Karlsruhe Reports in Informatics 2013,5
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

An Alternative Approach to Alternative Routes: HiDAR

Technical Report

Moritz Kobitzsch

2013

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

An Alternative Approach to Alternative Routes: HiDAR Technical Report

Moritz Kobitzsch

June 12, 2013

Abstract

Alternatives to a shortest path are a common feature for modern navigation providers. In contrast to modern speed-up techniques, which are based on the unique distance between two locations within the map, computing alternative routes that might include slightly suboptimal routes seems a way more difficult problem. Especially testing a possible alternative route for its quality can so far only be done utilizing considerable computational overhead. This forces current solutions to settle for any viable alternative instead of finding the best alternative routes possible. In this paper we show a new way on how to deal with this overhead in an effective manner, allowing for the computation of high quality alternative routes while maintaining competitive query times.

1 Introduction

The process of finding a single shortest path in a graph with respect to some weight function has come a long way from Dijkstra's algorithm [Dij59], which allows for solutions in almost linear time. Speed-up techniques [GKW06, GSSD08, DSSW09, DGPW11] offer real time shortest path computation through extensive use of preprocessing. These techniques are used in many web-based navigation services that provide driving directions to the general public.

Even though the computed paths do not rely on heuristics, the optimality of a shortest path remains a subjective measure. Navigation services offer multiple choices, hoping one of them might reflect the preferences of the user. These choices are referred to as alternative routes. The first formal mention of alternative routes, aside from the k-shortest path problem or blocked vertex/arc routes, can be found in [Cam]. Still, until Abraham et al. [ADGW12] introduced the idea of via-node alternatives, which describe an alternative route using source, target, and an additional vertex, only proprietary algorithms were used by navigation providers. This approach was later on improved by Luxen and Schieferdecker through use of extensive preprocessing [LS12], while Bader et al. [BDGS11] proposed an entirely different approach using penalties.

In this paper we focus on alternative routes using the definition of Abraham et al. [ADGW12]. So far, via-node alternatives [ADGW12, LS12] focus on reporting any alternative route, satisfying some minimal quality[1] requirements, as current techniques require three full shortest path queries, which we will describe later on, to check a potential route for its quality.

After the introduction of our terminology in Section 2 and a more extensive discussion of the work related to this paper in Section 3, we introduce a new algorithm to solve the problem of

[1] We will focus on the exact definition of quality later on.

alternative routes in Section 4. Our algorithm allows for the selection of the best possible among the found candidates by reducing the problem to a small graph representing all potentially viable alternative routes. We evaluate our algorithm in Section 9 where we show not only the possibility of calculating higher quality routes but also show the competitiveness regarding query times.

2 Definitions and Terminology

Every road network can be viewed as a directed and weighted graph:

Definition 1 (Graph,Restricted Graph)**.** *A weighted* ***graph*** $G = (V, A, c)$ *is described as a set of vertices* $V, |V| = n$*, a set of arcs* $A \subseteq V \times V, |A| = m$ *and a cost function* $c : A \mapsto \mathbb{N}_{>0}$*. We might choose to restrict* G *to a subset* $\widetilde{V} \subseteq V$*. This* ***restricted graph*** $G_{\widetilde{V}}$ *is defined as* $G_{\widetilde{V}} = (\widetilde{V}, \widetilde{A}, c)$*, with* $\widetilde{A} = \{a = (u, v) \in A \mid u, v \in \widetilde{V}\}$*.*

All methods described within this work are targeted at shortest path computations. We define paths and the associated distances as follows:

Definition 2 (Paths,Length,Distance)**.** *Given a graph* $G = (V, A, c)$*: We call a sequence* $p_{s,t} = \langle s = v_0, \ldots, v_k = t \rangle$ *with* $v_i \in V, (v_i, v_{i+1}) \in A$ *a* ***path*** *from* s *to* t*. Its* ***length*** $\mathcal{L}(p_{s,t})$ *is given as the combined weights of the represented arcs:* $\mathcal{L}(p_{s,t}) = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$*. If the length of a path* $p_{s,t}$ *is minimal over all possible paths between* s *and* t *with respect to* c*, we call the path a* ***shortest path*** *and denote* $p_{s,t} = \mathcal{P}_{s,t}$*. The length of such a shortest path is called the* ***distance*** *between* s *and* t*:* $\mathcal{D}(s, t) = \mathcal{L}(\mathcal{P}_{s,t})$*. Furthermore, we define* $\mathcal{P}_{s,v,t}$ *as the* ***concatenated path*** $\mathcal{P}_{s,v,t} = \mathcal{P}_{s,v} \cdot \mathcal{P}_{v,t}$*.*

In the context of multiple graphs or paths, we denote the desired restriction via subscript. For example $\mathcal{D}_G(s, t)$ *denotes the distance between* s *and* t *in* G*, with* $\mathcal{D}_{p_{s,t}}(a, b)$ *we denote the distance between* a *and* b *when following* $p_{s,t}$*.*

Our metric of choice is the average travel time. Therefore, we might choose to omit the cost function c when naming a graph and simply give $G = (V, A)$.

3 Related Work

The large amount of research available that addresses shortest paths in road networks cannot be appropriately discussed within the limits of this paper. We therefore focus on the work most closely related to ours. Existing overview papers like [DSSW09] can give a better impression of available speed-up techniques and their different properties. In the following, we first discuss speed-up techniques and focus on alternative routes afterwards.

3.1 Speed-up Techniques

While Dijkstra's algorithm [Dij59] still is the fastest algorithm to calculate a single shortest path on a single CPU, the same does not hold true any more in the case of static graph data and multiple queries. The assumption that a large number of queries will be directed at a static network allows to invest a lot of computing power in advance to speed up queries later on. This idea has led to a large number of speed-up techniques. Due to the close relation to our work we cover Reach [GKW06, Gut04], Contraction Hierarchies [GSSD08], and PHAST [DGNW12].

