Fast Computation of Shortest Smooth Paths and Uniformly Bounded Stretch with Lazy RPHAST

Tim Zeitz
Karlsruhe Institute of Technology, Germany

Abstract

We study the shortest smooth path problem (SSPP), which is motivated by traffic-aware routing in road networks. The goal is to compute the fastest route according to the current traffic situation while avoiding undesired detours, such as briefly using a parking area to bypass a jammed highway. Detours are prevented by limiting the uniformly bounded stretch (UBS) with respect to a second weight function which disregards the traffic situation. The UBS is a path quality metric which measures the maximum relative length of detours on a path. In this paper, we settle the complexity of the SSPP and show that it is strongly NP-complete. We then present practical algorithms to solve the problem on continental-sized road networks both heuristically and exactly. A crucial building block of these algorithms is the UBS evaluation. We propose a novel algorithm to compute the UBS with only a few shortest path computations on typical paths. All our algorithms utilize Lazy RPHAST, a recently proposed technique to incrementally compute distances from many vertices towards a common target. An extensive evaluation shows that our algorithms outperform competing SSPP algorithms by up to two orders of magnitude and that our new UBS algorithm is the first to consistently compute exact UBS values in a matter of milliseconds.

2012 ACM Subject Classification Theory of computation → Shortest paths; Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases realistic road networks, route planning, shortest paths, traffic-aware routing, live traffic, uniformly bounded stretch

Digital Object Identifier 10.4230/LIPIcs.SEA.2022.3

Supplementary Material *Software (Source Code)*: `https://github.com/kit-algo/traffic_aware` archived at `swh:1:dir:656ed74654a81353c3de75f0a7a79bbc65ec5818`

Acknowledgements I want to thank my colleagues for from the scalable algorithms group Christopher Weyand, Marcus Wilhelm and Thomas Bläsius for pointing out a crucial fact on subpath-optimality at our group workshop. Further, I want to thank my colleague Jonas Sauer for many helpful discussions on algorithmic ideas and proofreading of early drafts of this paper. I also want to thank the anonymous reviewers for their helpful comments. Finally, I would like to thank Jakob Bussas who did a proof-of-concept implementation of the ideas presented here for his bachelor's thesis.

1 Introduction

Over the past years, mobile navigation applications have become ubiquitous. A core feature of these applications is to compute routes between locations in road networks. These routes can be obtained by computing shortest paths on a weighted graph representing the road network with travel times as weights. To present users with *good* routes, it is crucial to take the current traffic situation into account. However, integrating the current traffic situation comes with its own challenges. As traffic feeds are derived from live data, they are inherently noisy and incomplete. Simply exchanging free flow for live traffic travel times and then solving the classical shortest path problem may lead to problematic routes. For example, such routes may include undesired detours such as briefly using a parking area to bypass a jammed highway.

We therefore study an extended problem model, the *shortest smooth path problem* (SSPP) [8]. To avoid undesired detours, a second weight function is taken into account. The first *volatile* weight function models the current traffic situation. The second *smooth* weight function models the free flow travel times and may include additional penalties, for example to avoid residential areas. The goal is to find the shortest path with respect to the volatile weights without too severe detours with respect to the smooth weights.

Related Work. The classical shortest path problem on weighted graphs can be solved with Dijkstra's algorithm [11]. To this day, no asymptotically faster algorithm is known. However, for many practical applications on continental-sized road networks, it is too slow. During the past decade, this has motivated a lot of research effort on engineering faster shortest path algorithms on road networks. Results from this research have played an important role in enabling modern routing applications. By introducing an offline preprocessing phase where auxiliary data is precomputed, queries can be accelerated by more than three orders of magnitude over Dijkstra's algorithm. For an extensive overview on these speed-up techniques, we refer to [2]. One particularly popular technique which we also utilize in this work is *Contraction Hierarchies* (CH) [14]. During preprocessing, additional shortcut arcs are inserted into the graph, which skip over unimportant vertices. On continental-sized road networks with tens of millions of vertices and arcs, CH preprocessing typically takes a few minutes. Shortest path queries can be answered in less than a millisecond. Applying CH to extended problem models is not always trivially possible. We therefore utilize CH indirectly as an A^* [15] heuristic. This approach is called *CH-Potentials* [18]. CH-Potentials is built on *Lazy RPHAST* [18], a CH-based many-to-one algorithm to incrementally compute exact distances from many vertices towards a common target.

So far, research on route planning algorithms and traffic has mostly focused on the interactions between traffic and the preprocessing. For live traffic, speed-up techniques which have two preprocessing phases have been proposed. The first one may be slow but must be independent of any weight function. The second one, called *customization*, typically takes few seconds or less and allows regular traffic updates. *Multi-Level Dijkstra*, commonly referred to as *CRP*, was the first such three-phase technique and has since been extended into a comprehensive framework of routing algorithms [2]. With *Customizable Contraction Hierarchies* (CCH), CH has also been extended to support a three-phase setup [10]. Another line of research studies the integration of predicted traffic [7, 3, 4, 17]. Here, edge weights are functions of the daytime instead of scalar values.

The SSPP was initially introduced by Delling et al. in [8]. The authors discuss the complexity of the problem and show some relations between SSPP and Knapsack but no definitive conclusions could be drawn in their work. The paper also includes two CRP-based algorithms for the SSPP. *Iterative Path Blocking* (IPB) is presented as an exact algorithm for the SSPP. However, it has two issues: First, it takes several seconds even on short-range queries. This makes it unsuitable for practical applications. Second, as we show in this work, it is, in fact, not exact. The authors also present a heuristic algorithm based on the via-node paradigm, i.e. it finds solutions which are concatenations of two shortest paths. It is much faster but may miss promising paths because only via-paths are considered and the UBS is checked heuristically. We are not aware of any other works studying the SSPP.

In the SSPP, limiting the relative length of detours is formalized with the *uniformly bounded stretch* (UBS). The UBS is a path quality measure and quantifies how much longer detours on a path are than their respective fastest alternative. So far, it has been primarily studied in the context of alternative routes [1]. While quite useful, it is expensive to compute

and requires evaluating all subpaths of a path. The authors of [1] state that it would be ideal to check the UBS in time proportional to the length of the path and a few shortest path queries, though they are not aware of any way to do that. To the best of our knowledge, this goal has not been achieved to this day.

Contribution. In Section 3, we settle the complexity of the SSPP by proving that it is strongly NP-complete. Section 4 contains algorithmic results. First, we show that IPB as described in [8] may not find optimal results. Second, we describe necessary adjustments to make it exact. Third, we present an alternative realization based on A* and CH-Potentials [18]. Fourth, we present an efficient algorithm to compute exact UBS values, typically with only a few shortest path queries and in time proportional to the path length as the authors of [1] had hoped for. Fifth, we present Iterative Path Fixing, a new SSPP heuristic. All our algorithms utilize Lazy RPHAST as a crucial ingredient to achieve fast running times. Section 5 contains a thorough evaluation of our algorithms. It clearly shows the effectiveness of our UBS algorithm and our CH-Potentials-based IPB realization, outperforming the state of the art by up to two orders of magnitude.

2 Preliminaries

Let $G = (V, A)$ be a directed graph with $n = |V|$ vertices and $m = |A|$ arcs. We use uv as a short notation for arcs. A *weight function* $w : A \to \mathbb{N}$ maps arcs to positive integers. The *reversed* graph $\overleftarrow{G} := (V, \{vu \mid uv \in A\})$ contains all arcs in reverse direction. The corresponding reversed weight function is $\overleftarrow{w}(vu) := w(uv)$. A sequence of vertices $P = (v_1, \ldots, v_k)$ where $v_i v_{i+1} \in A$ is called a *path*. We denote by $P_{i,j} = (v_i, \ldots, v_j), 1 \leq i < j \leq k$ a *subpath* of P. The length of a path with respect to a weight function w is denoted by $w(P) = \sum w(v_i v_{i+1})$. We refer to a shortest path between two vertices s and t by $\mathrm{OPT}_w(s, t)$ and call its length the *distance* $\mathcal{D}_w(s, t)$ between s and t.

