Traffic-Aware Routing in Road Networks

Daniel Delling, Dennis Schieferdecker, Christian Sommer

{ddelling,dschieferdecker,csommer}@apple.com

Abstract

We study how to compute routes that avoid traffic in road networks. Imperfections in real-time traffic feeds may yield routes with undesirable detours through parking lots or residential areas. The main challenge we address in this work is that of defining and computing paths that incorporate a volatile secondary cost function. We define the problem, study its complexity, and present algorithms that compute routes without undesirable detours. Experiments on continental-sized road networks demonstrate the feasibility of our approach.

1 Introduction

With turn-by-turn navigation as a standard feature of contemporary smart phones, driving navigation systems have become widely used. Since most smart phones are permanently connected to the internet, many users have access to real-time traffic information. As a result, users expect their navigation system to be smart, providing time-saving detours that *beat traffic*. However, computing the quickest (time-dependent) route according to traffic is harder than one might think. Traffic information is ever changing and can be incomplete or inaccurate, which requires very good traffic prediction models. But even with perfect traffic information at hand, some detours are forbidden, such as cutting through a gas station, a freeway ramp, or a parking lot. Also, sending many users onto residential streets may produce highly undesirable traffic jams. One solution is to penalize certain roads, yielding a time-dependent problem that does not optimize pure travel time, which is known to be NP-hard [1]. In this work, we study finding traffic-aware routes that do not contain *undesired* local detours. Our main approach is to compute paths with respect to two different length functions, one that is traffic-aware, and one that is traffic-independent. We define the problem mathematically, study its complexity, and present two approaches that find such paths.

There is a vast number of papers on route planning algorithms [2]. Most of these algorithms follow a two-stage approach: first preprocess the graph to compute auxiliary data, which is then used to accelerate shortest-path queries. With query times as low as a few memory accesses for the static case [3], some research has focussed on reducing preprocessing times [4, 5, 6, 7]. One of the main motivations for that line of work was to handle traffic updates quickly. However, all of these studies compute shortest paths on the updated graph, which may have undesired detours as mentioned above. Another line of work incorporates recurring traffic patterns (but not accidents and their effects) as a time-dependent length function [8, 9, 10], or as a probabilistic length function [11, 12], with similar practical challenges. In the following, we argue that *good* routes in traffic are *to some extent* optimal with respect to two different length functions. As such, our approach is related to multi-criteria search, which tries to find all Pareto-optimal solutions for multiple length functions [13, 14, 15, 16, 17]. However, according to our definition, the best path may not be Pareto-optimal. Finally, there has been work on shortest paths with restrictions [18] to efficiently compute routes avoiding certain types or roads.

This work is organized as follows. We introduce basic definitions and review some work relevant for this study in Section 2. We propose a new variant of the shortest-path problem, study its complexity, describe algorithms to solve it, and explain how it relates to routing in traffic in Section 3. We also describe a new

approach to routing in dynamic networks in Section 4. We close with experimental results in Section 5 and open questions in Section 6.

2 Definitions and Background

Let $G = (V, A)$ be a directed graph, and let $\ell : A \to \mathbb{N}$ be a *length function* of the graph. We set $n := |V|$ and $m := |A|$. We denote a path by $P = (v_0, \ldots, v_k)$, and by $\ell(P) = \sum_{(u,v)\in P} \ell(u, v)$ its length for a given length function ℓ. We call a length function incorporating traffic information *traffic-aware*. Given two nodes s and t, we denote by $\mathsf{OPT}_\ell(s, t)$ the shortest path between s and t for the length function ℓ, and by $dist_\ell(s, t)$ the length of that shortest path. By the shortest ℓ path we denote the shortest path according to the length function ℓ. We denote by $P_{ij} = (v_i, \ldots, v_j), 0 \le i < j \le k$ a *subpath* of P. The *uniformly bounded stretch* $\mathsf{UBS}_\ell(P)$ of a path P is defined as $\max_{P_{ij}} \ell(P_{ij})/dist_\ell(v_i, v_j)$, i.e., the maximum ratio of subpath length and the shortest-path distance between the end points of that subpath. The *local optimality* of a path P is the maximum length r such that all its subpaths $\ell(P_{ij}) \le r$ are shortest ℓ paths. The maximum length r is also called the *local optimality radius*. Note that for a path P that is locally optimal with radius r there may be nodes $v_j, v_j \in P$ with $\ell(P_{ij}) > r$ but $dist_\ell(v_i, v_j) < r$, which is arguably the main difference between UBS and local optimality [19].

