Robust Mobile Route Planning with Limited Connectivity*

Daniel Delling† Moritz Kobitzsch‡ Dennis Luxen§ Renato F. Werneck¶

Abstract

We study the problem of route planning on mobile devices. There are two current approaches to this problem. One option is to have all the routing data on the device, which can then compute routes by itself. This makes it hard to incorporate traffic updates, leading to suboptimal routes. An alternative approach outsources the route computation to a server, which then sends only the route to the device. The downside is that a user is lost when deviating from the proposed route in an area with limited connectivity. In this work, we present an approach that combines the best of both worlds. The server performs the route computation but, instead of sending only the route to the user, it sends a *corridor* that is robust against deviations. We define these corridors properly and show that their size can be theoretically bounded in road networks. We evaluate their quality experimentally in terms of size and robustness on a continental road network. Finally, we introduce several algorithms to compute corridors efficiently. Our experimental analysis shows that our corridors are small but very robust against deviations, and can be computed quickly on a standard server.

1 Introduction

Map services have motivated extensive research on the fast computation of driving directions in road networks. Finding the quickest route between two points can be modeled as a shortest path problem on weighted graphs. The standard solution to this problem is Dijkstra's algorithm [13]. Although it runs in essentially linear time [18], even on a server it still takes a few seconds on continental road networks (with tens of millions of intersections), which is not fast enough for most applications. Several acceleration techniques find routes much faster with a two-phase approach: an offline preprocessing phase augments the graph with some auxiliary data, which is then used to accelerate online queries. (A recent overview of these techniques is given by Delling et al. [10].) The fastest known technique needs only a few memory accesses to determine the distance between two random points [3], in a fraction of a microsecond.

Most of the techniques have been implemented on server-like machines for the scenario where the computer serves queries for a map service. Mobile devices such as smartphones and tablets are increasingly popular, however, and computing driving directions is one of their key features. There are two basic approaches to achieve this. In the *online* approach, the device holds the map data for rendering only, while the server keeps the auxiliary data for routing. The route is then computed on the server and sent to the device. This setting enables routes to incorporate the most recent available data, such as real-time traffic updates. Moreover, the mobile device must do almost no computation: the route is just a list to be traversed. The main drawback of this approach is that it is not *robust*: if the user makes a wrong turn and deviates from the proposed route, the server must be accessed again. If this happens in an area with no connectivity (or if the route was preloaded to a WiFi-only device), the user is lost. This is remedied by the *offline* approach, where the mobile device keeps all the data internally and computes routes by itself. Although some techniques are fast enough in this scenario [23], the computation effort on the device is higher than for the online approach. The main disadvantage, however, is that data can be outdated and real-time updates cannot be integrated easily since the auxiliary routing data has to be updated.

This work introduces a *hybrid* approach. As in the online version, we still compute the routes on a server, potentially using the most recent traffic data. Instead of submitting only the route itself to the user, however, we submit a *corridor* that is robust against deviations. A straightforward solution to this problem is to send to the user the entire shortest path tree into the destination. There are obvious downsides, however: even modern algorithms [7] take too long to compute the whole tree, and the amount of data sent would be too large. Instead, we want to find a subtree of the shortest path tree that is *small* but as *robust* as

*Partially supported by the BMWi project 'MEREGIOmobil' and DFG Grant SA 933/1

†Microsoft Research Silicon Valley. dadellin@microsoft.com

‡Karlsruhe Institute of Technology. kobitzsch@kit.edu

§Karlsruhe Institute of Technology. luxen@kit.edu

¶Microsoft Research Silicon Valley. renatow@microsoft.com

possible. Sending just the route (we cannot send less) or the entire tree (robust against all deviations) are the extreme cases. We want something in between that offers a good trade-off. There exist a variety of problems that are related to our setup, like the computation of detours [17], k-shortest paths [14, 22], alternatives [2, 6], and replacement paths [22, 24]. However, none of these approaches solves our problem.

Our contributions are as follows. First, we propose different approaches to determine corridors, and carefully evaluate their quality. Second, we show that the sizes of these corridors can be nontrivially bounded on road networks. Third, we show how the corridors can be efficiently computed. Finally, we perform a thorough experimental evaluation of our method, showing that it outperforms straightforward solutions by up to an order of magnitude. It takes us a few milliseconds to find corridors that are very small but extremely robust.

The paper is organized as follows. Section 2 reviews techniques our work is based upon. Section 3 defines our notion of corridors and proves that their size is limited in road networks. Section 4 shows how to compute these corridors. Section 5 presents an empirical evaluation of the proposed corridor in terms of size, robustness, and computational effort. Section 6 concludes our work.

2 Preliminaries

We interpret a road network as a directed graph $G = (V, A, \ell)$ with length function $\ell : A \to \mathbb{N}_0^+$. In order to incorporate turn costs, we use the so-called *edge-based* representation of the road network [5]. Each vertex represents a road segment, and two vertices v and w are connected by an arc (v, w) if it is possible to turn from one to the other through a single intersection. The length of (v, w) represents the travel time from the start of road segment v to the start of road segment w. We denote by P_{st} the shortest s–t path in G, by $|P_{st}|$ its size (number of vertices), and by $\ell(P_{st})$ its length, i.e., $\sum_{(u,v)\in P_{st}} \ell(u, v)$. We also refer to $\ell(P_{st})$ as the *distance* between s and t, denoted by $d(s, t)$. We denote by $T(t)$ the inverse shortest path tree into t.

The literature often considers the *point-to-point shortest path problem*, which asks for the path P_{st} between a source s and a target t. An extension is the *corridor problem*. Here we ask for a superset of the shortest path that is robust against deviations from it. Note that this problem is rather fuzzy because it highly depends on the definition of robustness. In fact, one contribution of our work is how to define and evaluate the robustness of corridors.