Dijkstra's algorithm [11] computes $\mathcal{D}_w(s, t)$ by traversing vertices by increasing distance from s until t is reached. Vertices are inserted into a priority queue when they are discovered. In each iteration the closest vertex u is popped from the queue and *settled*. Its distance is now final. Outgoing arcs uv are *relaxed*, i.e. the algorithm checks if the path from s to v via u is shorter than the previously known distance from s to v. If this is the case, v will be inserted into the priority queue. To keep track of the best-known distances, the algorithm maintains for each vertex v a tentative distance $\mathtt{D}[v]$. By storing the predecessor vertex on the shortest path from s to v in a parent array $\mathtt{P}[v]$, shortest paths can be efficiently reconstructed. By construction, Dijkstra's algorithm visits all vertices closer to s than the target. The visited vertices are sometimes called the *search space*. It can be reduced with the *A* algorithm* [15] by guiding the search towards the target. Here, the queue is ordered by $\mathtt{D}[v] + h_t(v)$ where h_t is a *heuristic* which estimates $\mathcal{D}(v, t)$.

2.1 (C)CH-Potentials

Contraction Hierarchies (CH) is a two-phase speed-up technique to accelerate shortest path computations on road networks through precomputation. For a detailed discussion we refer to [14]. Here, we only briefly introduce necessary notation and algorithms used in this paper. In a preprocessing phase, vertices are ordered totally by "importance" where more important vertices should lie on more shortest paths. Intuitively, vertices on highways are more important than vertices on some rural street. For CH, such an ordering is obtained heuristically. Then, all vertices are contracted successively by ascending importance. To

contract a vertex means to temporarily remove it from the graph while inserting *shortcut arcs* between more important neighbors to preserve shortest distances among them. The result is an *augmented graph* G^+ with original arcs and shortcuts. G^+ can be split into $G^\uparrow$ and $G^\downarrow$ where $G^\uparrow$ only contains arcs uv where u is less important than v and $G^\downarrow$ vice versa. The augmented graph has the property that for any two vertices s and t, there always exists an *up-down st*-path of shortest distance which first uses only arcs from $G^\uparrow$ and then only arcs from $G^\downarrow$. Such a path can be found by running Dijkstra's algorithm from s on $G^\uparrow$ and from t on the reversed downward graph $\overleftarrow{G^\downarrow}$ graph. Reconstructing the full path without any shortcuts is possible by recursively unpacking shortcuts. For this, one can store for each shortcut the vertex which was contracted when the shortcut was inserted. The set of vertices reachable in $G^\uparrow$ and $\overleftarrow{G^\downarrow}$ is called the *CH search space* of a vertex.

Algorithm 1 Computing the distance from a single vertex u to t with Lazy RPHAST.

Data: $\mathtt{D}^\downarrow[u]$: tentative distance from u to t computed by Dijkstra's algorithm on $\overleftarrow{G^\downarrow}$
Data: $\mathtt{D}[u]$: memoized final distance from u to t, initially $\perp$
Function `ComputeAndMemoizeDist`(u)**:**
  if $\mathtt{D}[u] = \perp$ **then**
    $\mathtt{D}[u] \leftarrow \mathtt{D}^\downarrow[u]$;
    for *all arcs* uv *in* $G^\uparrow$ **do**
      $d \leftarrow$ `ComputeAndMemoizeDist`(v);
      if $d + w(uv) < \mathtt{D}[u]$ **then**
        $\mathtt{D}[u] \leftarrow d + w(uv)$;
  return $\mathtt{D}[u]$;

Lazy RPHAST [18] is a CH-based algorithm to quickly compute distances from many sources to a single target. Lazy RPHAST starts by running Dijkstra's algorithm from t on $\overleftarrow{G^\downarrow}$, similar to a standard CH query. The forward search space, however, is explored through a recursive DFS-like search while memoizing distances to t as depicted in Algorithm 1. This allows reusing the already computed distances for following sources. Lazy RPHAST can be used analogously to compute distances from one vertex to many targets by swapping $G^\uparrow$ and $\overleftarrow{G^\downarrow}$. Using Lazy RPHAST as an A* heuristic is called *CH-Potentials* [18].

Customizable Contraction Hierarchies (CCH) [10] is a three-phase variant of CH. It allows fast updates to the preprocessing, for example to integrate information on the current traffic situation. However, this only affects the preprocessing. The result of the CCH preprocessing is also an augmented graph, only with some additional properties. The CH query algorithms and Lazy RPHAST can be applied without any modification. For the algorithms we discuss in this paper, there is no practical difference between CH and CCH, and we describe our algorithms based on augmented graphs. Our implementation is built on CCH-Potentials to support quick updates to live traffic weights. For a detailed discussion of the differences between CH and CCH and the changes to the preprocessing see [10].

2.2 Smooth Paths

The *stretch* of a path is defined as $S_w(P) = \frac{w(P)}{\mathcal{D}_w(v_1, v_k)}$, i.e. the ratio between the path length and the shortest distance between its endpoints. The *uniformly bounded stretch* $\mathrm{UBS}_w(P) = \max_{0 \le i < j \le k} S_w(P_{i,j})$ indicates the maximum stretch over all subpaths. We observe the following useful property of UBS:

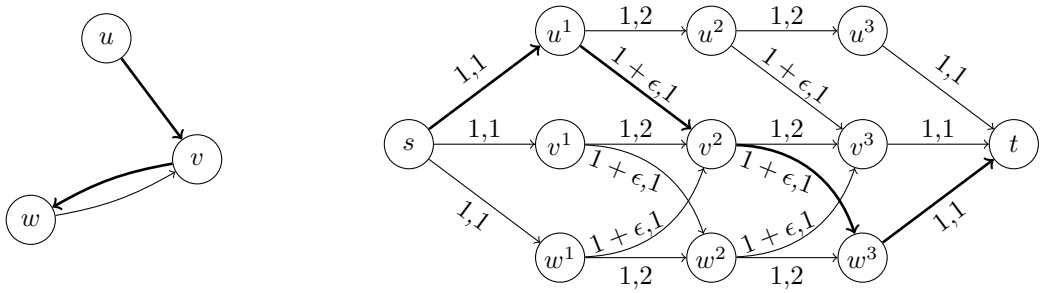

Figure 1 Illustration of our transformation from HamiltonPath to ShortestSmoothPath. The first arc weight is the smooth weight, the second the volatile weight. The thick arcs indicate a Hamiltonian path and the corresponding shortest ϵ-smooth path.

▶ **Observation 1.** *The UBS of a path* $P = (v_1, \ldots, v_i, \ldots, v_j \ldots, v_k)$ *where* $P_{1,i} = \mathrm{OPT}(v_1, v_i)$ *and* $P_{j,k} = \mathrm{OPT}(v_j, v_k)$ *is equal to* $\mathrm{UBS}(P_{i,j})$.

This is because the stretch of any subpath only decreases when appending optimal segments to the beginning or end.

In [8], Delling et al. introduce the shortest smooth path problem (SSPP). A path P is ϵ-smooth with respect to a weight function w when $\mathrm{UBS}_w(P) < 1 + \epsilon$. Given a graph G, vertices s and t, a smooth weight function w and a volatile weight function w^* and a parameter $\epsilon > 0$, the shortest smooth path problem is to find the shortest path with respect to w^* that is ϵ-smooth with respect to w.