Dijkstra's algorithm [20] computes $dist_\ell(s, t)$ by scanning nodes in increasing distance from s until it scans t. When scanning a node u, its outgoing arcs (u, v) are scanned and their heads are *evaluated*, i.e., it is checked whether the path from s to v via u is the shortest path from s to u seen so far, in which case v is inserted into a priority queue. We can reconstruct the path by storing a *parent* for each node.

The Customizable Route Planning (CRP) method [4] accelerates queries in road networks by first partitioning the input graph into cells. Then, in a quick customization step, all distances between the *boundary nodes* (nodes with at least one neighbouring node in a different cell) of a cell are computed. When performing a query from s to t, only the cells containing s and t must be explored by a Dijkstra-like exploration, whereas all other cells can be skipped using the precomputed distances between boundary nodes. The algorithm gains additional performance by exploiting small separators [21, 22], multi-level partitioning, and parallelism and cache-locality for the customization step. By running a many-to-many query [23], CRP can compute the UBS of a path in reasonable time.

3 The Shortest Smooth Path Problem (SSPP)

Motivated by computing best routes in traffic, we introduce the point-to-point *shortest ϵ-smooth path problem*: Given two nodes s and t, a graph G, two length functions ℓ and $\tilde{\ell}$, and a parameter $\epsilon > 0$, the *shortest ϵ-smooth path* is defined as the shortest path P between s and t with respect to $\tilde{\ell}$ such that $\mathsf{UBS}_\ell(P) < 1+\epsilon$. In other words, P is the shortest path according to a volatile length function $\tilde{\ell}$, avoiding long detours according to a smooth length function ℓ. When computing routes in traffic, $\tilde{\ell}$ could represent the minimum travel time in traffic, which can be a rather noisy length function, whereas ℓ could ignore traffic and may incorporate other factors like penalties for undesired roads.

Note that the shortest smooth path is not a Pareto solution when optimizing both length functions by multi-criteria search. The main reason why we believe multi-criteria search is not a feasible option to compute routes in traffic is that Pareto optimality is with respect to the entire path on a global scale. When routing with traffic, however, we wish to avoid local detours.

3.1 Complexity

Finding shortest smooth paths is related to the NP-hard problem of finding paths that are approximately optimal with respect to two different length functions. Mihalák, Srámek, and Widmayer [24], motivated by a novel *robust* optimization framework, prove results on shortest paths with respect to multiple length functions. Their theorem is stated as a counting result, but the proof is actually a reduction to the decision version, which we paraphrase here.

Theorem 1 (Mihalák, Srámek, and Widmayer [24]) *Let $G = (V, A)$ be a graph with $n := |V|$ nodes, let $s, t \in V$ be two nodes, let $\ell_1, \ell_2 : A \to \mathbb{R}^+$ be two length functions, and let $L_1, L_2 \in \mathbb{R}^+$ be two length thresholds. It is* NP*-hard to decide whether there exists a path P from s to t that is shorter than L_1 with respect to ℓ_1 and shorter than L_2 with respect to ℓ_2, i.e., $\ell_1(P) < L_1$ and $\ell_2(P) < L_2$.*

The hardness proof is by reduction from KNAPSACK with $n-1$ items. ℓ_1 represents weights w_i, and ℓ_2 represents values v_i. In the graph there are two parallel arcs between node $i-1$ and node i, representing the choice between including item i or not. See [24] for details.

The problem of deciding whether there is a path that is approximately shortest with respect to two different length functions is closely related to finding shortest smooth paths but we cannot provide a formal reduction here. The reduction fails due to the definition of UBS, since enforcing stretch $1+\epsilon$ for every subpath may help the algorithm in ruling out certain subpaths (or item combinations for KNAPSACK). For the classical definition of *stretch* $\ell(P) < (1+\epsilon) \cdot dist_\ell(s,t)$ instead of uniformly bounded stretch the reduction is straightforward: set $\epsilon := L_1/dist_\ell(s,t) - 1$, then KNAPSACK has a solution if and only if the shortest path P between s and t with respect to $\tilde{\ell}$ such that $\ell(P) < (1+\epsilon) \cdot dist_\ell(s,t)$ is short, i.e., $\tilde{\ell}(P) < L_2$.