Dijkstra. The standard approach for computing shortest path trees in road networks with nonnegative arc lengths is Dijkstra's algorithm [13]. It maintains, for every vertex u, an upper bound $d(u)$ on the length of the shortest path from the source s, as well as the predecessor (*parent*) $p(u)$ of u on the path. These variables are initialized with $d(s) = 0$, $d(u) = \infty$ for all other vertices, and $p(u) = null$ for all u. The algorithm also maintains a priority queue with *unscanned* vertices. At each step, it removes a vertex u from the queue with minimum $d(u)$ value and *scans* it: for each arc $(u, v) \in A$ with $d(u)+\ell(u, v) < d(v)$, it sets $d(v) = d(u)+\ell(u, v)$ and $p(v) = u$. The algorithm terminates when the queue becomes empty. If we are only interested in the shortest path from s to t, one can terminate the algorithm as soon as t is about to be scanned. In road networks, Dijkstra's algorithm runs in essentially linear time [18].

Contraction Hierarchies. As mentioned before, many speedup techniques have been developed for the point-to-point shortest path problem [10]. One of the most prominent is Geisberger et al.'s Contraction Hierarchies (CH) [15]. Like most other techniques, it has an (offline) preprocessing phase that augments the graph by auxiliary data, which is then used to accelerate (online) queries.

The preprocessing phase orders the vertices and *contracts* them in this order. Contracting a vertex u temporarily deletes it from the graph and adds arcs between its neighbors to preserve the distances among them. This is done as follows. For any ordered pair (v, w) of neighbors of u such that $(v, u) \cdot (u, w)$ is the only shortest v–w path in the current graph, we add a *shortcut* (v, w) with $\ell(v, w) = \ell(v, u) + \ell(u, w)$. We call (v, u) the *prefix* of (v, w) and (u, w) its *suffix*. The output of the preprocessing phase is the set A^+ of shortcut arcs, together with the position of each vertex u in the order (denoted by $rank(u)$). The algorithm is correct with any contraction order, but query times and the size of A^+ may vary. In practice, each vertex u is given a priority computed by an online heuristic that measures its importance [15]. In this work, we use a linear combination of its *edge quotient* $E(u)$ and its *level* $L(u)$. The edge quotient of u is given by the number of arcs added divided by the number of arcs deleted if u would be contracted next. The level of u is computed during contraction. Initially, $L(u) = 0$ for all vertices u. When contracting u, we set $L(v) = \max\{L(v), L(u)+1\}$ for all uncontracted neighbors v of u.

The query phase of CH runs a bidirectional version of Dijkstra's algorithm on the graph $G^+ = (V, A \cup A^+)$, but only looking at *upward* arcs, i.e., those leading to neighbors with higher rank. More precisely, let $A^\uparrow = \{(u, v) \in A \cup A^+ : \ rank(u) < rank(v)\}$ and $A^\downarrow = \{(u, v) \in A \cup A^+ : \ rank(u) > rank(v)\}$. The forward search works on $G^\uparrow = (V, A^\uparrow)$, and the reverse search on $G^\downarrow = (V, A^\downarrow)$. Each vertex v maintains (possibly infinite) upper bounds $d_s(v)$ and $d_t(v)$ on its distances from s (found by the forward search) and to t (found by the reverse search). The algorithm keeps track of the vertex u minimizing $\mu = d_s(u) + d_t(u)$, and stops when the minimum value in either priority queue is at least as large as μ.

The shortest path is computed in G^+, so it may contain shortcuts. If we are interested in the corresponding path in G, we need to *unpack* it. Note that a shortcut (v, w) is built from a prefix (v, u) and a suffix (u, w) when contracting the vertex u. Storing this *middle* vertex u with (v, w) allows us to recursively unpack a shortcut into the corresponding sequence of arcs in G.

PHAST. Delling et al. [7] developed an algorithm that uses a contraction hierarchy to accelerate the computation of full shortest path trees. At first sight, it seems one could not do much better than Dijkstra's algorithm, which runs in linear time and is only twice as slow as a plain BFS [18]. Both Dijkstra and BFS, however, have very poor locality. Since they grow a ball of increasing radius around the source, the vertices in their queues are usually spread over different regions of the graph—and in memory, leading to many cache misses. Changing the graph layout in memory can help, but no single layout works well for all possible sources s.

In road networks, PHAST [7] can avoid these problems. Unlike Dijkstra, it works in two phases. Preprocessing is the same as in CH. Given a source s, the query first computes the full shortest path tree as follows. Initially, set $d(s) = 0$ and $d(v) = \infty$ for all other $v \in V$. Then run an upward search from s in $G^\uparrow$(a forward CH search), updating $d(v)$ for all vertices v scanned. Finally, the *scanning phase* of the query processes all vertices in $G^\downarrow$ in reverse rank order. To process v, we check for each incoming arc $(u, v) \in A^\downarrow$ whether $d(u) + \ell(u, v)$ improves $d(v)$. If it does, we update the value. After all updates, $d(v)$ will represent the exact distance from s to v. If we need actual parent pointers in the original graph, we can run an additional sweep over A (the arcs of the original graph) setting $p(u) = v$ if $d(u) + \ell(u, v) = d(v)$ holds.

The main advantage of PHAST over Dijkstra is that only the (cheap) upward CH search depends on the source s. The (more costly) scanning phase visits its vertices and arcs in the same order for *any source*. Permuting vertices appropriately during preprocessing ensures the scanning phase accesses the lists of vertices and arcs sequentially, minimizing cache misses. This alone makes PHAST about 15 times faster than Dijkstra in large road networks. Another advantage of PHAST over Dijkstra's algorithm is that it can easily be parallelized. Implemented on an NVIDIA GTX 580 GPU, the speedup increases to up to 500 compared to a single core of a modern CPU.