3 Complexity

In this section, we prove that SSPP is strongly NP-complete for any $\epsilon \in \mathbb{Q}^{>0}$. We define the decision variant of the problem as follows: An instance (G, w, w^*, s, t, k) of the of ϵ-ShortestSmoothPath-Dec problem admits a feasible solution if and only if there exists a path $P = (s, \ldots, t)$ in G with $w^*(P) \leq k$ and $\mathrm{UBS}_w(P) < 1 + \epsilon$.

▶ **Theorem 2.** *ϵ-ShortestSmoothPath-Dec is strongly NP-complete for any $\epsilon \in \mathbb{Q}^{>0}$.*

Proof. A solution can be verified in polynomial time. Determining the path weight in w^* takes running time linear in $\mathcal{O}(|P|)$. To check the UBS, shortest distances have to be computed for all $\mathcal{O}(|P|^2)$ subpaths. This shows that ShortestSmoothPath-Dec $\in$ NP.

To prove the hardness, we give a reduction from the strongly NP-complete HamiltonPath problem [13]. The goal is to find a *Hamiltonian* path, i.e. a simple path which traverses every vertex exactly once. Let $G = (V, A)$ be the HamiltonPath instance. To distinguish them from the vertices in the SSPP instance, we will denote the vertices in the HamiltonPath instance as *nodes*. We construct the vertices of our SSPP instance by copying each node $|V|$ times (forming $|V|$ layers) and creating two additional vertices s and t. Arcs only connect successive layers. There are arcs between vertices corresponding to the same node and arcs corresponding to arcs from the HamiltonPath instance. Any $(s, \ldots, t)$ path has exactly $|V| + 1$ arcs and has to traverse all layers. We will choose the arc weights in such a way that the shortest ϵ-smooth path between s and t has to use a different node in each layer. Paths using the same node in different layers will always be non-ϵ-smooth in w or too long in w^*.

Formally, we construct the graph $G' = (V', A')$ for our SSPP instance as follows: We set the vertices $V' = \{v^i \mid v \in V, i \in [1, n]\} \cup \{s, t\}$. The arc set A' is the union of three groups of arcs A_{orig}, A_{self} and A_{terminal} where $A_{\text{orig}} = \{(u^i, v^{i+1}) \mid uv \in A, 1 \leq i < n\}$

are the arcs between the layers corresponding to arcs in the HamiltonPath instance, $A_{\text{self}} = \{(v^i, v^{i+1}) \mid v \in V, 1 \leq i < n\}$ are the additional arcs between the same nodes in successive layers and $A_{\text{terminal}} = \{(s, v^1) \mid v \in V\} \cup \{(v^n, t) \mid v \in V\}$ are the arcs connecting the terminals with the first and last layer. In both weight functions, all arcs $a_{\text{term}} \in A_{\text{terminal}}$ get the weight $w(a_{\text{term}}) = w^*(a_{\text{term}}) = 1$. The arcs in $a_{\text{self}} \in A_{\text{self}}$ get a smooth weight $w(a_{\text{self}}) = 1$ and a volatile weight $w^*(a_{\text{self}}) = 2$. The arcs in $a_{\text{orig}} \in A_{\text{orig}}$ get a smooth weight $w(a_{\text{orig}}) = 1 + \epsilon$ and a volatile weight $w^*(a_{\text{orig}}) = 1$. Setting $k = |V| + 1$, this forms our ShortestSmoothPath-Dec instance. This transformation has a running time in $\mathcal{O}(n \cdot (n + m))$. See Figure 1 for an illustrated example of the construction. For the sake of readability, we use non-integer weights of $1 + \epsilon$ in this proof. The weights can be turned into integers by multiplying them with the denominator of ϵ.

Now, assume that the HamiltonPath instance admits a Hamiltonian path $P = (v_1, \ldots, v_n)$. Then, $P' = (s, v_1^1, \ldots, v_n^n, t)$ is a solution to the SSPP problem. The path uses two arcs from A_{terminal} and $|V| - 1$ arcs from A_{orig}. Thus, $w^*(P') = |V| + 1 = k$. Also, its $\text{UBS}_w(P')$ must be smaller than $1 + \epsilon$. Due to Observation 1, it is sufficient to show that the UBS is small enough for $P'' = (v_1^1, \ldots, v_n^n)$. For any subpath $P''_{i,j} = (u^i, \ldots v^j)$ for $i < j$, u must not be equal to v because P is a Hamiltonian path. As all arcs are from A_{orig}, $w(P''_{i,j}) = (j - i) \cdot (1 + \epsilon)$. The shortest path (with respect to w) between u^i and v^j has to use at least one A_{orig} arc because $u \neq v$. Thus, $\mathcal{D}_w(P''_{i,j}) \geq (j - i - 1) + (1 + \epsilon)$. This yields

$$\text{UBS}_w(P''_{i,j}) = \frac{w(P''_{i,j})}{\mathcal{D}_w(P''_{i,j})} \leq \frac{(j-i)\cdot(1+\epsilon)}{j-i+\epsilon} < \frac{(j-i)\cdot(1+\epsilon)}{j-i} = 1 + \epsilon$$

which proves that P' is a valid solution for the SSPP instance.

Conversely, suppose that our SSPP instance has an ϵ-smooth path $P' = (s, v_1^1, \ldots, v_n^n, t)$ of weight $w^*(P') = |V| + 1$. Such a path cannot contain any arcs from A_{self} because their volatile weight is 2. We now show that no two vertices in the path can correspond to the same node and thus that $P = (v_1, \ldots, v_n)$ is indeed a Hamiltonian path in G. Suppose for contradiction that $P'_{i,j} = (v^i, \ldots, v^j)$ was a subpath of P'. The length $w(P'_{i,j})$ is $(i-j)\cdot(1+\epsilon)$. Since start and end vertex correspond to the same node, the shortest path with respect to w between these vertices is made up of arcs from A_{self} and has distance $\mathcal{D}_w(v^i, v^j) = i - j$. Thus, $\text{UBS}_w(P'_{i,j}) = (1 + \epsilon)$ which means that this subpath must not be part of a solution for the SSPP instance. This is a contradiction. Thus, the SSPP solution induces a valid solution for the HamiltonPath instance. ◀

4 Algorithms

In [8], the *Iterative Path Blocking* (IPB) algorithm is proposed to solve the SSPP optimally. The algorithm repeats two steps until a valid path is found. It maintains a set of blocked paths, which is initially empty. In the first step, a shortest path with respect to w^* is computed while avoiding any blocked paths. In the second step, the obtained path is checked for subpaths violating the UBS constraint. Any violating subpaths are added to the list of blocked paths and the algorithm continues with the next iteration. If no violating subpath is found, the final path is returned.

This framework can be implemented with different concrete algorithms for both steps. The implementation described in [8] is based on CRP [5]. In this paper, we propose optimized implementations for both steps based on Lazy RPHAST and (C)CH-Potentials.

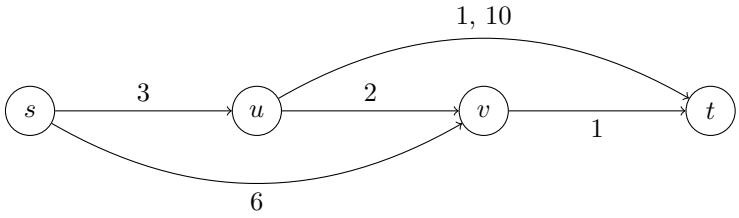

Figure 2 Example graph where for $\epsilon = 1$ the shortest ϵ-smooth path (s, v, t) is not prefix-optimal. For all arcs except ut, the smooth and the volatile weight function are equal. For ut, the smooth weight is 1 and the volatile weight 10.