3.2 Algorithms

We focus on two algorithms to find shortest smooth paths, and how to combine them with known acceleration techniques. The first algorithm solves our problem to optimality but may have exponential running times, whereas the second algorithm runs in essentially linear time at the cost of exactness.

3.2.1 Iterative Path Blocking

The first method to compute shortest smooth paths is based on iterative blocking of paths violating the UBS constraint. We maintain a set B of blocked paths. Then we iteratively repeat the following two steps.

1. Compute the shortest $\tilde{\ell}$ path avoiding all subpaths in B. This can be done with a variant of Dijkstra's algorithm. Whenever we are about to evaluate a node v that is the endpoint of a path $Q \in B$, we backtrack the parent pointers of v comparing the reconstructed path to Q. If the two paths match, we prune the search at v.
2. We check the uniformly bounded stretch by several shortest ℓ path computations. If we find no violation, the algorithm terminates, otherwise all violating subpaths are added to B, and we perform the first step again.

This algorithm solves our problem to optimality since we start with the shortest $\tilde{\ell}$ path and find longer and longer paths until we find a path with $\mathsf{UBS}_\ell \leq 1+\epsilon$. We can use CRP to perform one iteration step in our scenario efficiently. We use all nodes v being part of a path in B as *anchors* for the exploration, i.e., we make sure we skip no cell containing any node in B [25]. As long as the number of blocked paths is not too high, this is efficient. The second step ignores the blocked paths, so we can just use a many-to-many query algorithm [23]. Identifying violations can be done by a single sweep over the path, checking the computed distances.

Faster UBS Computation As experiments indicate, the most expensive part of this approach is the UBS computation. One possible optimization is to use *hub labels* [26] for accelerating this, but even with compression [27] the memory footprint of hub labels is rather large. A natural heuristic improvement is to limit the shortest ℓ path computations to a certain radius r. The problem with this approach is that we might accept violating paths.

3.2.2 Best Via-Node Paths

The main problems with iterative path blocking are that the number of paths we need to evaluate before we find a non-violating path may be exponential, and exact UBS computations are costly. As a more efficient

alternative, we propose to only consider single via-node shortest ℓ paths, which are concatenations of two shortest ℓ paths. The properties of such paths have been studied in the context of finding good alternate routes in road networks [19]. The number of such path candidates is linear, and by simply sorting all these path by $\tilde{\ell}$, filtering all paths that have an $\mathsf{UBS}_\ell \leq 1+\epsilon$, we ensure to find the best among this limited set of candidates.

An advantage of this approach is that known techniques for finding alternate routes can also compute and rank candidates efficiently. In the following, we describe these techniques. We perform two searches with length function ℓ, a forward search from s and a backward search from t. We can stop each search as soon as we scan a node with a distance greater than $(1+\epsilon) \cdot dist(s,t)$. During exploration we keep an extra distance label for the length of the shortest ℓ path according to $\tilde{\ell}$. With a linear sweep over all nodes scanned by both directions, we can then construct a sorted list of paths according to length function $\tilde{\ell}$. We check the paths for their UBS_ℓ by increasing length. As soon as we have found a path with $\mathsf{UBS}_\ell \leq 1+\epsilon$, we return it.

The most expensive part of that approach is the UBS_ℓ computation. We can avoid this by evaluating the *plateau* of each via node [19]. It is defined by the length of the overlap of the two constructed shortest-path trees. The plateau gives a lower bound on the local optimality of a path, which can be used to approximate the UBS of via-node paths. Due to [19, Lemma 4.3], the following holds. If a via-node path P with length $(1+\delta) \cdot dist(s,t)$ passes the so-called T *test* of length $\beta \cdot dist(s,t)$, P has a UBS of $\beta/(\beta-\delta)$. The length of the plateau gives a lower bound on $T/2$, which allows us to use the (easy to evaluate) plateau length to approximate the UBS of the path. As a result, we can determine a good path in the same time-frame as finding an alternative, which is known to be roughly six times slower than a bidirectional point-to-point shortest-path query [19].