Contraction Hierarchies: Contraction Hierarchies [GSSD08] are a hierarchical speed-up technique. The hierarchy itself is defined through some strict order of the vertices $\prec$: $V \times V \mapsto \{\texttt{true}, \texttt{false}\}$. Vertices are processed, or *contracted*, in this order. During the contraction of a vertex v we inspect all pairs of $u \in V_i = \{u \in V \mid (u,v) \in A^+, v \prec u\}$ and $w \in V_o = \{w \in V \mid (v,w) \in A^+, v \prec w\}$, where A^+ is initially given as A but continually augmented with additional arcs. So to speak, we check whether the distance between two neighbouring vertices of v depends on v; whenever $\mathcal{D}_{G^+_{v\prec}}(u,w) > \mathcal{D}_{G^+_{v\preccurlyeq}}(u,w)$, we add the arc (u,w) with cost $\mathcal{D}_{G^+_{v\preccurlyeq}}(u,w)$ to the current A^+, where $G^+_{v\prec} = (V_{v\prec}, A^+_{v\prec})$, which represents G augmented with additional arcs and restricted to vertices $u \in V$ with $v \prec u$. These new arcs are called *shortcuts* and preserve shortest path distances between vertices of the restricted graph $G^+_{v\prec}$. Each $G^+_{v\prec}$ can be interpreted as an overlay graph to G in which added shortcuts represent whole shortest paths containing vertices $u \prec v$. When all vertices have been processed, the graph G^+ is split up into two new graphs: $G^\uparrow$and $G^\downarrow$, which represent sets of arcs leading away from ($G^\uparrow$) or to ($G^\downarrow$) the vertices.

Formally, $G^\uparrow$ is defined as $G^\uparrow = (V, A^\uparrow)$ with $A^\uparrow = \{a = (v,u) \in A^+ \mid v \prec u\}$. $G^\downarrow$ is defined the same way: $G^\downarrow = (V, A^\downarrow)$ with $A^\downarrow = \{a = (u,v) \in A^+ \mid v \prec u\}$.

A search within a Contraction Hierarchy consists of a forward search from s in $G^\uparrow$ and a backwards search from t in $G^\downarrow$. Geisberger et al. [GSSD08] prove the correctness of this approach by showing that for any $\mathcal{P}_{s,t}$ in the original graph, a concatenated path $\mathcal{P}_{s,v,t}$ can be found with $\mathcal{P}_{s,v,t} = \mathcal{P}_{G^\uparrow,s,v} \cdot \mathcal{P}_{G^\downarrow,v,t}$ with identical length to $\mathcal{P}_{s,t}$ within the original, unprocessed graph. This specific form of path is called an *up-down* path, as it climbs up the hierarchy to v and then descends to t. To retrieve the full shortest path information the concatenated path has to be *unpacked*: All shortcuts are of the form (u,v,w) with (u,v) and (v,w) potentially describing shortcuts as well. By remembering the middle vertex v for every shortcut, the full shortest path in the original graph can be extracted recursively.

PHAST: This technique, though not really a speed-up technique itself, is a method to exploit Contraction Hierarchies and modern hardware architectures to compute full shortest path trees [DGNW12]. It exploits the up-down characteristic of shortest paths within a Contraction Hierarchy as well as the fact that, though originally designed as an n-*level* hierarchy (compare the strict ordering), the actual hierarchical information of a Contraction Hierarchy order represents a rather shallow hierarchy; whenever two vertices cannot reach each other within the hierarchy, they are completely independent from one another and can be viewed as if they were in the same level. The method performs two consecutive steps: In a first step Delling et al. perform a full upwards search from s in $G^\uparrow$. In the second step, vertices are processed (or *swept*) in reverse contraction order, updating distances whenever a better distance value than the current one can be found by using an arc from $G^\downarrow$ in combination with the by now correct distance values of higher level vertices. The nature of a Contraction Hierarchy guarantees the distance value of the vertex with the highest level to be set correctly after this initial search. Inductively, the correctness of lower level vertices follows. This principle can be localized [DGW11, DKLW12] by restricting the set of swept vertices to those reachable in backwards direction in $G^\downarrow$, starting at one of the desired targets.

Reach: Reach is a goal-directed technique introduced by Gutman [Gut04]. Gutman defines the *reach* of a vertex v as $r(v) = \max_{\mathcal{P}_{s,t}, v \in \mathcal{P}_{s,t}} \min(\mathcal{P}_{s,v}, \mathcal{P}_{v,t})$. During a search between s and t, the reach of a vertex v allows for pruning of v whenever $\mathcal{D}(s,v) > r(v)$ and $\mathcal{D}(v,t) > r(v)$. Goldberg et al. [GKW06] improved Gutman's approach through the inclusion of shortcuts, allowing for better

pruning.

3.2 Alternative Routes

Alternative routes in road networks have first been formally studied by Abraham et al. [ADGW12]. By now two general approaches can be found in the literature: via-node alternative routes or the related plateaux-method [ADGW12, LS12, BDGS11, Cam], and penalty-based approaches [BDGS11]. Due to the large differences between the two approaches and the limited scope of this paper, we only cover the via-node approach at this point.

Via-Node Alternative Routes: Within a graph $G = (V, A)$, a via-node alternative to a shortest path $\mathcal{P}_{s,t}$ can be described by a single vertex $v \in V \setminus \mathcal{P}_{s,t}$. The alternative route is described as $\mathcal{P}_{s,v,t}$. As this simple description can result in arbitrarily bad paths, for example paths containing loops, Abraham et al. [ADGW12] define a set of criteria to be fulfilled for an alternative route to be *viable*. A viable alternative route provides the user with a real alternative, not just minimal variations (*limited sharing*), is not too much longer (*bounded stretch*) and does not contain obvious flaws, i.e. sufficiently small sub-paths have to be optimal (*local optimality*). Formally, we define these criteria as follows:

Definition 3 (Viable Alternative Route)**.** *Given a graph $G = (V, A)$, a source s, a target t, and a via-candidate v as well as three tuning parameters γ, ϵ, α; v is a viable via-candidae, and thus defines a viable via-node alternative route $\mathcal{P}_{s,v,t} = \mathcal{P}_{s,v} \cdot \mathcal{P}_{v,t}$, if following three criteria are fulfilled:*

1. $\mathcal{L}(\mathcal{P}_{s,t} \cap \mathcal{P}_{s,v,t}) \leq \gamma \cdot \mathcal{D}(s,t)$ *(limited sharing)*
2. $\forall a, b \in \mathcal{P}_{s,v,t}, \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,a) < \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,b) :$
   $\mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) \leq (1+\epsilon) \cdot \mathcal{D}(a,b)$ *(bounded stretch)*
3. $\forall a, b \in \mathcal{P}_{s,v,t}, \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,a) < \mathcal{D}_{\mathcal{P}_{s,v,t}}(s,b),$
   $\mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) \leq \alpha \cdot \mathcal{D}(s,t) : \mathcal{D}_{\mathcal{P}_{s,v,t}}(a,b) = \mathcal{D}(a,b)$ *(local optimality)*

The usual choice of γ, ϵ, and α is to allow at most $\gamma = 80$ % overlap between $\mathcal{P}_{s,v,t}$ and $\mathcal{P}_{s,t}$. Furthermore the user should never travel more than $\epsilon = 25$ % longer than necessary between any two points on its track, and every subpath that is at most $\alpha = 25$ % as long as the original shortest path should be an optimal path.