4.1 Avoiding Blocked Paths

The authors of [8] describe their approach to the first phase as a variant of Dijkstra's algorithm. When relaxing an arc uv where v is the endpoint of a blocked path, they backtrack the parent pointers of v, comparing the reconstructed path to the blocked path. Should the paths match, the search is pruned at v. This algorithm correctly avoids blocked paths. However, it also avoids some additional paths because Dijkstra's algorithm by construction only finds prefix-optimal paths. But optimal shortest smooth paths may not be prefix-optimal with respect to the volatile weight function w^*. See Figure 2 for an example. To the best of our understanding, IPB as described in [8] will not find the shortest smooth path in this example. The algorithm will find the path (s, u, v, t) in the first iteration. This path is not 1-smooth because (u, v, t) has stretch 3 and (u, v, t) will be added to the blocked path set. With (u, v, t) blocked, the algorithm will find the path (s, u, t) in the next iteration and return it as the final result. However, the shortest 1-smooth path is (s, v, t). It was missed because the prefix (s, v) is not optimal in w^* and was therefore pruned at v by (s, u, v). We will refer to this variant from now on by *heuristic iterative path blocking* (IPB-H). IPB-H will still find an ϵ-smooth path though it may not necessarily be the shortest.

To find shortest smooth path with Dijkstra's algorithm, we need to adjust the notion of optimality used to compare labels. It might be necessary to keep a label with suboptimal distance from the start as in the example from Figure 2 where the label for (s, v) needs to be kept at v despite being longer than (s, u, v). This leads to a *label-correcting* variation of Dijkstra's algorithm with possibly multiple labels per vertex. A *label* l at a vertex v consists of a distance $\mathtt{D}_l$ from the source, a set of active blocked paths $\mathtt{A}_l$, and a pointer to the parent vertex and label for efficient reconstruction of the labels' path $P(l) = (s, \dots, v)$. The active blocked path set $\mathtt{A}_l$ contains all blocked paths which have a prefix which is a suffix of $P(l)$. A label l can be discarded when v has another label l' with $\mathtt{D}_{l'} \leq \mathtt{D}_l$ and $\mathtt{A}_{l'} \subseteq \mathtt{A}_l$.

The search is initialized with a single label at s with distance zero and an empty set of active blocked paths. When a vertex u with a label l_u is popped from the queue and an arc uv is relaxed, we create a new label l_v as follows: We set the distance $\mathtt{D}_{l_v} = \mathtt{D}_{l_u} + w^*(uv)$ and the parent label to l_u. We also need to keep track of traversed blocked paths. If uv is the first arc of a blocked path $B = (u, v, \dots)$, the path B needs to be added to $\mathtt{A}_{l_v}$. For any active blocked path $B = (\dots, u, x, \dots) \in \mathtt{A}_{l_u}$, we need to check if $x = v$, i.e. uv lies on B. If this is the case, B is contained in $\mathtt{A}_{l_v}$, or, if uv is the last arc of B, the label l_v must be dropped. If uv is not on B, the blocked path is not in $\mathtt{A}_{l_v}$.

An efficient implementation of this algorithm requires careful engineering. For each arc, we keep track of the blocked paths it lies on. Labels use a bitset to store the active blocked paths. This allows for efficient subset checks with bit-wise operations. The bitset size is

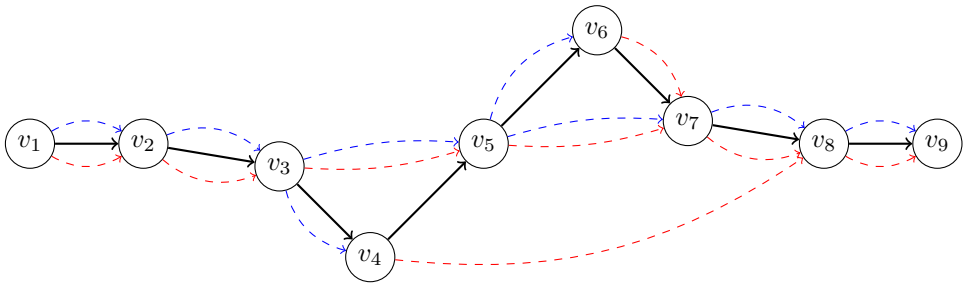

Figure 3 Example path (solid, black) with shortest path tree from v_1 to all vertices on the path (dashed, blue) and reverse shortest path tree from all vertices on the path to v_9 (dashed, red).

determined individually for each vertex by the number of blocked paths the respective vertex lies on. In our implementation, we use at least one 128-bit integer which suffices for most queries. Should the number of blocked paths for a vertex exceed 128, we switch to using a dynamically sized array of integers for that vertex. Additionally, each vertex maintains its own queue of labels ordered by distance from s. When the vertex is popped from the queue, it pops the next label from its queue and propagates only this label. If there are any remaining labels in the queue, the vertex is reinserted into the global queue. Finally, we utilize A* with CCH-Potentials on the volatile weight function to guide the search towards the target. As our experiments show, disallowing non-ϵ-smooth paths increases distances only very slightly. Thus, the heuristic is close to perfect and A* very effective for this problem.

4.2 Efficient UBS Computation

According to Delling et al. [8], the UBS computation is one of the bottlenecks of the IPB approach. They employ a many-to-many algorithm. Here, we introduce an algorithm which can compute exact UBS values of typical paths with only a few shortest path queries. We also present a worst-case example where each subpath has a distinct stretch value. This suggests that achieving subquadratic worst-case running time may not be possible.

Consider a path $P = (s = v_1, \dots, t = v_k)$ as depicted in Figure 3. Our algorithm works iteratively. We start with the full path and successively remove prefixes and suffixes until the path is empty or only a shortest path remains. We start by computing shortest distances from s to all vertices on the path. This can be done with a single run of Dijkstra's algorithm which can terminate once all v_i have been settled. Beside the shortest distances, this yields a shortest path tree represented though parent pointers. We also run Dijkstra's algorithm from t on the reversed graph which yields a backward shortest path tree to t. Now we find the greatest index i such that $P_{1,i}$ is a prefix of all shortest paths $\mathrm{OPT}(s, v_l)$ where $i < l \leq k$, i.e. the first branching vertex in the forward shortest path tree. In the worst case this may be s. In the example in Figure 3 this is v_3. We analogously obtain the first branching vertex in the reverse shortest path tree to v_k (v_8 in our example). Stated formally, this is the smallest index j such that $P_{j,k}$ is a suffix of all shortest paths $\mathrm{OPT}(v_l, t)$ where $1 \leq l \leq j$. By Observation 1, subpaths starting from vertices in the segment $P_{1,i-1}$ and subpaths ending at vertices from $P_{j+1,k}$ are not relevant to the UBS computation. We exploit this and only check paths starting from v_i or ending at v_j in the current iteration.

We check the stretch of all subpaths $P_{i,l}$ where $i < l \leq j$ with a linear sweep over the v_l. Since $P_{1,i}$ is a prefix of all shortest paths from s, we can compute the distance $\mathcal{D}(v_i, v_l)$ as $\mathcal{D}(s, v_l) - \mathcal{D}(s, v_i)$. Thus, each stretch can be checked in constant time with the distances computed by Dijkstra's algorithm. When we are only interested in violating

Algorithm 2 Path unpacking for Lazy RPHAST.