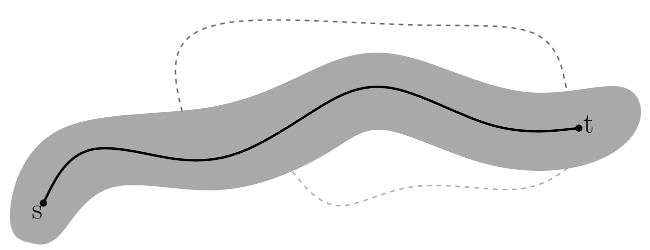
PHAST can be extended to a one-to-many scenario, where one must compute shortest paths from a query vertex s to a (fixed) set $\mathcal{T}$ of vertices. The resulting algorithm, called RPHAST [9], introduces a *target selection* phase between preprocessing and query, which extracts from $G^\downarrow$ the smallest subgraph that is necessary to compute distances to all $t \in \mathcal{T}$.

3 Corridors

As already mentioned in Section 1, a straightforward solution to the corridor problem is to send the entire shortest path tree $T(t)$ to the user. Besides the obvious drawback of transmitting a large amount of data, even with PHAST the computaiton is not fast enough to enable real-time queries. Instead, we are interested in a relevant subtree of $T(t)$ that is as small as possible but also robust against deviations. Our idea is as follows. We choose a set S of *seed* vertices with $S \cap P_{st} = \emptyset$. Then, the s–t corridor $C(s, t)$ is given by the union of P_{st} and all u–t shortest paths with $u \in S$. Note that with this definition, $C(s, t)$ is a subtree of $T(t)$. We now discuss two ways of choosing a seed set.

An obvious choice for S is to include all vertices that are "close" to the shortest path. More precisely, we set $S_\tau = \{u \in V \mid u \notin P_{st} \wedge \exists v \in P_{st} : d(v, u) \leq \tau\}$. Intuitively, S_τ contains all vertices that are within distance τ from the shortest path. We call the resulting corridor the *τ-perimeter corridor* (τ-PC) which is robust against deviations of up to τ from the optimal route. Moreover, we call the problem of finding such corridors the *τ-perimeter corridor problem*. Although this definition looks promising, our experiments (see Section 5) indicate that we can do better. The main drawback is that we add too many vertices in urban areas and not enough important roads.

For this reason, we propose (and from here on use) the notion of *turn corridor* (TC). Here, S contains all vertices adjacent to P_{st}; more formally $S = \{u \in V \mid u \notin P_{st} \wedge \exists v \in P_{st} : (v, u) \in A)\}$. This makes the corridor robust against exactly one deviation from the shortest path. We call a vertex $u \in S$ a *deviation vertex*. We can easily extend this approach to recursively construct a k-turn s–t corridor $TC_k(s, t)$ that is robust against k wrong turns. Therefore, we determine the *k-th order* deviation vertices $D_k = \{u \in V \mid u \notin TC_{k-1}(s, t) \wedge \exists v \in TC_{k-1}(s, t) : (v, u) \in A)\}$, i.e., all vertices adjacent to $TC_{k-1}(s, t)$, with $TC_0(s, t) = P_{st}$. Then $TC_k(s, t)$ is seeded by $S_k = \bigcup_{i=1}^{k} D_i$. We call the corresponding problem of finding such corridors the *k-turn corridor problem*. Note that if shortest paths are unique, turn corridors are unique as well. Figure 1 visualizes the differences between PC and TC.



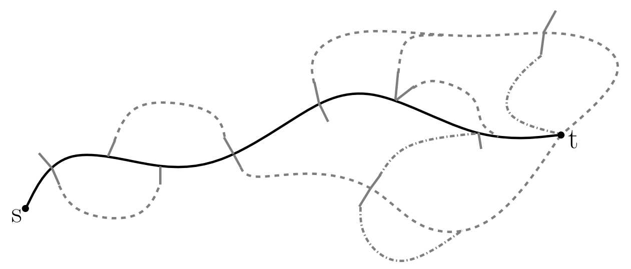


Figure 1: Schematics for perimeter (left) and turn (right) corridors.

Theoretical analysis. Given the definition of turn corridors, we now show that their size is limited in graphs with low *highway dimension* [1, 4]. The concept of highway dimension has been introduced to explain the good performance of hierarchical point-to-point speedup techniques on road networks (which are believed to have small highway dimension). Since it is a theoretical model, it makes some simplifying assumptions such as the graph being undirected.

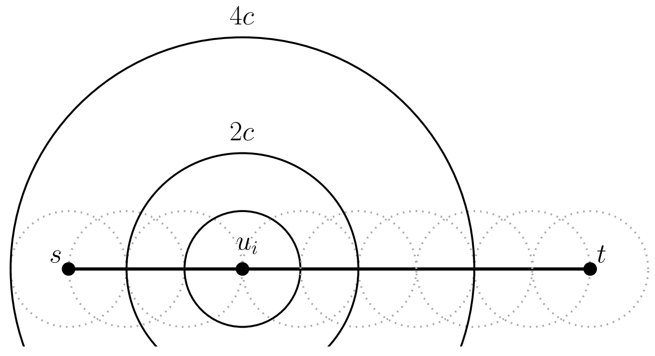
We can use highway dimension to show that, under reasonable assumptions, a 1-turn corridor has at most $\mathcal{O}(h \cdot n^{0.75})$ vertices, where h is the highway dimension of the graph. To explain highway dimension in more detail, we need the concept of a *shortest path cover* (SPC). An (r,k)-SPC S is a set of vertices with two properties. First, it hits all shortest paths of length between r and $2r$. Second, the set is *sparse*: for any vertex $u \in V$, the ball $B_{2r}(u)$ (containing all vertices v with $d(u,v) < 2r$) contains at most k vertices from S. The highway dimension of a graph is the minimum h such that an (r,h)-SPC exists for all r.

Following Abraham et al. [4], we also consider the graph to be undirected. For simplicity, we assume the maximum degree of G is constant, shortest paths are unique, and edge weights are bounded by a constant, i.e., $\ell(a) \in \Theta(1)$. The last assumption implies that $\ell(P_{st}) \in \mathcal{O}(|P_{st}|)$, and can be enforced by splitting long edges into multiple ones. Finally, we assume that $|P_{st}| \in \mathcal{O}(\sqrt{n})$, which is reasonable in road networks (cf. Section 5).