These criteria require a quadratic number of shortest path queries to be fully evaluated. Therefore, Abraham et al. propose a possibility to approximate these criteria [ADGW12]. The *approximated viability test* requires three shortest path queries. The first two queries are required to calculate the actual candidate itself, which is necessary to test the amount of sharing; this is unless we use Dijkstra's algorithm, which already computes the shortest path information. For the stretch, we have to check the distance between the two vertices d_1, d_2 where both $\mathcal{P}_{s,t}$ and the alternative candidate $\mathcal{P}_{s,v,t}$ deviate/meet up. As long as $\mathcal{D}_{\mathcal{P}_{s,v,t}}(d_1, d_2)$ does not violate the stretch criteria in respect to $\mathcal{D}(d_1, d_2)$, the path passes the test. Subpath optimality gives the distance necessary for the comparison. The third query is used to approximate the local optimality and is performed between the two vertices o_1, o_2 which are closest to v on $\mathcal{P}_{s,v,t}$ and still fulfil $\mathcal{D}_{\mathcal{P}_{s,v,t}}(o_1, v) \geq T$ and $\mathcal{D}_{\mathcal{P}_{s,v,t}}(v, o_2) \geq T$ respectively, where $T = \alpha \cdot \mathcal{D}(d_1, d_2)$. This check is called the *T-test*. Note that for a correct approximation of the above mentioned criteria, $T = \alpha \cdot \mathcal{D}(s,t)$ would be required. Due to the nature of the test, that would make it impossible to allow for alternatives with up to 80%

sharing without setting α to less than 10%. Therefore Abraham et al. [ADGW12] propose to check against α, and for consistency reasons ϵ, in relation to $\mathcal{D}(d_1, d_2)$.

Definition 3 can be directly extended to allow for second or third alternatives (alternative routes of *degree* 2, 3 or even n). Only the limited sharing parameter has to be tested against the full set of alternatives already known. Bounded stretch and local optimality are only checked against the original shortest path and therefore translate directly.

Abraham et al. [ADGW12] give multiple algorithms to compute alternative routes. The reference algorithm (X-BDV) is based on a bidirectional implementation of Dijkstra's algorithm and is used as the gold standard. To avoid the long query time of Dijkstra's algorithm on continental-sized networks, they also give techniques based on Reach and Contraction Hierarchies. Due to the strong restrictions of the search spaces caused by the speed-up techniques, they present weakened search criteria which they call relaxation. For example, in the Contraction Hierarchy they allow to look downwards the hierarchy under certain conditions. The relaxation can be applied in multiple levels. The respective algorithms are referred to by X-CHV-3/X-CHV-5 for the Contraction Hierarchy-based methods with relaxation levels of 3/5 and X-REV-1/X-REV-2 for the Reach-based variants and their respective relaxation levels[2].

Luxen and Schieferdecker [LS12] improved the algorithm of Abraham et al. in terms of query times by storing a precomputed small set of via-candidates for pairs of regions within the graph. While their method is faster than our algorithm, we refrain from comparing ours to their implementation as they require significantly more preprocessing and storage overhead. Additionally, the small set of candidates lowers the chance to find high quality routes even more.

Plateaux and Via-Nodes: A *plateau* is defined with respect to two shortest paths $\mathcal{P}_{s,u}$ and $\mathcal{P}_{v,t}$. If $\mathcal{P}_{s,u} \cap \mathcal{P}_{v,t} \neq \emptyset$, the common segment is called a plateau. It is easy to see that if two paths share a plateau of appropriate length, any vertex on said plateau can be chosen as via-candidate and will fulfil the local optimality criterion. It is therefore equally valid to find plateaux of length T (compare *approximated viability*) instead of performing the T-test. Even prior to [ADGW12], Cambridge Vehicle Information Technology Ltd. published a method based on full shortest path trees and plateaux [Cam] called *Choice Routing*. Our algorithm takes a rather similar approach to [Cam] while not relying on full shortest path trees.

4 Hierarchy Decomposition for Alternative Routes

One of the most unsatisfying properties of the known algorithms for via-node alternatives is the selection process. Due to the three necessary shortest path queries, testing all potential candidates proves expensive. The relaxation process, which is necessary to increase the chance of finding alternative routes in the first place, does not only increase the number of valid via candidates, it also increases the number of candidates that do not provide viable alternative routes immensely. Therefore, current algorithms order the candidates heuristically. The candidates are tested in this order until a first candidates passes the viability test; the respective candidate is reported as the result. Potentially better candidates are discarded.

For our new algorithm, we want to take a different approach. The main goal is to make the check for viability fast enough to test all potential candidates. Our algorithm (HiDAR), makes the approach of [Cam] viable:

[2] We refer to these in the experimental section.

1. Compute (*pruned*) shortest path DAGs from s and directed at t
2. Utilize plateaux to create a compact and meaningful alternative graph
3. Extract alternative routes from this graph

The main task is the calculation of the alternative graph. In this alternative graph $H = (V_H, A_H)$ we want to encode the following information: $\mathcal{D}(s, v)$ and $\mathcal{D}(v, t)$ for any $v \in V_H$. Furthermore, we encode for any $a \in A_H$ whether a forms a plateau with respect to the paths in the shortest path DAGs from the first step. No vertex in V_H, except for s, t or the first/last vertex of a plateau should have indegree and outdegree equal to one. It is obvious that alternative graph extraction on such graphs is simple, as all the information for the (approximated) viability check is present. Sufficiently small graphs allow for very fast alternative graph extraction. This process is illustrated graphically in Figure 2.

4.1 High Level Algorithm Description

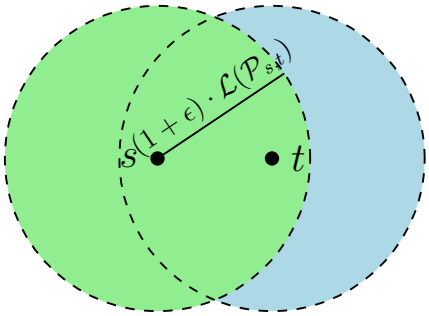
The first step of our algorithm computes shortest path trees within a Contraction Hierarchy, instead of the full graph (compare Figure 1). Due to possible overlaps hidden within shortcuts, these shortest path trees actually describe a shortest path DAG in the original graph.

