Annual German Conference on Artificial Intelligence (KI)*, 2010. 45, 47, 58

[AMO93] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993. 68

[BD09] R. Bauer and D. Delling. Sharc: Fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14, 2009. 10, 67

[BDS$^+$10] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *ACM J. of Experimental Algorithmics*, 15, 2010. 10

[BDSV09] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 97–105, 2009. 12, 67

[BDW95] B. Barshan and H.F. Durrant-Whyte. Inertial navigation systems for mobile robots. *Robotics and Automation, IEEE Transactions on*, 11(3), jun 1995. 22

[Bel58] R.E. Bellman. On a routing problem. *Quart. Appl. Math.*, 16:87–90, 1958. 8, 59

[BFM$^+$07] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007. 10

[BGNS10] G. Batz, R. Geisberger, S. Neubauer, and P. Sanders. Time-dependent contraction hierarchies and approximation. In Paola Festa, editor, *Experimental Algorithms*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010. 12, 13

[CLRS90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 1990. 10

[dBCI11] M. de Berg and A. Cook IV. Go with the flow: The direction-based frechet distance of polygonal curves. In *Proc. 1st Int. ICST Conf. on Theory and Practice of Algorithms in Computer Systems (TAPAS)*, 2011. 22

[Del08] D. Delling. Time-dependent sharc-routing. In *16th Annual European Symposium on Algorithms (ESA)*, pages 332–343, 2008. 12, 67

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390. 8

[Dre68] S. E. Dreyfus. An appraisal of some shortest path algorithms. In *ORSA/TIMS Joint National Mtg.*, volume 16, page 166, 1968. 49

[DSSW09] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, pages 117–139, 2009. 10

[DW07] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In Camil Demetrescu, editor, *Experimental Algorithms*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer Berlin / Heidelberg, 2007. 10

[DW09] D. Delling and D. Wagner. Time-dependent route planning. In Ravindra Ahuja, Rolf Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer Berlin / Heidelberg, 2009. 15, 49, 53, 67

[EFH$^+$11] J. Eisner, S. Funke, A. Herbst, A. Spillner, and S. Storandt. Algorithms for matching and predicting trajectories. In *Proceedings of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2011. 13, 21, 34

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large networks. In *Twenty-Fifth Conference on Artificial Intelligence (AAAI)*, 2011. 3, 46, 58, 63

[FB03] M. Frenkel and R. Basri. Curve matching using the fast marching method. *Energy Minimization Methods and Applications in Computer Vision and Pattern Recognition (EMMCVPR)*, pages 35–51, 2003. 22

[For62] L.R. Ford. Flows in networks. *Princeton Univ. Press*, 1962. 8, 59

[FS11] S. Funke and S. Storandt. Path shapes - an alternative method for map matching and fully autonomous self-localization. In *GIS*, 2011. 3, 19

[FT87] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, July 1987. 9, 50

[GKS10] R. Geisberger, M. Kobitzsch, and P. Sanders. Route planning with flexible objective functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 124–137, 2010. 11, 13, 67

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th International Workshop on Experimental Algorithms (WEA)*, pages 319–333, 2008. 10, 11

[Has92] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematical Operational Research*, 17(1):36–42, 1992. 14

[HNR68] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100 –107, july 1968. 10

[IY10] M. Ibrahim and M. Youssef. Cellsense: A probabilistic rssi-based gsm positioning system. *CoRR*, abs/1004.3178, 2010. 21

[Joh73] D. B. Johnson. A note on dijkstra's shortest path algorithm. *J. ACM*, 20:385–388, July 1973. 9

[Joh77] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13, January 1977. 9

[Jok66] H. Joksch. The shortest route problem with constraints. *Journal of Mathematical Analysis and Application*, 14:191–197, 1966. 13

[KMS05] E. Köhler, R. Möhring, and H. Schilling. Acceleration of shortest path and constrained shortest path computation. In *Experimental and Efficient Algorithms*, volume 3503 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2005. 15

[Lau97] U. Lauther. Slow preprocessing of graphs for extremely fast shortest path calculations. *Lecture at the Workshop on Computational Integer Programming at ZIB.*, 1997. 10, 11

[LZZ$^+$09] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate gps trajectories. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, pages 352–361, 2009. 39

[MB09] R. Muhandiramge and N. Boland. Simultaneous solution of lagrangean dual problems interleaved with preprocessing for the weight constrained shortest path problem. *Networks*, 53:358–381, July 2009. 13, 15

[MZ00] K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In Mike Paterson, editor, *8th Annual European Symposium on Algorithms (ESA)*, volume 1879 of *Lecture Notes in Computer Science*, pages 326–337. Springer Berlin / Heidelberg, 2000. 13

[NDSL12] G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional a* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012. 12

[QON07] M. Quddus, W. Ochieng, and R. Noland. Current map-matching algorithms for transport applications: state-of-the art and future research directions. *Transportation Research Part C: Emerging Technologies*, 15:312 – 328, 2007. 21

[SF12] S. Storandt and S. Funke. Cruising with a battery-powered vehicle and not getting stranded. In *AAAI*, 2012. 3, 46

[SS05] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In Gerth Brodal and Stefano Leonardi, editors, *13th Annual European Symposium on Algorithms*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer Berlin / Heidelberg, 2005. 10

[Sto12a] S. Storandt. Quick and energy-efficient routes - computing constrained shortest paths for electric vehicles. In *5th International Workshop on Computational Transportation Science (IWCTS)*, 2012. 4, 46

[Sto12b] S. Storandt. Route planning for bicycles - exact constrained shortest paths made practical via contraction hierarchy. In *ICAPS*, 2012. 4, 46

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995. 10.1007/BF01206331. 31

[VCdL$^+$06] A. Varshavsky, M.Y. Chen, E. de Lara, J. Froehlich, D. Haehnel, J. Hightower, A. LaMarca, F. Potter, T. Sohn, K. Tang, and I. Smith. Are gsm phones the solution for localization? In *7th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 34 –42, 2006. 21

[Yan10] H. Yanagisawa. An offline map matching via integer programming. In *Proc. 20th International Conference on Pattern Recognition (ICPR)*, pages 4206–4209. IEEE, 2010. 21