Theorem 3.1. (Turn Corridor Size) *Let G be an undirected weighted graph with constant maximum degree, highway dimension h, and length function $\ell : A \to \mathbb{N}_0^+$, where $\forall a \in A : \ell(a) \in \Theta(1)$. Moreover, $\ell(P_{st}) \in \mathcal{O}(\sqrt{n})$ for all $s,t \in V$. Then, $|TC_1(s,t)| \in \mathcal{O}(h \cdot n^{0.75})$ holds.*

Proof. Let $c := \Theta(\sqrt[4]{n})$. Then $\ell(P_{st}) \in \mathcal{O}(c^2)$ holds for all shortest paths. We consider two cases:

Case 1 ($\ell(P_{st}) \in \omega(c)$): We partition P_{st} into $\mathcal{O}(c)$ parts, each of length at most c. In each subpath P^i, consider a vertex u_i and the balls of size $2c$ and $4c$ around it. See Figure 2 for an illustration. Since the length of any edge is bounded by a constant, any deviation vertex v is contained in $B_{2c}(u_i)$. Due to the subpath optimality, the shortest path from v to the target starts with a shortest path of length $2c$ that is fully contained in $B_{4c}(u_i)$. Moreover,



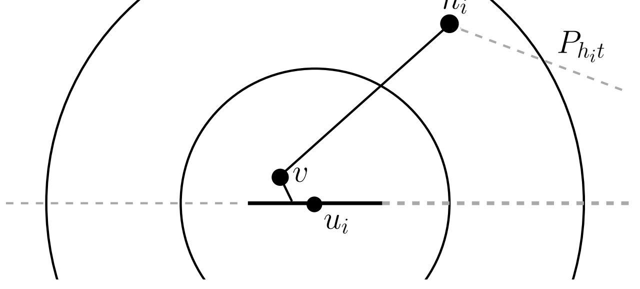


Figure 2: Illustration of the proof of Theorem 3.1. **Left:** We split P_{st} into $\mathcal{O}(c)$ parts and consider the balls of size $2c$ and $4c$ around a vertex u_i on each subpath P^i. **Right:** The deviation vertex v is contained in the ball of size $2c$. The shortest path from v to t starts with a shortest path of length $2c$ that is covered by h_i.

because the graph has highway dimension h, all shortest paths of length $2c$ in that ball are covered by at most h vertices. Since there exist $\mathcal{O}(c)$ deviation vertices of P^i, the sum of the lengths of all shortest paths from the deviation vertices to the h covering vertices is bounded by $\mathcal{O}(c^2)$. From each of these h vertices, the length of the shortest path to t is bounded by $\mathcal{O}(c^2)$. Since we have $\mathcal{O}(c)$ subpaths, the total size of the corridor is bounded by $\mathcal{O}(h \cdot c^3) = \mathcal{O}(h \cdot n^{0.75})$ edges.

Case 2 ($\ell(P_{st}) \in \mathcal{O}(c)$): P_{st} has $\mathcal{O}(c)$ deviation vertices v, and $\ell(P_{vt}) \in \mathcal{O}(c)$ holds for all of them. This directly gives us a bound of $\mathcal{O}(c^2)$ on the size of $TC_1(s,t)$.

Note that these bounds are very conservative: they do not use the fact that the shortest paths from the covering vertices to the target share many edges. Indeed, experiments suggest that corridors are much smaller: their size depends linearly on the size of the shortest path.

4 Computation of Corridors

In this section, we discuss how we can solve the k-turn corridor problem efficiently. We first show how existing techniques can be applied and then turn to a tailored solution. All techniques follow the same pattern: we perform $k+1$ rounds, with each round extending the corridor (denoted by C) by one turn. Therefore, in each round i, we first determine the deviation vertices $D = \{u \in V \mid u \notin C \wedge \exists (v,u) : v \in C\}$ of C. Then we add for each $u \in D$ the u–t shortest path to C. In round 1, we set $D = \{s\}$. With this general approach, $C = TC_{i-1}(s,t)$ after round i. Recall that $TC_0(s,t) = P_{st}$.

4.1 Straightforward Approaches. The most natural approach to compute $TC_k(s,t)$ is to first construct $T(t)$ with Dijkstra's algorithm. Then, we maintain two stacks S_0 and S_1, where S_0 is initialized with s. In each round, we process the vertices of S_0. When removing the top vertex u from S_0, we add u to C and scan all its outgoing edges $(u,v) \in A$ with $v \notin C$. If v is the parent of u in $T(t)$, we add v to S_0, otherwise to S_1. The round ends by setting $S_0 = S_1$ and $S_1 = \emptyset$. Note that at the end of round i, S_0 contains all the deviation vertices from C, and $C = TC_{i-1}(s,t)$. We call this approach *corridor Dijkstra* (cDijkstra).

The main drawback of cDijkstra is the prohibitive running time to build the shortest path tree. This can be partly remedied by replacing Dijkstra with PHAST; we call the resulting algorithm *cPHAST*. Still, even on multiple cores PHAST cannot compute trees fast enough on continental road networks using the edge-based representation. The GPU version of PHAST can, but it has other issues. First, not all servers nowadays are equipped with GPUs. Second, all remaining computations would have to be performed on the GPU as well, since the GPU–CPU communication is expensive. In fact, even copying just the corridor from the GPU can already be costly.

Instead of computing full shortest path trees, we could use a point-to-point speedup technique such as CH (other techniques could be used as well). In each round, we first compute the deviation vertices D of C by traversing C. Then, we run for each $u \in D$ a CH query and add the unpacked shortest path to C. We call this approach *corridor CH* (cCH).

The major drawback of this approach is that it does a lot of unnecessary computations—in particular, it performs the upward search from t multiple times. We can remedy this by adapting the bucket-based approach of Knopp et al. [20] to our scenario. We run the upward search from t only once and store its search space by keeping the distance to t for each vertex we scan. Then, to compute a new shortest path from a deviation vertex u, we only need to run the upward search from u. When we scan a vertex v that has also been scanned from t, we check whether $d(u,v) + d(v,t) < d(u,t)$ holds and update $d(u,t)$ accordingly. Of course, we still have to unpack all shortcuts that appear on shortest paths. This *corridor bucket CH* (cBCH) approach saves roughly 30% of the work of cCH.