Shortest Path Search: Our approach resembles the behaviour described in [DGW11, DKLW12]. In a first step, we perform two exhaustive upwards searches from s in $G^{\uparrow}$ and backwards from t in $G^{\downarrow}$. These searches describe two sets of vertices V_s and V_t which contain those vertices which were reached by the two respective searches, together with their (tentative) distance values. We aim at computing the correct distance values from s to $V_s \cup V_t$ as well as from $V_s \cup V_t$ to t. In the following, we describe the process for s, t is handled analogously. Following the principles of [DGW11, DKLW12], we can compute $\mathcal{D}(s, v)$ for any $v \in V_t$ by sweeping (see Section 3) V_t in reverse contraction order. To compute $\mathcal{D}(s, v)$ for all $v \in V_s$, we need to extract what we call the *backwards hull* $H_{G^{\downarrow}}(V_s)$ of V_s: the union of all vertices reachable by backwards arcs in $G^{\downarrow}$ from any vertex in V_s. Note that in an undirected graph $H_{G^{\downarrow}}(V_s) = V_s$. By sweeping $H_{G^{\downarrow}}(V_s)$ in reverse contraction order, we can calculate $\mathcal{D}(s, v)$ for all $v \in V_s$.

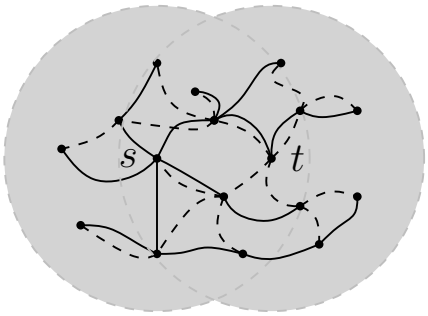
For the further steps, we restrict the processing to vertices $v \in V_s \cup V_t$ with $\mathcal{D}(s, v) + \mathcal{D}(v, t) \leq (1 + \epsilon) \cdot \mathcal{D}(s, t)$ (compare Definition 3, Figure 2(b)).

Hierarchy Decomposition: The data calculated in the first step contains all necessary distance information for our alternative graph. The alternative graph itself has to be calculated by finding all vertices that actually provide relevant information. Those vertices might currently be encoded within the shortcuts of the shortest path trees: any two shortcuts, whether they belong to the tree rooted at s or the one rooted at t, can have common segments, as each of the shortcuts represents a shortest path in the original graph. We need to find the vertices at which two of these paths meet up or deviate from one another and decide whether they form a plateau, i.e. do not belong to the same DAG.

We perform a Hierarchy Decomposition [LS11] to unpack all shortcuts at once. During the process, we handle shortcuts in reverse contraction order of their middle vertices, ordering them with a priority queue. The order enables us to decide locally whether a given middle vertex is important by



(a) Choice Routing [Cam]



(b) HiDAR; dots mark $V_s \cup V_t$, forward tree painted solid, backward tree dashed

Figure 1: Schematic representation of graph exploration for shortest path tree calculation

just storing its predecessor and successor in the fully unpacked shortcut, as all shortcuts containing the same middle vertex are processed subsequently. All vertices with $indegree + outdegree > 2$ are deemed important for the alternative graph.

We can avoid special cases in this process by including some dummy arcs and vertices[3]. By handling the shortcuts in a carefully chosen order, we also manage to calculate some additional data during the unpacking process that allows for direct generation of a compacted version of the alternative graph, consisting only of important vertices as well as source/target vertices of the input shortcuts[4]. This step is illustrated schematically in Figure 2(c).

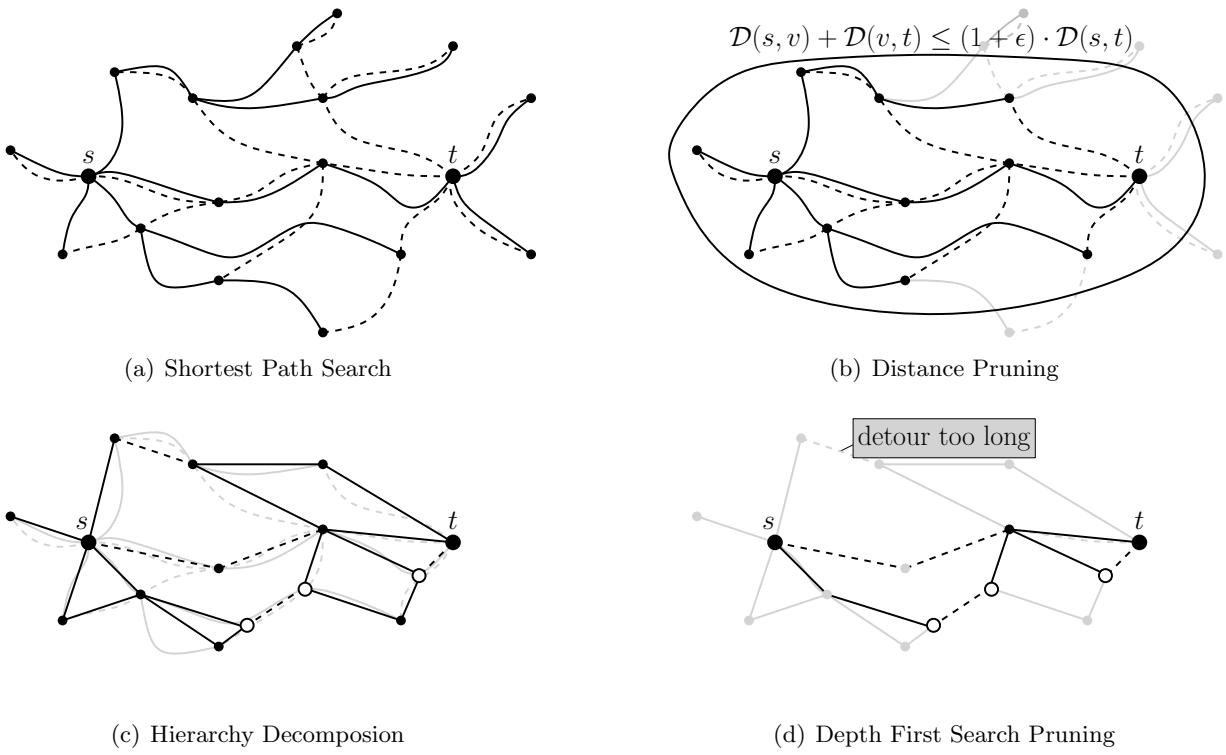
Alternative Graph Extraction: While the graph calculated in the previous step already contains all necessary information desired for our alternative graph, it still contains a lot of unwanted information, too. Actually important for the alternative route extraction are only paths that connect plateaux of sufficient size to the source and the target vertex. To finalize our alternative graph, we perform a depth first search (DFS) from the source. Whenever this DFS encounters a plateau large enough to justify the necessary detour to reach it, which we approximate by trivial lower bounds, we mark the paths from s to the plateau and from the plateau to t to be included in the alternative graph. All other plateaux, compare Figure 2(d), are discarded[5]. Finally, we compact the graph to get the alternative graph we report as the result of our algorithm.