Data: $\texttt{P}[u]$: parent vertex on the shortest path from u to t, as computed by Dijkstra's algorithm on $\overleftarrow{G^{\downarrow}}$ and an extended Algorithm 1
Data: $\texttt{U}[u]$: whether the path from u to t has been fully unpacked
Function $\texttt{Unpack}(u)$**:**
  if $\neg(\texttt{U}[u] \vee u = t)$ **then**
    $\texttt{ComputeAndMemoizeDist}(u)$;
    $\texttt{Unpack}(\texttt{P}[u])$;
    if $(u, \texttt{P}[u])$ *is a shortcut for* $(u, v, \texttt{P}[u])$ **then**
      $\texttt{P}[v] \leftarrow \texttt{P}[u]$;
      $\texttt{Unpack}(v)$;
      $\texttt{P}[u] \leftarrow v$;
      $\texttt{Unpack}(u)$;
    $\texttt{U}[u] \leftarrow$ **true**;

subpaths (rather than computing the exact UBS value of P), the sweep can be stopped after the first (i.e. shortest) violating segment has been found. Forbidding the shortest violating segment starting at v_i is sufficient because it is contained in all longer segments. Checking the stretches of the subpaths $P_{l,j}$ where $i \leq l < j$ works analogously.

Having checked all these stretches, we continue with the next iteration by applying the whole algorithm to the subpath $P_{i+1,j-1}$. We can stop when $i+1 \geq j-1$ or when the entire considered path is a shortest path between its endpoints.

This algorithm can be adopted to efficiently compute other path quality measures such as the *local optimality* [1].

4.2.1 Worst-Case Running Time

This algorithm performs great when long segments are shortest paths, which will often be the case when searching shortest smooth paths. But in the worst case, it still has to check $\Theta(n^2)$ subpath stretches. Consider a complete graph with unit weights and the path $P = (v_1, \ldots, v_n)$. In this graph, the shortest path between any two vertices is always the direct arc and the distance is exactly one. Thus, the shortest path tree from any vertex is a star with the direct arcs and our algorithm can only advance by a single vertex in each iteration. This results in a worst case running time of n runs of Dijkstra's algorithm.

We suspect that it is not possible to compute the UBS asymptotically faster. Consider the same graph as before but with weights of unique powers of two for the arcs of the path. Now any subpath has a unique length. As all subpaths of three or more vertices still have a shortest distance of one between their endpoints, there are $\Theta(n^2)$ unique stretch values. Thus, computing the UBS of the whole path without checking all $\Theta(n^2)$ stretch values should be difficult if not impossible.

4.2.2 Lazy RPHAST with Path Unpacking

While this algorithm typically needs few stretch checks, running Dijkstra's algorithm a couple of times is still prohibitively slow on large road networks. Luckily, we can speed these computations up drastically by employing Lazy RPHAST, which we already used as an A* heuristic in the shortest path search phase. Recall that Lazy RPHAST allows us to select

one target vertex and then to compute shortest distances quickly from many vertices to this target. For the efficient UBS computation, we use two instantiations of this algorithm. In each iteration, we select both endpoints of the considered path and compute distances from and to the endpoints for all vertices on the path. However, we also need the shortest path trees. We therefore extend Lazy RPHAST to also compute shortest path trees.

Dijkstra's algorithm on $\overleftarrow{G^{\downarrow}}$ yields initial parent pointers. We adjust Algorithm 1 to continue to maintain these parent pointers during arc relaxation. Thus, after having called `ComputeAndMemoizeDist`, we have the shortest path through the CH search space in G^+. Algorithm 2 depicts the routine to efficiently unpack shortcuts on this path and retrieve shortest path trees in the original graph. We use a bitvector `U` (using a clearlist for fast reinitialization) to mark vertices for which the shortest path has already been fully unpacked which is checked before any actual work is performed. Then, we have to call `ComputeAndMemoizeDist` to ensure that the path through the CH search space has been obtained for u. For vertices encountered through recursive shortcut unpacking this might have not happened before. In the next step, we can now recursively unpack the full path up to the parent $\texttt{P}[u]$ of our current vertex u. Now, all that remains is to unpack the arc $(u, \texttt{P}[u])$ if it is a shortcut. If so, the middle vertex v is set in `P` as the vertex between u and $\texttt{P}[u]$ and `Unpack` is invoked recursively first for v and then again for u to unpack the arcs $(v, \texttt{P}[v])$ and (u, v).

4.3 Iterative Path Fixing

With an efficient algorithm to find UBS-violating segments we can introduce another natural heuristic to find short smooth paths: Find the shortest path with respect to w^* and replace each UBS violating subpath $(v_i, \ldots, v_j)$ with $\text{OPT}_w(v_i, v_j)$. The result may still contain UBS violating subpaths. In this case, we iteratively continue to replace violating segments. When a path contains overlapping violating subpaths, we replace the first, ignore following overlapping subpaths and continue with the next non-overlapping segment. We denote this algorithm as *iterative path fixing* (IPF).

5 Evaluation

In this section, we present our experimental results. Our benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.5 GHz and 8 × 64 KiB of L1, 8 × 1 MiB of L2, and 24.75 MiB of shared L3 cache. All running times are sequential. We implement our algorithms in Rust[1] and compile them with `rustc 1.58.0-nightly (b426445c6 2021-11-24)` in the release profile with the `target-cpu=native` option.

Table 1 shows the road networks we use in our experiments alongside sequential preprocessing times. OSM Europe is the same network used in [8] and publicly available.[2] The DIMACS Europe instance was made available by PTV[3] for the 9th DIMACS implementation challenge [9]. It is not publicly available but can be obtained on request for research purposes[4]. We derived the OSM Germany instance from an early 2020 snapshot of OpenStreetMap and

[1] The code for this paper, all implemented algorithms, scripts to perform experiments and to aggregate the results is available at `https://github.com/kit-algo/traffic_aware`
[2] `https://i11www.iti.kit.edu/resources/roadgraphs.php`
[3] `https://ptvgroup.com`
[4] `https://i11www.iti.kit.edu/resources/roadgraphs.php`

Table 1 Instances used in the evaluation with sequential preprocessing running times to construct a CCH-Potential. Phase 1 needs to be run only once for each graph, Phase 2 once for each weight function, or when a weight function changes.

	Vertices	Edges	Preprocessing [s]	
	$[\cdot 10^6]$	$[\cdot 10^6]$	Phase 1	Phase 2
DIMACS Europe	18.0	42.2	2 260.7	11.3
OSM Europe	173.8	348.0	4 270.0	58.8
OSM Germany	11.1	26.2	1 314.0	7.5

converted into a routing graph using RoutingKit.[5] For this instance, we have proprietary traffic data provided by Mapbox[6] which, unfortunately, we cannot provide. The data includes two live traffic snapshots in the form of OSM node ID pairs and live speeds for the edge between the vertices. One is from Friday 2019/08/02 afternoon and contains 320K vertex pairs and the other from Tuesday 2019/07/16 morning and contains 185K pairs. For both Europe instances, we do not have any real world traffic data. Thus, we resort to the approach suggested in [8] and generate synthetic live traffic: For each road where the average speed is greater than 30 kph, we reduce the speed to 5 kph with a probability of 0.5%.

We evaluate our algorithms by performing batches of point-to-point shortest ϵ-smooth path queries. As the distance between source and target has a significant influence on the performance, we generate different query batches. For each batch, we pick 1000 source vertices uniformly at random. We then run Dijkstra's algorithm from each source vertex on the graph with the smooth weight function. Following the Dijkstra rank methodology, we store every 2^ith settled vertex [16]. This allows evaluating the performance development against varying path lengths. In [8], 1-hour queries were performed. For comparison, we also generate an *1h* batch by picking the first settled vertex with a distance greater than one hour. In addition, we generate a *4h* batch for medium-range queries with the same method and a *random* batch for long range queries where the target is picked uniformly at random.

Preliminary experiments showed that some queries take prohibitively long to answer. Since we are solving an NP-hard problem, this is not very surprising. We abort queries if the algorithm has not found a path with UBS $< (1 + \epsilon)$ after 10 seconds. We report these queries as failed but, nevertheless, do include their running times in our measurements.