4.2 Our Approach. Analyzing cBCH, one may notice that the algorithm still does a lot of unnecessary computations. For example, the upward searches from neighboring deviation vertices most probably overlap a lot and we unpack shortcuts multiple times. Indeed, as Section 5 will show, all above mentioned approaches fall short in terms of performance. We now present our *tailored corridor computation* algorithm (TCC), which avoids unnecessary computations as much as possible. It uses a contraction hierarchy and borrows some ideas from PHAST and RPHAST.

Like cBCH, TCC runs an upward search from t in $G^{\downarrow}$ during initialization, storing the search space. TCC still works in rounds, but each now consists of three phases. First, the *upward* phase, borrowed from RPHAST [9], determines all vertices D' that are reachable from D in $G^{\uparrow}$. Then, the *sweep* phase computes the distances to all vertices in D'. At the end of the sweep phase, we have computed the shortest paths from each vertex in D to t in G^{+}. Finally, the *unpacking* phase then extracts all these packed paths and adds them to C. Since we again have computed all shortest paths from all deviation vertices to t, $C = TC_{i-1}(s,t)$ after round i. In the following, we detail each phase in turn.

We keep a global marker, called *final*, that identifies vertices with correct distance value to t. During initialization, we can only ensure correct distance values for all vertices on the contracted path from t to the highest ranked vertex. Hence, we mark only them as final.

The upward phase identifies all vertices D' that are reachable from D in $G^{\uparrow}$. Recall that D is the set of deviation vertices of the current corridor C. Therefore, we first traverse the current corridor C to identify all deviation vertices. In the first round, $D = \{s\}$. Then, we set $D' = D$ and maintain a queue Q initialized by D. Whenever we remove a vertex u from Q, we first check whether u is marked as final. If it is, we discard u; otherwise, we scan all upward edges $(u,v) \in A^{\uparrow}$ and add each $v \notin D'$ to both Q and D'.

In the sweep phase we process D' in a top-down manner, as in PHAST. For each $u \in D'$, we scan all $(u,v) \in A^{\uparrow}$ and check whether $\ell(u,v) + d(v) < d(u)$; if so, we update $d(u)$ and $p(u)$. Due to the correctness of PHAST, this process computes the shortest paths from all deviation vertices to t in G^{+}.

The unpacking phase now expands all these packed shortest paths. We start by adding all vertices in D to a queue Q. Whenever we extract a vertex u from Q, we check whether $u \in C$. If it is, we discard u; otherwise, we add u to C, mark it as final, and identify its parent v. We recursively unpack the shortcut (v,u), setting the distance values and parent pointers of the middle vertices accordingly. Moreover, we mark all middle vertices as final and add them to C. Note that we do not need to unpack the suffix of a shortcut if its middle vertex w is already contained in C because $w \in C$ implies that P_{wt} is already fully contained in C. We need, however, continue to unpack the prefix because P_{uw} does not need to be part of C. Note that, with this unpacking routine, each shortcut contributing to C is unpacked exactly once.

Optimizations. We can apply some optimization techniques to accelerate TCC. First, we can reorder vertices by their level in the contraction hierarchy to improve locality. We also exploit the fact that the number of levels is small in road networks by keeping a bucket per level. While determining the vertices (which we need to process) that are reachable from the deviation vertices, we add each reachable vertex to the bucket associated with its level. Then, we can process the levels in decreasing order, scanning each level in a linear fashion. This accelerates the algorithm since it increases locality.

5 Experiments

We implemented all algorithms from Section 4 in C++ and compiled them with GCC 4.4.3, using the optimization flags `-O3` and `-mtune=native`. We use a binary heap as priority queue. The experiments were conducted on an Intel Core-i7 920 (4 cores, each clocked with 2.67 GHz) with hyper-threading activated (2 threads per core) and 12 GB of DDR3-1066 RAM running SuSE Linux 11.3. As input, we use the road network of Western Europe, made publicly available by PTV AG [21] for the 9th DIMACS implementation challenge [12]. The published graph is *node-based*, however, with each intersection modeled by a vertex and each road segment by a directed arc. This model is not very realistic: it does not incorporate turn costs, which makes routing artificially easier [8, 16]. We therefore expand the network to use the *edge-based* representation (see Section 2), which leads to a graph that has 42.5 million vertices and 95.5 million directed arcs. Since we do not have access to publicly available turn costs, we follow the approach of Delling et al. [8] and set U-turn costs to 100 seconds; the remaining turns are free.

All CH-based algorithms we tested share the same preprocessed data. We implemented the CH preprocessing following the implementation of Kieritz et al. [19], using $4 \cdot E(u) + 2 \cdot L(u)$ as priority term (cf. Section 2). With these parameters, computing the contraction hierarchy takes 100 minutes on one core and results in 102 (132) million upward (downward) arcs.

5.1 Quality. We first evaluate the quality of our corridors for various parameters. We test two properties: size and robustness against deviations. While the first one is easy to measure, we need to come up with a measure for robustness. For this, we consider two *driver profiles* that model drivers deviating from the route from time to time. The first is the *p deviation driver* DD(p), which leaves the optimal route at every intersection with probability p. The second profile is the *(p,p')-nervous deviation driver* NDD(p,p'). Here, a driver also deviates from the optimal route with probability p; after a wrong turn, however, the probability of making another wrong turn increases to $p' > p$ (because the driver gets nervous). Once the driver gets back on track, the deviation probability goes back to p.

For these driver profiles, we evaluate how often a user following these patterns reaches the target without leaving the corridors we computed. We call this the *success rate* of a drive. Table 1 reports the size and the average success rates over 100 drives for two profiles, DD(5%) and NDD(5, 10%). We average the numbers over 1 000 corridors, each corresponding to a random query.