5 Algorithmic Details

In Section 4 we implied that unpacking data multiple times can be avoided by our algorithm and that we can directly calculate usage data for every input arc in the alternative graph. Now that we have thoroughly explained out algorithm on a high level, we can go on to give a full explanation on how the unpacking process is implemented and what problems have to be addressed.

We first give some insight into the process of artificial shortcuts. Afterwards we explain the decomposition phase in more detail than in Section 4.

[3] See Appendix 6 for details [4] See Appendix 7 for details [5] See Appendix 8 for details



(a) Shortest Path Search

(b) Distance Pruning

(c) Hierarchy Decomposion

(d) Depth First Search Pruning

Figure 2: Schematic representation of the alternative graph generation process of HiDAR. $H(V_s, V_t)$ is marked with solid dots. The forward shortest path tree is marked with solid curves, the backward tree with dashed ones. Hollowed circles represent newly found important vertices. Solid straight lines (c,d) mark arcs in the alternative graphs. Dashed straight lines mark plateaux within the alternative graph. Grayed out information represents pruned information.

6 Artificial Shortcuts

We mentioned earlier that artificial (or dummy) shortcuts prevent us from dealing with special cases. Our main problem during the decomposition phase is that the process is solely designed to handle shortcuts. Arcs stemming from the original graph do not fit the pattern but still have to be handled, as other shortcuts might just contain these arcs. Also, these arcs contribute to the decision whether some vertex is important or not. And this is not even limited to non-shortcut arcs. Even full shortcuts can be contained within another shortcut themselves. Instead of handling these original arcs directly, we can integrate them into the whole process transparently, thus creating a more elegant solution. We could try and treat the original shortcuts somehow; however, it is way more elegant to transparently integrate them into the whole process. As mentioned above, this is done through artificial shortcut generation. Essentially, for every arc we want to process, we add two vertices together with their connecting arcs to our graph. For the new artificial vertices, we perform a few contraction steps. Each of these two vertices is connected to either source or target of the input arc with an arc of length zero. We then create two shortcuts with the source and target as middle vertices and feed the necessary unpacking information directly into the data structures handling our unpacking process. The full construct of the artificial shortcuts is illustrated

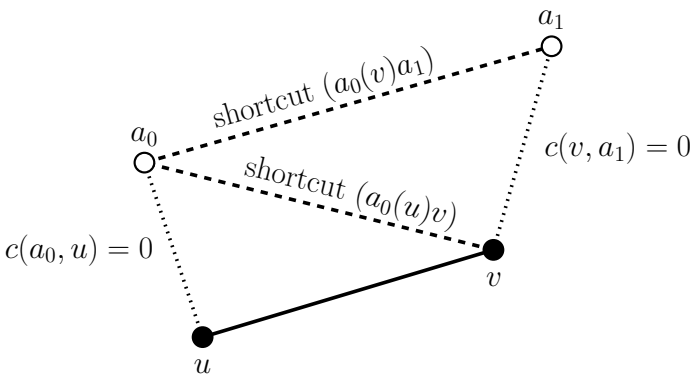


Figure 3: Artificial Shortcut Construct: a_0 and a_1 are invalid vertices. They are connected to u and v, the source vertices of the input arc (u, v) with arcs of length zero. Note that these arcs are never actually created. The unpacking process just needs to know that their ids are invalid and thus they do not have to be processed. Actually created are only the two shortcuts from a_0 to a_1 with v as middle vertex and a_0 to v with u as middle vertex. For this example we assume $u \prec v$

in Figure 3.

This comes at next to no overhead, as only a few hundred shortcuts are created. The arcs of length zero ensure that we can still calculate the distance of a given middle vertex from the source of the shortcut correctly.

7 Detailed Decomposition

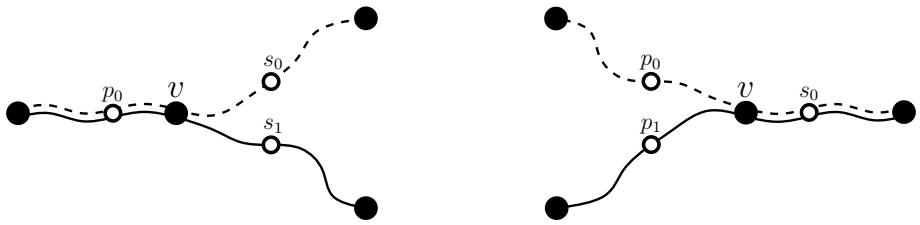
While Section 4 only claims that we process arcs in reverse contraction order of their middle vertices, we actually use an additional sorting criterion. First we assign a unique id to every input arc that we have to process during the decomposition phase. We utilize these ids by storing tuples of shortcuts and these ids as well as the middle distance from the source of the input arc to the currently processed middle vertex. We then extract all arcs with the same middle vertex at once and process them in order of the unique id. In a first step we loop over all extracted pairs and check whether the vertex is important by calculating the number of unique predecessors/successors. Note that the inclusion of the artificial shortcuts makes every source vertex or target vertex of an input arc important.

In the case of an unimportant vertex and if parts of the currently looked at arc are shortcuts themselves, we just update the priority queue to process these shortcuts, too.

Additional overhead, aside from the unpacking process, is only necessary at important vertices.

Fortunately, we can handle the work locally and only a few potential cases exist. Consider an important vertex v. n_1 arcs enter this vertex and n_2 arcs exit it with $n_1 + n_2 > 1$. Two potential cases are given in Figure 4. All other cases can be described by a combination of the two depicted cases with an arbitrary number of possible paths. The number of paths however does not complicate the handling at all, as only the path with lowest id has to be handled specially.