We start by evaluating different UBS algorithms in isolation. The paths checked by the UBS algorithms are the paths we find while performing IPB-H to find shortest smooth paths with $\epsilon = 0.2$ on the Dijkstra rank queries. We limit the time per rank and UBS algorithm to one hour. Thus, slow algorithms may not get to check all paths. Our baseline is computing all distances between pairs of path vertices at once with *SSE RPHAST* [6], which to the best of our knowledge is the fastest known many-to-many algorithm. The second algorithm, denoted as *Lazy RPHAST Naive*, uses Lazy RPHAST to compute distances between all pairs of path vertices. The third one is *UBS Trees Dijkstra* which is the non-accelerated, i.e. Dijkstra-based, implementation of the efficient UBS algorithm introduced in Section 4.2. *UBS Trees Lazy RPHAST* denotes the accelerated variant of this algorithm utilizing Lazy RPHAST as described in Section 4.2.2.

[5] `https://github.com/RoutingKit/RoutingKit`
[6] `https://mapbox.com`

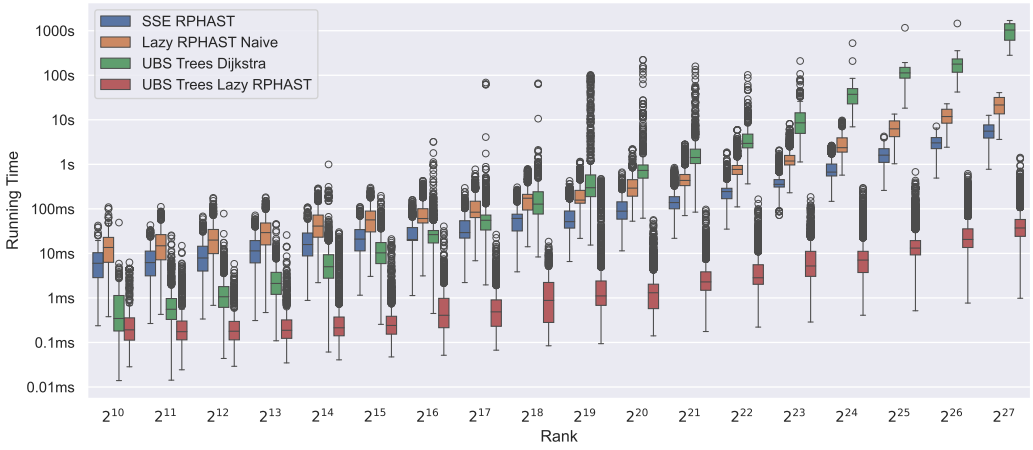

Figure 4 Running times of different UBS checking algorithms for paths encountered by IPB-H when answering queries of different ranks with $\epsilon = 0.2$ on OSM Europe. The boxes cover the range between the first and third quartile. The band in the box indicates the median, the whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

Figure 4 depicts the results of this experiment. We observe that SSE RPHAST is consistently faster than the naive Lazy RPHAST variant by a roughly constant factor. SSE RPHAST was designed as a many-to-many algorithm and is thus more efficient than naively applying a many-to-one algorithm $|P|$ times. The non-accelerated UBS Trees algorithm is very fast for short paths but quickly becomes prohibitively slow for longer paths. Running Dijkstra's algorithm will traverse a large part of the network if source and target are sufficiently far apart from each other. Doing this multiple times is not feasible. However, the accelerated variant beats SSE RPHAST by about two orders of magnitude across all path lengths.

UBS Trees running times have significantly greater variance than the many-to-many algorithms. This is because the amount of work which UBS Trees can avoid varies strongly between different paths. In contrast, the many-to-many-based algorithms will always check $\Theta(|P|^2)$ subpath distances. Note that the UBS Trees Dijkstra outliers disappear because we limit the time per rank and algorithm. If we checked all paths, the outliers would be present too, but the experiment would take prohibitively long.

Next, we evaluate the performance of our query algorithms on realistic queries and instances. Table 2 depicts the results. Both the query set and the instance have a strong influence on the running time. Note that random queries on OSM Germany are on average shorter than four hours which is the reason why the running times on OSM Germany for random queries are faster than for 4h queries. The length increase of the solutions primarily depends on the instance and less on the query set. The synthetic traffic affects DIMACS Europe more strongly than OSM Europe. We suspect that this is because OSM is modeled in much greater detail and contains more shorter arcs. In terms of running time, IPB-H is significantly faster than IPB-E and IPF is significantly faster still, which is roughly what we expected. Conversely, the heuristics find somewhat longer paths than the exact IPB-E algorithm and IPF appears to find worse paths than IPB-H. However, one has to be careful interpreting these numbers as a non-negligible amount of queries did not terminate with IPB-E and IPB-H. Because the length increase numbers are averages over different sets, it is not immediately clear if the differences appear because the heuristics find worse paths or because the heuristics find long solutions where the exact algorithm did not finish within

Table 2 Average performance of our implementations of IPB-E, IPB-H and IPF for different query sets on all instances with $\epsilon = 0.2$. The Increase column denotes the length increase with respect to w^* of the obtained path over OPT_{w^*} and includes only successful queries. The running time column also includes the running time of queries aborted after 10 seconds.

		Increase [%]			Running time [ms]			Failed [%]		
		IPB-E	IPB-H	IPF	IPB-E	IPB-H	IPF	IPB-E	IPB-H	IPF
1h	DIMACS Eur Syn	0.8	2.5	4.1	718.0	168.4	1.5	6.4	1.6	0.0
	OSM Eur Syn	0.2	0.3	0.3	59.8	22.4	2.7	0.4	0.2	0.0
	OSM Ger Fri	0.2	1.5	2.2	1 373.7	219.1	4.7	12.7	1.2	0.0
	OSM Ger Tue	0.1	0.3	0.4	261.5	9.2	1.8	2.2	0.0	0.0
4h	DIMACS Eur Syn	0.8	3.4	5.1	3 513.6	435.4	6.1	33.1	3.1	0.0
	OSM Eur Syn	0.2	0.3	0.3	331.4	73.2	8.4	2.1	0.6	0.0
	OSM Ger Fri	0.2	2.1	4.2	6 597.1	2 568.1	89.2	63.4	15.8	0.0
	OSM Ger Tue	0.1	0.4	0.5	1 449.3	93.1	9.8	13.1	0.0	0.0
Random	DIMACS Eur Syn	0.8	2.8	5.3	6 700.6	2 436.7	30.6	64.2	17.1	0.0
	OSM Eur Syn	0.2	0.4	0.4	4 758.3	654.2	140.3	38.9	3.1	0.0
	OSM Ger Fri	0.2	2.1	4.1	5 771.1	2 419.8	84.5	56.0	16.3	0.0
	OSM Ger Tue	0.1	0.4	0.5	1 366.0	111.6	9.6	12.0	0.0	0.0

10 seconds. For running times, the averages are also difficult to interpret. They are heavily skewed by outliers and there is no reason to assume a normal distribution. In fact, *median* running times for 1h queries of all algorithms on all instances are all below 2 ms. Clearly, drawing statistically sound conclusions from this experiment requires a closer look.