Table 1: Quality of different corridors. Column *size* gives the average number of vertices in the corridor, and *success rate* reports the percentage of successful drives by (5%)-deviation and (5%, 10%)-nervous deviation drivers.

turn corridor				perimeter corridor			
	size	success rate [%]		τ	size	success rate [%]	
k	$\|V\|$	DD(5%)	NDD(5, 10%)	[s]	$\|V\|$	DD(5%)	NDD(5, 10%)
0	1 351	0.0	0.0	0	1 351	0.0	0.0
1	4 835	10.3	7.1	10	6 223	7.0	4.4
2	12 204	73.2	64.7	20	7 689	8.7	5.5
3	25 892	96.8	94.8	30	9 570	10.6	6.9
4	50 271	99.7	99.4	50	14 284	15.0	10.2
5	88 742	100.0	99.9	100	31 547	28.2	21.7
6	148 370	100.0	100.0	300	209 513	79.6	77.2

We observe that turn corridors dominate perimeter corridors. Not only are turn corridors smaller, but they also have better success rates. One reason is that turn corridors are favored by our driver profiles, but they also adapt better to different scales. In fact, for the profiles we consider almost all drives are successful for k as low as 3. At this point, the corridor is only 19 times bigger than the route itself. Also note that one could use succinct data structures [11] to obtain an extremely compact representation of the corridors: since the corridor is a tree, one needs only a few bits per vertex. More concretely, for $k = 3$ we would need to send less than 10 kB to the user.

We also note that perimeter corridors stay small up to deviations of 100 seconds (combined with low success rates though). Increasing the deviation to 300 seconds increases the size of the corridors by a factor of 7 compared to 100 seconds, making them even bigger than 6-turn corridors. A reason for this is that we use U-turn costs of 100 seconds; only after increasing the deviation beyond this point can the corridors handle U-turns. This explains the low success rate of perimeter corridors, but only partially: even with 300 seconds the success rate is still well below that of much smaller turn corridors.

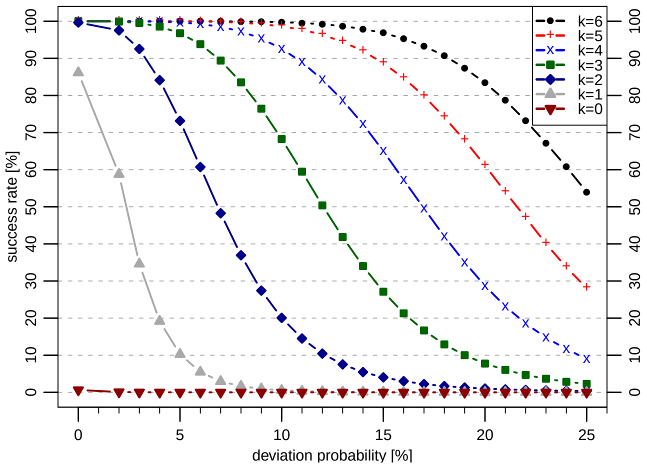


Figure 3: Success rate of different k turn corridors for varying p deviation driver profiles.

We also evaluate the impact of the deviation probability p (for the DD(p) profile) on the success rate of our turn corridors, Figure 3 shows the results. As expected, the higher k, the more robust the corridors are. For $k = 6$, we achieve a success rate of nearly 100% up to $p = 10\%$. This means that the corridor is robust against drivers that on average deviate from the route at every tenth chance. If we increase p to 25%, still less than half the drives leave the corridor. We also observe that for $k = 3$, the success rate drops below 70% for $p \geq 10\%$. Hence, this experiments reveals that increasing k above 3 does yield advantages. In particular, for WiFi-only devices it may pay off to send bigger corridors to the user.

We stress that these success rates are achieved *without* updating the corridor after a deviation. Of course, we can further increase the success rate by allowing updates to the corridor as soon as the user enters an area with connectivity again. Our experiments show that we only have to send about 80 vertices (on average) to update a corridor for $k = 1$ after a deviation. This is 60 times less than resending the whole corridor. For $k = 4$, we have to send 2 000 vertices after a single deviation, still only a small fraction of the full corridor and only twice the size of a shortest path.

Local queries. Up to now, we only evaluated the quality of corridors for random queries, which are mostly long-range. Since most queries are not cross-continental in practical applications, we now evaluate corridors depending on the Dijkstra rank of the query. (The Dijkstra rank of a vertex u with respect to s is i if u is the i-th vertex taken from the priority queue when running Dijkstra's algorithm from s.) Figure 4

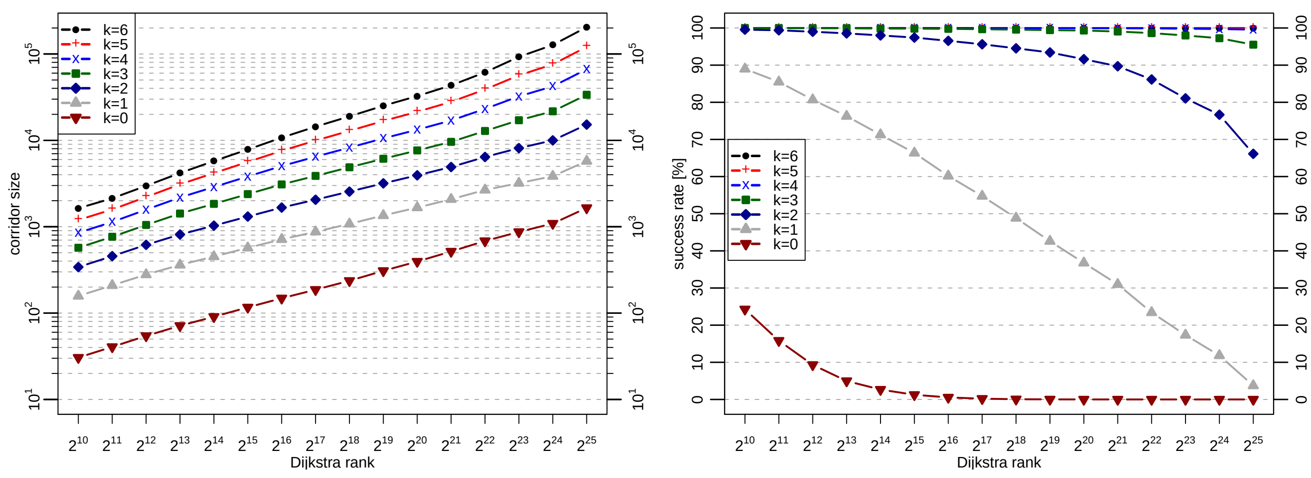


Figure 4: Corridor size (left) and success rate of a DD(5%) (right) depending on the Dijkstra rank of a query.