For every unique predecessor/successor and unique associated arc (compare p_0, p_1 and s_0, s_1 in Figure 4), we insert a new tuple into the priority queue. As id, we set the minimal id of all ids that had the same predecessor and arc associated. For every unique path, we store v, and the distance



(a) Paths splitting up at an important vertex v

(b) Paths meeting up at an important vertex v

Figure 4: Illustration of important vertex classification in the local view during decomposition

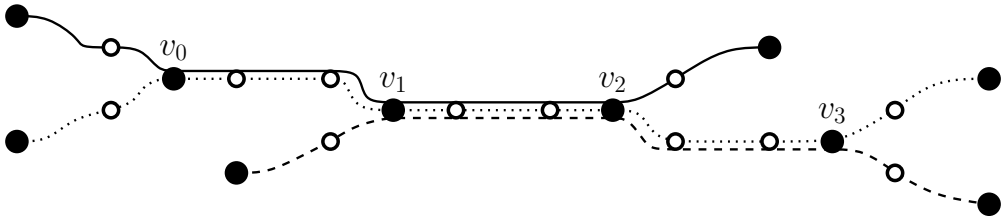


Figure 5: Sequence of joining and splitting paths and their respective important vertex processing

from the source of its path, in the list that can be identified by the unique id of the path. Every other path than the one with the lowest id is also marked to not include the segment starting at v in the output. This ensures that in case of two paths joining at v, for every unique successor only a single arc will be generated leaving v (compare Figure 4(b)). In the case of multiple paths splitting up (cf. Figure 4(a)), this ensures that the now distinct paths are correctly represented in the output graph again.

The full process is illustrated in Figure 5. The figure shows three paths (shortcuts); s_0 (solid), s_1 (dashed), and s_2 (dotted) with respective ids 0, 1, and 2. At v_0, s_2 is marked to be not included in the alternative graph, as it is covered by s_0 at this point and s_0 has a lower id. At v_1, s_1 also joins this set of paths and is marked as covered. Going on to v_2, these paths split from s_0, while s_1 and s_2 continue on in the same direction. At this point, s_1 becomes the path with the lowest id for the next unique successor and is marked as not covered again. Finally, at v_3, s_2 is marked as not covered too, as it does not share vertices with s_1 any more. What is decisive in this process is the fact that the actual order we process the important vertices in is irrelevant. We cannot miss a covering path ending in between, as the artificial shortcuts result in every source and target vertex of a shortcut to be important.

While it is rather obvious that we do not unpack arcs multiple times, as we only generate a single tuple in the priority queue for every arc, the correctness of this approach is less obvious. The following claims make for its correctness:

Claim 1. *Throughout our decomposition process, the following holds:*

1. *No parallel paths are created in the alternative graph where no parallel paths exist in the original graph.*

2. *Not a single path is lost.*

Proof. Assume the existence of a new path in the alternative graph that was not present in the original graph. This implies a vertex v exists at which the paths split up in the alternative graph while only a unique successor of v exists within the original graph. As we only generate new arcs at important vertices, v is important. Whenever we process a vertex v, we only generate a single tuple is stored for every unique successor at v with at least one of the availability flags is set to `true`. Since we process all arcs containing v at the same time, no two such tuples can exist in the lists of the input arcs. Thus, v cannot exist.

For the second claim, we have to check that every arc a that is marked as unavailable, starting at some vertex v, will become available again the moment it is not represented by another arc anymore. Assuming this is not the case, would imply that some vertex v_a exists, at which the arc that triggered the unavailability of a and a itself split up. Therefore, at some point, we would process v_a. At this point, the successor of a and the arc that triggered the unavailability are distinct. This leaves two possibilities. Either, a is associated with the minimal id over all arcs with the same successor at v_a, which would result in a becoming active again, or another arc exists which is covering a at this vertex. In that case, we can recursively turn to the same argument, until we either reach the target vertex of a or the first case holds. □

To be able to track plateaux within this process and to not unpack arcs contained in both forward and backward DAG twice, we keep track of the availability of segments of the paths in forward and backward DAG with flags.

As we process the vertices in reverse contraction order, the list of tuples for any arc can be in arbitrary order. The distance values from their respective source vertices ensure that we can generate a meaningful set of arcs for the output graph. For the final step, we now only have to sort each of the lists with respect to the distance values. The original arcs, in combination with the data we calculated, give us the vertices and arcs of the alternative graph.

8 DFS-filtering

The DFS based filtering process of our algorithm is actually implemented as a heuristic. To check whether an alternative route is viable, we need to know about the chosen shortest path (as there might be multiple shortest paths encoded in the graph). Without this information, it is impossible to decide whether a given plateau is long enough. Given $\mathcal{P}_{s,t}$ and $p_{s,t}$, however, we can use $d = \mathcal{L}(p_{s,t}) - \mathcal{L}(\mathcal{P}_{s,t})$ to approximate potential viability later, as d forms a lower bound on the actual length of $p_{s,t} \setminus \mathcal{P}_{s,t}$. We describe our DFS pruning following the representation in Figure 6. The plateau p_0 gives an example of pruned information. Consider $\mathcal{L}(p_0) < alpha \cdot (\mathcal{L}(\mathcal{P}_{s,v_0}) \cdot \mathcal{L}(\mathcal{P}_{v_0,t}) - \mathcal{L}(\mathcal{P}_{s,t}))$. The lower bound for the distinct part of $\mathcal{P}_{s,v_0} \cdot \mathcal{P}_{v_0,t}$ in relation to $\mathcal{P}_{s,t}$ guarantees that p_0 will not describe a viable alternative route. Therefore, we can omit p_0 and the path to/from it. In contrast to this discarded information are the plateaux p_1 and p_2. Both pass our test and should be included in the output, even though they actually might not produce a viable alternative. When we notice a successful test at v_1, we directly mark the path to the target $\langle v_1, m_{1,2}, t \rangle$ to be a path in the alternative graph.