Figure 5 depicts *performance profiles* [12] for running times and obtained path lengths on all queries from Table 2 combined. Investigating queries across all instances combined is reasonable because we study the relative performance of the different algorithms on each query. Let $\mathcal{A}$ be the set of algorithms, $\mathcal{Q}$ the set of queries and $\text{obj}(a, q)$ denote the considered measurement from the computation of $a \in \mathcal{A}$ to answer $q \in \mathcal{Q}$. In our case, this is either the running time or the length with respect to w^* of the computed path. The performance ratio $r(a, q) = \frac{\text{obj}(a,q)}{\min\{\text{obj}(a',q) | a' \in \mathcal{A}\}}$ indicates by what factor a deviates from the best solution or the shortest running time for the query q. The performance profile $\rho_a : [1, \infty) \to [0, 1], \tau \mapsto \frac{|\{q \in \mathcal{Q} | r(a,q) \leq \tau\}|}{|\mathcal{Q}|}$ of a is the fraction of queries for which a is within a factor of τ of the best measurement. For computations that were aborted after 10 seconds, $\text{obj}(a, q) = \infty$. For the sake of completeness, we also include the same performance profiles separated per instance and query set in the appendix (see Figure 6 and 7). However, discussing the results in such detail is beyond the scope of this paper.

The running time performance profile in Figure 5 allows for some more nuanced observations: IPF is the fastest algorithm on about 65% of the queries and almost never more than 10 times slower than the fastest one. Surprisingly, IPB-H is also sometimes the fastest to answer a query (in 35% of the queries) but it may also be up to 300 times slower than the fastest algorithm. However, for 83% of all queries it stays within a factor of 10. The exact algorithm is never the fastest but still within a factor of 10 for 65% of the queries. It still may be several thousand times slower than the fastest algorithm in extreme cases, even with the running time limited to 10 seconds.

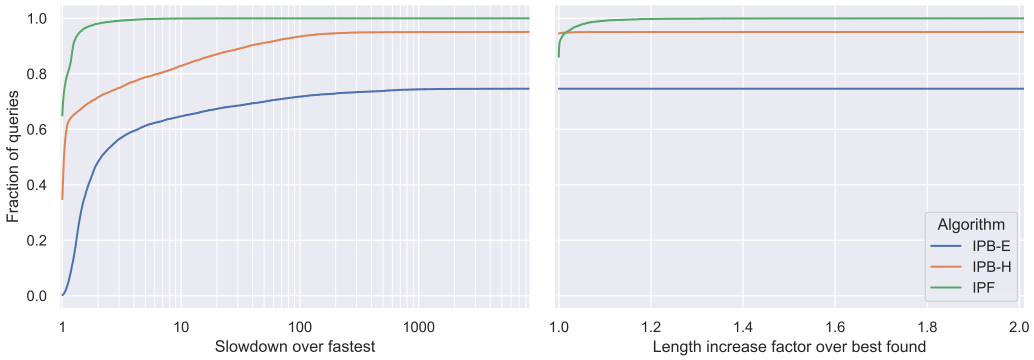

Figure 5 Relative performance profiles of our algorithms on all queries from Table 2.

The path length performance profile also yields useful insights. Since IPB-E is an exact algorithm, its performance profile contains only a single data point, i.e. for all queries which terminated successfully IPB-E finds the shortest path. The line for IPB-H is almost constant. This means, there are only few queries where it does not find the best solution. Even when it does not find the best solution, it is close to the best one, i.e. the maximum length increase factor over the best solution is 1.36 and all other values are below 1.1. It is quite possible that IPB-H found the optimal solution even for some queries where IPB-E did not terminate. The qualitative performance of IPF varies more strongly. It also finds the best solution on 85% of the queries. More than 99% of the obtained solutions are within a factor of 1.2 to the best found. In the worst case IPF found a path 1.96 times the length of the best one found by another algorithm.

In combination with the averages reported in Table 2 we can now draw solid conclusions on the performance of the algorithms. IPF is the algorithm with the most stable running time. Even though it is not always the fastest, it is never much slower than any other algorithm. It is the only algorithm able to answer all queries in less than 10 seconds. In fact, it usually needs only a few milliseconds and only up to several hundreds of milliseconds for extreme cases. It sometimes pays for this with worse solution quality but is still very close to the best found for the vast majority of queries. This makes it an algorithm suitable for practical applications. IPB-H is also a very effective heuristic. It is drastically faster than the exact algorithm and sometimes even faster than IPF. Its performance in terms of quality is much more stable than IPF and often IPB-H will find the best path or something very close to it. The difference in average length increase between IPB-E and IPB-H was not because IPB-H finds much worse paths but because it is able to answer queries which IPB-E cannot answer. However, it still fails to answer about 5% of all queries in less than 10 seconds. The running time of IPB-E varies even more strongly. On the one hand, many easy queries can be answered in a few milliseconds, but on the other hand, 25% of all queries cannot be answered in less than 10 seconds. The feasibility of solving the problem to exactness with IPB-E strongly depends on the distance of queries and on the smoothness of w^*.

For our final experiment, we evaluate the performance of our algorithms with different choices for ϵ with 1000 queries of 1h range on OSM Europe. Table 3 depicts the results. This experiment was also performed in [8] but with only 100 queries. Given the observation from the previous experiment, it should be clear that reported averages allow only for very rough comparisons. However, it is the only data available to compare against related work. Also

Table 3 Average performance of our implementations of IPB-E, IPB-H and IPF for different values of ϵ with 1h queries on OSM Europe with synthetic live traffic. The Increase column denotes the length increase with respect to w^* of the shortest smooth path over the shortest w^* path. It includes only values from successful queries. All other columns indicate average values over all queries, including the ones terminated after 10 seconds.

		Increase	Iterations	Blocked	Running time [ms]			Failed
ϵ		[%]		paths	A*	UBS	Total	[%]
	IPB-E	0.43	137.90	676.2	307.6	22.7	335.9	2.4
0.01	IPB-H	0.56	22.38	24.9	52.8	21.0	74.0	0.6
	IPF	0.61	1.73	-	-	-	2.3	0.0
	IPB-E	0.34	68.10	351.7	132.5	14.8	150.3	0.9
0.05	IPB-H	0.39	32.78	39.8	19.6	38.7	58.6	0.5
	IPF	0.41	1.54	-	-	-	2.3	0.0
	IPB-E	0.27	47.35	256.4	103.3	12.7	118.3	0.8
0.10	IPB-H	0.33	27.10	27.1	3.5	28.9	32.7	0.3
	IPF	0.34	1.45	-	-	-	2.7	0.0
	IPB-E	0.23	24.92	141.7	51.1	7.5	59.7	0.4
0.20	IPB-H	0.26	19.33	19.0	2.6	19.6	22.4	0.2
	IPF	0.28	1.36	-	-	-	2.1	0.0
	IPB-E	0.16	13.64	80.0	41.1	3.8	45.6	0.1
0.50	IPB-H	0.17	19.54	18.9	2.5	19.4	22.1	0.2
	IPF	0.19	1.26	-	-	-	2.0	0.0
	IPB-E	0.11	10.51	55.5	28.1	4.4	33.4	0.2
1.00	IPB-H	0.12	15.13	14.3	2.4	9.6	12.2	0.1
	IPF	0.14	1.19	-	-	-	2.5	0.0

note that due to the presence of heavy outliers, performing too few queries can distort the numbers drastically. For example, when we ran the same experiment with only 100 queries, the average running times of IPB-H were an order of magnitude faster.

We observe similar trends as the authors of [8]. The smaller the choice of ϵ, the harder the problem becomes. Consequently, the length increase, the number of iterations, the number of blocked paths and the running time increase. However, for our implementation of IPB-H, we measure slightly bigger path increases and slightly more iterations. Our implementation of IPB-H achieves running times two orders of magnitude faster than the CRP-based IPB-H implementation in [8]. One reason for this is our UBS algorithm which only needs a couple of milliseconds for all values of ϵ. In [8], the UBS checking phase takes between 1.3 and 1.9 *seconds*. The CH-Potentials-based shortest path phase is also very efficient across the entire range of ϵ values. Even with many blocked paths, the path lengths increase only little and the CH-Potentials heuristic remains tight and yields good speed-ups. Our exact IPB-E implementation is still an order of magnitude faster than the IPB-H implementation in [8].