shows the results for Dijkstra ranks between 2^{10} and 2^{25}. As expected, the corridor sizes increases with the Dijkstra rank. In road networks, we can assume that if a vertex has Dijkstra rank i, the shortest path from s to v contains $\mathcal{O}(\sqrt{i})$ vertices. This is reflected by $k = 0$. Moreover, we observe that the size blowup $(|TC_k(s,t)|/|P_{st}|)$ is almost independent of the rank of the query: it is only slightly bigger for short-range queries. The reason for this is that short-range queries often do not touch the highways and the number of deviation vertices is higher in urban areas than on highways.

We also observe that the success rate of DD(5%) highly depends on the rank of the query: the lower the rank, the higher the success rates. Since most real-world queries are short- and mid-range, we expect our corridors to perform even better than reported in Table 1.

5.2 Performance. After evaluating the quality of corridors, we now check how fast we can compute them with the algorithms we presented in Section 4. Our input again is Europe, and we evaluate the (sequential) running times for 1000 queries with s and t chosen at random. In this scenario, running cDijkstra takes roughly 14 seconds per query, independent of k. Table 2 reports the results for the other algorithms we consider. Note that we could build parallel versions of cCH, cBCH, cPHAST, and TCC that would scale very well with the number of cores used.

Table 2: Sequential running times of different algorithms for computing turn corridors. All running times are given in milliseconds.

k	cCH	cBCH	cPHAST	TCC
0	0.33	0.34	968.42	0.73
1	7.35	5.26	969.32	5.67
2	44.45	30.96	970.36	9.73
3	156.95	100.81	973.36	16.26
4	382.51	263.00	974.14	27.34
5	795.26	545.97	977.30	42.54
6	1643.34	1132.72	982.83	68.66

We observe that cPHAST is about 14 times faster than cDijkstra. This is expected, since most of the work is spent on constructing the shortest path tree, and confirms the speedup reported by Delling et al. [7]. This is still not fast enough, however, and would not enable real-time queries even with a parallel implementation.

Both cCH and cBCH outperform cPHAST for $k \leq 5$, but they are slower for $k = 6$. It is easy to see why: the number of CH searches run by cCH and cBCH increases with the number of deviation vertices, which grows with k. In contrast, cPHAST computes a (costly) shortest path tree only once, and can add deviation vertices almost for free.

Comparing cCH and cBCH with TCC, we observe that for $k \leq 1$ all three algorithms have very similar performance. For $k > 1$, however, TCC outperforms any other algorithm. More importantly, query times remain below 28 ms for $k \leq 4$, which is still fast enough for interactive applications. (It is comparable to typical network latencies.) For $k = 4$, TCC is more than an order of magnitude faster than any other algorithm. As noted in Section 5.1, increasing k further is of limited use. Summarizing, TCC is the best choice to compute corridors.

Local queries. As in the quality experiments, we now turn to local queries. Figure 5 reports the running times of cCH, cBCH, and TCC when varying k between 1 and 6 and Dijkstra ranks between 2^{10} and 2^{25}. We