The same happens later on at v_2, which leads to $m_{1,2}$ to be marked as a vertex of the alternative graph. For marking the path that leads to p_1, we propagate a flag upwards our DFS tree that indicates the inclusion of arcs. Whenever more than one of the arcs leaving a given vertex v have to be included, based on to the propagated flags, the vertex v is marked to be included in the DFS

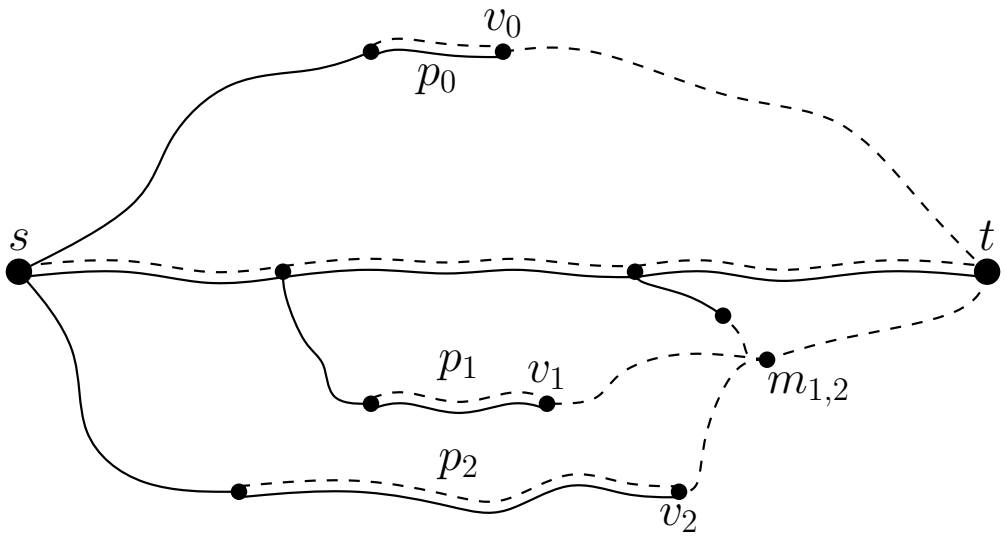


Figure 6: Schematic representation of important information encountered during our DFS pruning: DFS tree (potentially DAG) marked in solid black, backwards tree given in dashed lines. v_i depict vertices at ends of plateaux. This is where actual decisions regarding inclusion are made. p_i denotes some plateaux, $m_{1,2}$ a vertex at which the paths from p_1 and p_2 to the target of the search meet.

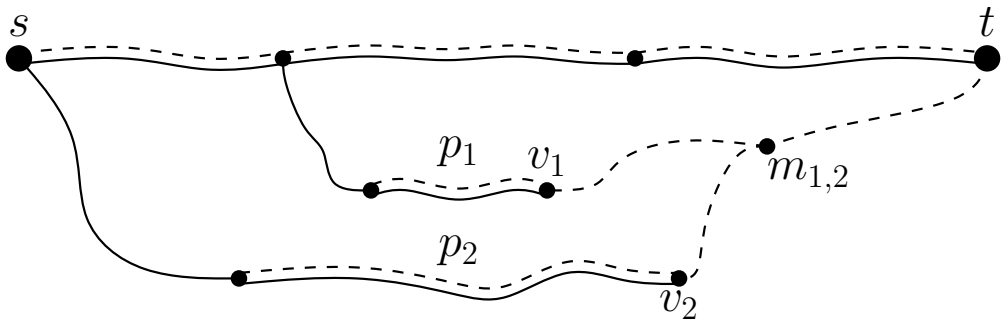


Figure 7: Schematic representation of the output of our DFS-pruning algorithm after processing the data in Figure 6.

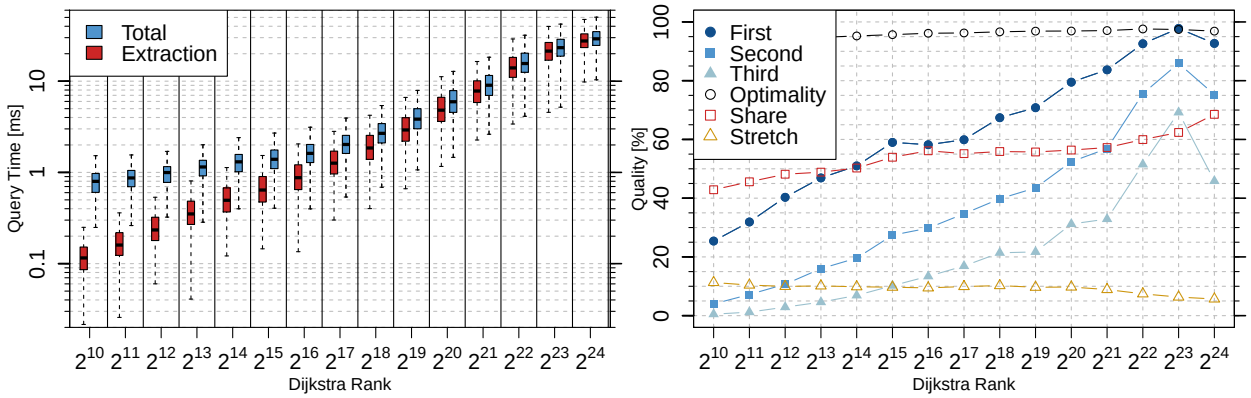


Figure 8: Query time, in total and restricted to the graph generation process (left). Quality measurements in terms of success rates for the first three alternatives and specifically for the first alternative (right).

graph. All arcs not marked are discarded. The final output is illustrated in Figure 7.

9 Experiments

Our experimental configuration is as follows: All our experiments were performed on a single core of an Intel(R) Core(TM) i7 920 CPU clocked at 2.67 GHz. The machine is equipped with 12GB DDR3-1333 RAM, runs SUSE Linux (kernel 2.6.34.10-0.6-default), and approximates the machine used in [ADGW12]. The code was developed in `c++` and compiled using `g++` in version 4.5.0 using `-std=c++0x -O3 -mtune=native` as parameters. We use two kinds of inputs to test our algorithm. The first set T_n is composed of 10 000 source and target pairs, the second set T_r of 1 000 source and target pairs per Dijkstra rank. All queries are chosen uniformly at random. We use classic parallel Contraction Hierarchies preprocessing, allowing to process the European road network within a few minutes. In this section, we focus only on the runtime and the quality of our algorithm. *Success rates* given describe the number of times an alternative route of a given degree was found. For alternative route selection, we follow the approach of [Cam] and select long plateaux first.

Runtime: First, we compare the query time and success rates of our algorithm to the techniques presented in [ADGW12]. For the first alternative, our algorithm is already competitive in its query time. Higher degree alternative route extraction does not really affect our algorithm. In fact, we can extract all alternatives within less than 0.1 ms. In contrast, the other algorithms, which provide similar success rates, require significantly longer query times for higher degree alternatives. The algorithm by Luxen and Schieferdecker would outperform ours in pure query time, but requires additional sets of via candidates for any possible further alternative and a significant amount of additional storage and preprocessing time.