6 Conclusion

In this paper, we studied the shortest smooth path problem and proved its NP-completeness. We introduced a new algorithm for practically efficient UBS computation. This algorithm can compute the exact UBS of typically occurring paths with very few shortest path computations. It outperforms state-of-the-art exact UBS algorithms by around two orders of magnitude and makes computing exact UBS values feasible in practice. Also, it can be used for other path quality measures such as local optimality.

We adapted the existing IPB-H algorithm and realized it with our new UBS algorithm and A* with CH-Potentials. This realization of IPB-H outperforms the original implementation by two orders of magnitude. Also, we present necessary modifications to make the algorithm exact. IPB-E is still about an order of magnitude faster than the CRP-based heuristic implementation. As IPB-H and IPB-E are not always able to find solutions in reasonable time, we introduce another heuristic, IPF. It can consistently find smooth paths even for random queries on massive continental sized instances in a few tenths of milliseconds.

For future work we would like to apply our algorithms not only to live traffic but also to predicted traffic, i.e. find smooth paths in a time-dependent setting. Further, it would be interesting to study what causes IPB-H to be so much faster than IPB-E while retaining most of the quality. Maybe this could be traced to specific structures in road networks which then could be exploited to speed up IPB-E.

References

1 Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. *ACM Journal of Experimental Algorithmics*, 18(1):1–17, 2013.

2 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

3 Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.

4 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic Time-Dependent Route Planning in Road Networks with User Preferences. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*, volume 9685 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2016.

5 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/trsc.2014.0579.

6 Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster Batched Shortest Paths in Road Networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 52–63, 2011.

7 Daniel Delling and Giacomo Nannicini. Core Routing on Dynamic Time-Dependent Road Networks. *Informs Journal on Computing*, 24(2):187–201, 2012.

8 Daniel Delling, Dennis Schieferdecker, and Christian Sommer. Traffic-Aware Routing in Road Networks. In *Proceedings of the 34rd International Conference on Data Engineering*. IEEE Computer Society, 2018. doi:10.1109/ICDE.2018.00172.

9 Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

10 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 21(1):1.5:1–1.5:49, April 2016. doi:10.1145/2886843.

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

12 Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

13 Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, 1979.

14 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

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

16 Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

17 Ben Strasser, Dorothea Wagner, and Tim Zeitz. Space-efficient, Fast and Exact Routing in Time-Dependent Road Networks. *Algorithms*, 14(3), January 2021. URL: `https://www.mdpi.com/1999-4893/14/3/90`.

18 Ben Strasser and Tim Zeitz. A Fast and Tight Heuristic for A* in Road Networks. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.SEA.2021.6`.

A Detailed Performance Profiles by Instance

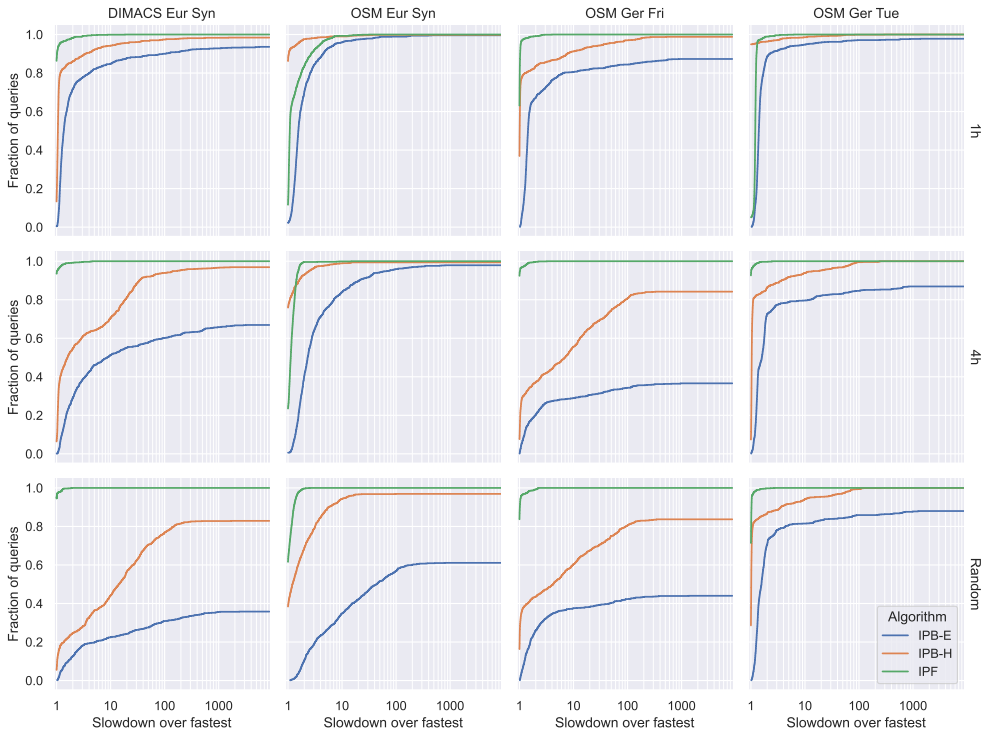

Figure 6 Relative performance profile for the running time of our algorithms on all queries from Table 2 split by graph and query set.

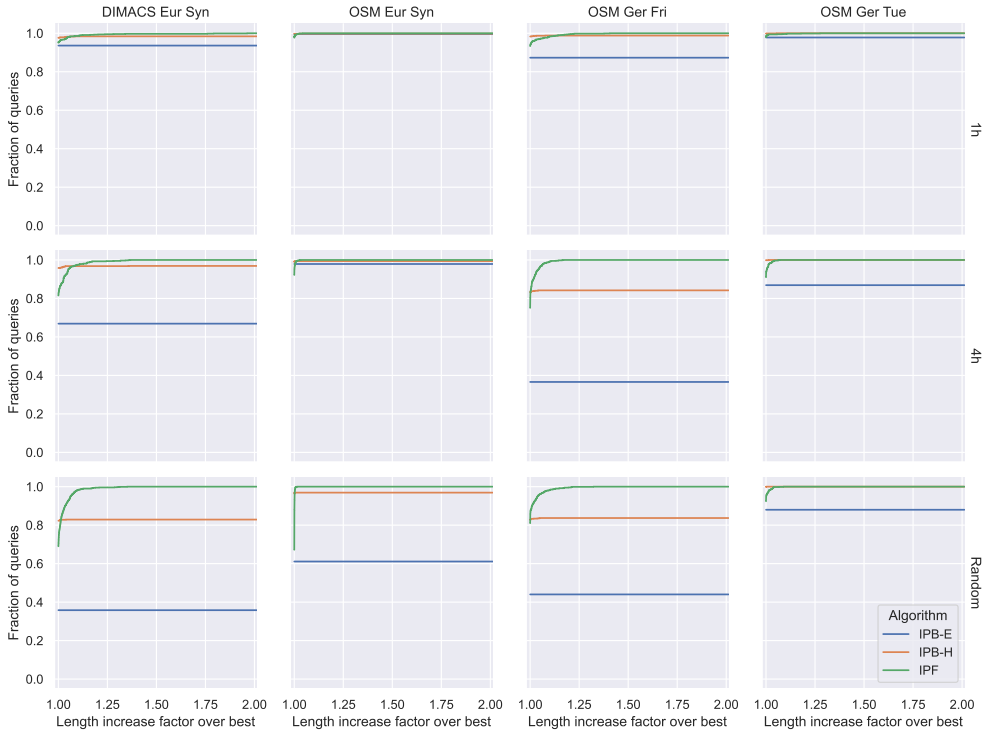

Figure 7 Relative performance profile for solution quality of our algorithms on all queries from Table 2 split by graph and query set.