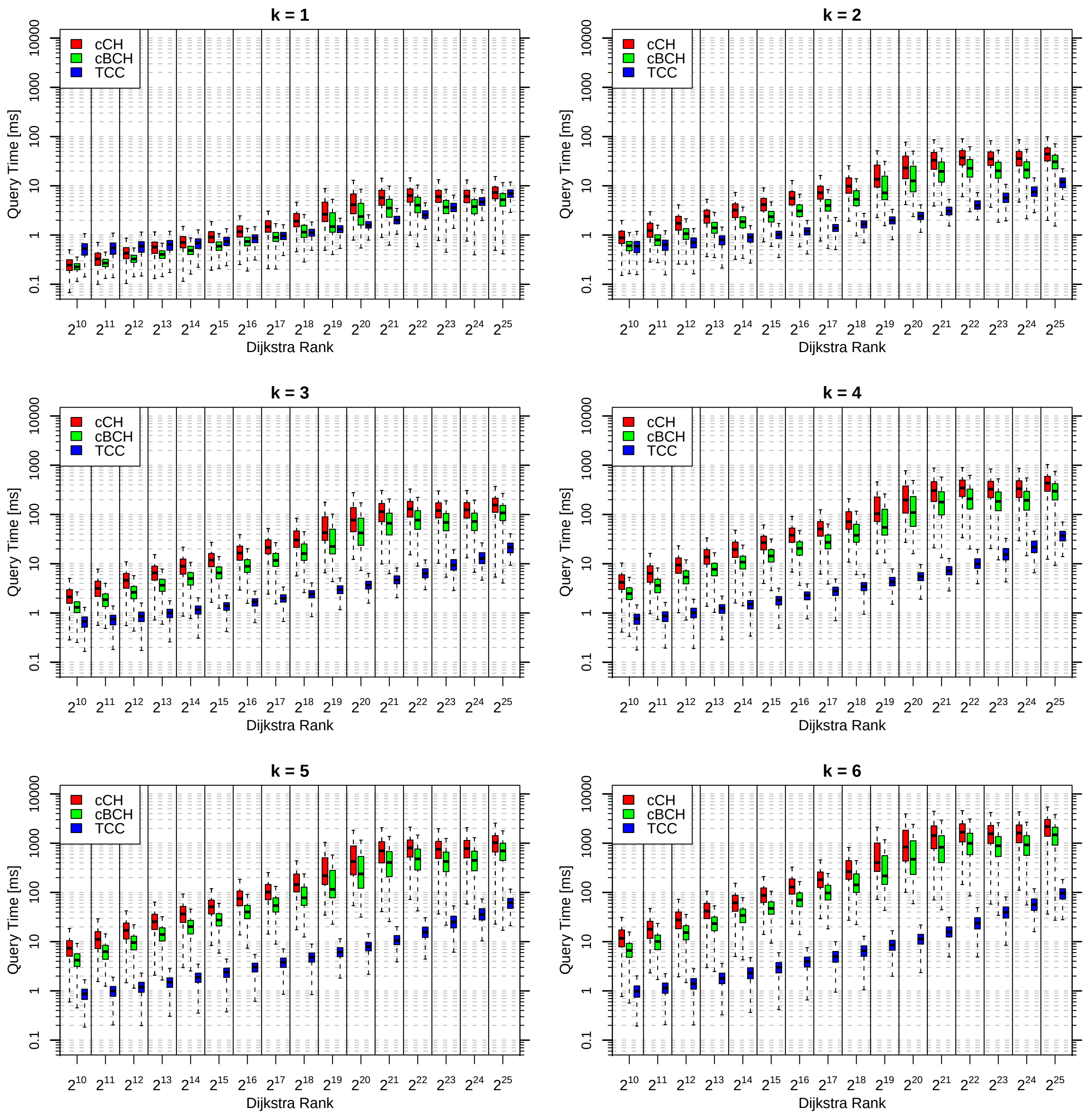


Figure 5: Performance of TCC, cCH, and cBCH when computing k-TCs with k between 1 and 6 for varying Dijkstra ranks.

observe that TCC outperforms cCH and cBCH on almost all scales. Only for $k = 1$ and low Dijkstra ranks are cCH and cBCH slightly faster than TCC. Moreover, cBCH outperforms TCC slightly for $k = 1$ and high Dijkstra ranks as well. However, for our most interesting scenario, $k = 4$, TCC always is the fastest algorithm.

6 Conclusion

In this paper, we introduced the concept of shortest path corridors. Motivated by driving directions for mobile devices, we argued that existing approaches to mobile route planning have disadvantages, such as using outdated information or an inability to update directions when the user deviates from the optimal route. Shortest path corridors achieve the best of both worlds: the can use the most recent data and are robust against deviations. Its key idea is to identify a good subtree of the shortest path tree into the target. We have shown that the corridors we proposed are small and robust. We also presented several algorithms to compute them, including a tailored one, called TCC. On a continental-sized road network, TCC yields the best computation times, sometimes by more than an order of magnitude compared to straightforward approaches. This speedup makes the difference between queries being interactive (real-time) or not.

Acknowledgements. We would like to thank Dennis Schieferdecker for vivid discussions and helpful comments. We also thank Iqbal Mohomed for bringing up the problem in the first place.

References

[1] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. VC-Dimension and Shortest Path Algorithms. In *ICALP*, LNCS 6755, pp. 690–699. Springer, 2011.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative Routes in Road Networks. In *SEA*, LNCS 6049, pp. 23–34. Springer, 2010.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *SEA*, LCNS 6630, pp. 230–241. Springer, 2011.

[4] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA*, pp. 782–793. SIAM, 2010.

[5] T. Caldwell. On Finding Minimum Routes in a Network With Turn Penalties. *Communications of the ACM*, 4(2), 1961.

[6] Y. Chen, M. G. H. Bell, and K. Bogenberger. Reliable Pretrip Multipath Planning and Dynamic Adaptation for a Centralized Road Navigation System. *IEEE Transactions on Intelligent Transportation Systems*, 8(1):14–20, March 2007.

[7] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. In *IPDPS*, pp. 921–931. IEEE, 2011.

[8] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *SEA*, LNCS 6630, pp. 376–387. Springer, 2011.

[9] D. Delling, A. V. Goldberg, and R. F. Werneck. Faster Batched Shortest Paths in Road Networks. In *ATMOS*, OASIcs 20, pp. 52–63, 2011.

[10] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. Zweig, editors, *Algorithmics of Large and Complex Networks*, LNCS 5515, pp. 117–139. Springer, 2009.

[11] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS Succinct Tree Representation. In *WEA*, LNCS 4007, pp. 134–145. Springer, 2006.

[12] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74. American Mathematical Society, 2009.

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

[14] D. Eppstein. Finding the k shortest paths. In *FOCS*, pp. 154–165, 1994.

[15] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*, LNCS 5038, pp. 319–333. Springer, 2008.

[16] R. Geisberger and C. Vetter. Efficient Routing in Road Networks with Turn Costs. In *SEA*, LNCS 6630, pp. 100–111. Springer, 2011.

[17] L. Gewali and R. Koganti. Detour Admitting Short Paths. In *ITNG*, pp. 970 –975, 2011.

[18] A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM Journal on Computing*, 37:1637–1655, 2008.

[19] T. Kieritz, D. Luxen, P. Sanders, and C. Vetter. Distributed Time-Dependent Contraction Hierarchies. In *SEA*, LNCS 6049, pp. 83–93. Springer, 2010.

[20] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *ALENEX*, pp. 36–45. SIAM, 2007.

[21] PTV AG - Planung Transport Verkehr. `http://www.ptv.de`, 1979.

[22] L. Roditty and U. Zwick. Replacement Paths and k Simple Shortest Paths in Unweighted Directed Graphs. In *ICALP*, LNCS 3580, pp. 249–260. Springer, 2005.

[23] P. Sanders, D. Schultes, and C. Vetter. Mobile Route Planning. In *ESA*, LNCS 5193, pp. 732–743. Springer, 2008.

[24] V. V. Williams. Faster Replacement Paths. In *SODA*, pp. 1337–1346. SIAM, 2011.