Our algorithm not only allows for finding second and third degree alternatives with a high success rate; we observe a fourth alternative in about 36.4% of all cases and a fifth/sixth alternative in 22.4% and 12.18% of the cases while still only requiring 18.2 *ms* on average.

The runtime of our algorithm is dominated by the graph generation process, as can be seen in

Table 1: Comparison of runtime and success rates for multiple alternatives based on the test set T_n. Numbers for X-BDV, X-CHV-3/X-CHV-5, X-REV-1/X-REV-2 are based upon the data given in [ADGW12]. Query times are accumulated times up to the n-th alternative.

	first		second		third	
algorithm	time [ms]	success rate [%]	time [ms]	success rate [%]	time [ms]	success rate [%]
X-BDV	11 451.5	94.5	12 225.9	80.6	13 330.9	59.6
X-CHV-3	16.9	90.7	20.3	70.1	22.1	42.3
X-CHV-5	55.2	92.9	65.0	77.0	73.2	53.3
X-REV-1	20.4	91.3	33.6	70.3	42.6	43.0
X-REV-2	34.3	94.2	50.3	79.0	64.9	56.9
HiDAR	18.2	91.5	18.2	75.7	18.2	55.9

Figure 8. While unpacking shortcuts in advance does speed up most techniques, it does not so in ours. On the contrary, we measured significantly longer query times for our algorithm, well beyond the 80 *ms* mark. But due to the large amount of overlaps present in our data and due to the faster compression of our new technique, our live extraction manages to outperform pre-unpacked shortcuts.

Quality: As our method to alternative routes differs greatly from the approaches presented in the literature so far, we also evaluate the quality of the alternative routes generated by our algorithm based on stretch, sharing and local optimality. The quality values are given in Table 2. In the worst case, we report stretch values worse than previous algorithms. These values originate from a very small set of outliers that would have been deemed viable by the other algorithms as well. On the other hand, we outperform the classic algorithms by far by providing better stretch values than previously known algorithms. What strikes the eye the most are the values for local optimality. The worst case performance of our algorithm can nearly compete with the average case performance of the other algorithms. On average, we can report local optimality of 90 %. These routes are probably not reported by X-BDV, as the selection method prefers other alternatives first that share some parts with the high quality route, thus making it inadmissible later on. However, the computational overhead of X-BDV does not allow to justify this assumption by fully exploring all potential candidate selection schemes. Only in terms of sharing our first alternative performs worse than other algorithms.

The quality for higher degree alternatives still remains high. The uniformly bounded stretch gradually increases to 21.4 % on average for the 10th alternative, while the amount of sharing averages out at around 60 % and local optimality remains high with an average of around 82.9 %. Figure 8 shows that we can achieve this high quality over the full range of queries.

10 Conclusion and Future Work

We have presented a new method to compute alternative routes in a competitive way, allowing for high success rates and the extraction of a maximum number of alternative routes without additional effort. By this, we present the only algorithm providing reasonable means for extracting up to 10

Table 2: Quality of HiDAR alternatives (based on test set T_n) compared to X-BDV, X-CHV and X-REV. For optimality we report local optimality restricted to the detour (as also done in [ADGW12]). For second and third alternatives, no quality measures are given for the relaxed variants of X-CHV and X-REV.

		success	UBS		sharing [%]		loc opt [%]	
#	algorithm	rate [%]	avg	max	avg	max	avg	min
first	X-BDV	94.5	9.4	35.8	47.2	79.9	73.1	30.3
	X-CHV-3	90.7	11.5	45.4	45.4	80.0	67.7	30.0
	X-REV-2	94.2	9.7	31.6	46.6	79.9	71.3	27.6
	HiDAR	91.5	6.6	64.2	63.0	80.0	97.4	70.5
second	X-BDV	80.6	11.8	38.5	62.4	80.0	71.8	29.6
	HiDAR	75.7	10.6	57.7	63.2	80.0	94.4	64.0
third	X-BDV	59.6	13.2	41.2	68.9	80.0	68.7	30.6
	HiDAR	55.9	12.9	76.2	62.5	80.0	92.4	65.3

alternative routes. The high performance during the actual alternative route extraction allows for a precise evaluation of the alternative candidates and for extraction schemes based on personal preferences instead of accepting any acceptable candidate.

Still, the runtime of our algorithm itself could possibly be improved. The full extraction of shortcuts does seem like an unnecessary amount of effort compared to the low number of important vertices. Finding methods to restrict the extractions to only shortcuts promising some results would greatly speed up our algorithm.

Although every plateau defines a viable via-node alternative route, this does not hold true the other way around. Therefore, it would be interesting to see how many more alternative routes can be found through a combination of both via-node alternatives and our method or even to incorporate other approaches like [DKLW12].

Acknowledgements.

We would like to thank Dennis Schieferdecker for interesting discussions and insight into the world of alternative routes.

References

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Alternative Routes in Road Networks*, ACM Journal of Experimental Algorithmics (2012).

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders, *Alternative Route Graphs in Road Networks*, Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS'11), Lecture Notes in Computer Science, vol. 6595, Springer, 2011, pp. 21–32.

[Cam] Cambridge Vehicle Information Technology Ltd., *Choice Routing*.

[DGNW12] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck, *PHAST: Hardware-accelerated shortest path trees*, Journal of Parallel and Distributed Computing (2012).

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

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Faster Batched Shortest Paths in Road Networks*, Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11), Open Access Series in Informatics (OASIcs), vol. 20, 2011, pp. 52–63.

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

[DKLW12] Daniel Delling, Moritz Kobitzsch, Dennis Luxen, and Renato F. Werneck, *Robust Mobile Route Planning with Limited Connectivity*, Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), SIAM, 2012, pp. 150–159.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner, *Engineering Route Planning Algorithms*, Algorithmics of Large and Complex Networks, Lecture Notes in Computer Science, vol. 5515, Springer, 2009, pp. 117–139.

[GKW06] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck, *Reach for A*: Efficient Point-to-Point Shortest Path Algorithms*, Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06), SIAM, 2006, pp. 129–143.

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

[Gut04] Ronald J. Gutman, *Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), SIAM, 2004, pp. 100–111.

[LS11] Dennis Luxen and Peter Sanders, *Hierarchy Decomposition for Faster User Equilibria on Road Networks*, Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11), Lecture Notes in Computer Science, vol. 6630, Springer, 2011, pp. 242–253.

[LS12] Dennis Luxen and Dennis Schieferdecker, *Candidate Sets for Alternative Routes in Road Networks*, Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12), Lecture Notes in Computer Science, vol. 7276, Springer, 2012, pp. 260–270.