Of course, running bidirectional Dijkstra is slow, but the alternate query algorithm harmonizes well with CRP [4]. The adaptation to our scenario is straightforward. We simply consider *all* via-node paths with a certain plateau length, unpack all paths, and sort them by $\tilde{\ell}$. Unfortunately, when using CRP instead of Dijkstra, we may end up with a worse path since we no longer consider all possible via-node paths, and plateau lengths may be underestimates. However, this makes computing shortest smooth paths feasible for interactive systems. Another advantage is that with this approach, we can provide alternate routes without much additional computational effort.

3.3 Application to Traffic-Aware Routing

We describe how we could apply the techniques for shortest smooth paths to find routes in traffic. As mentioned above, in this context, the primary length function ℓ is time-independent, ignores traffic, and may incorporate certain penalties. In the ideal case, the secondary length function $\tilde{\ell}$ is time-dependent, traffic-aware (including predictive and historical models), and minimizes travel time only. Recent work showed how one can indeed handle such a time-dependent length function with CRP without losing too much flexibility and preprocessing times [28]. However, the approach is much more complicated than CRP with a time-independent length function. For the sake of simplicity, in this paper, we are using a time-independent length function that uses the *current* traffic situation.

4 On-Demand Customization

We introduce a new concept for CRP, called *on-demand customization*, which allows running queries *without* any customization phase. This turns out to be very useful for traffic-aware routing.

Even when computing routes with a time-independent traffic-aware length function, we must update it regularly, e.g., every k seconds. When using CRP, three approaches from the literature can be used. The first one is to mark all nodes of the graph affected by traffic as anchors such that cells containing traffic are not skipped [29]. Unfortunately, this is not feasible in practice because during rush hour most cells can no longer be skipped, which means we then basically fall back to Dijkstra's algorithm with significantly higher latency. The second option is to re-run the customization after each traffic update for all cells with changed traffic.

This is feasible and fast enough, especially when servers have GPUs [30]. However, it turns out that this is often a waste of resources. Moreover, with traffic only being valid for a limited time window, a single server may not respond to queries from all over the world until traffic changes again. In other words, not every server needs traffic-aware precomputed distances. The third is to run customization on a separate machine and distribute the updated shortcut costs to the routing servers. However, it turns out that this increases network traffic by a lot, requiring large amounts of data being distributed and synchronized regularly.

Our new concept of on-demand customization introduces a fourth option for CRP. Here, we assume we have not run the customization phase, and thus have no precomputed distances available. We perform a normal CRP query, but whenever the query algorithm requests the shortest-path distances of a cell, we compute them for that cell *on-demand*. In order to reuse these precomputed distances, we maintain an LRU cache for the distances of entire cells. Moreover, we make this approach more efficient by introducing the concept of *reduction rules* to further increase the cache hit rate. For example, when requesting the distances for a traffic-aware length function, we first check whether the cell is affected by traffic at all, and if not, we can fall back to a traffic-independent length function. The latter is more likely to be already in the LRU cache because it is not invalidated by a traffic update.

Note that on-demand customization in combination with reduction rules can be used beyond dynamic routing scenarios. For example, when offering user options, such as avoiding toll roads, we do not need to compute any distances for the avoid-toll length function if the cell contains no toll roads. We believe that this approach is superior to metric-independent speed-up techniques [6, 7], and opens the door for personalized route planning in dynamic settings.

5 Experiments

We implemented all algorithms in C++ and compiled them with LLVM 8.1.0 using full optimization.[1] All experiments were conducted on a Mac Pro with a quad-core Intel Xeon E5-1620 v2 CPU with 64 GiB of RAM, running macOS 10.12.4. All running times are sequential.

Methodology and Inputs We use a publicly available OpenStreetMap road network of Europe.[2] The input has 174 million intersections and 183 million road segments, of which 165 million are drivable in both directions. The input is an intersection-based graph with nodes representing intersections, including degree-2 intersections, and arcs representing roads. Similar to previous work [4], we augment the graph by introducing turn costs.

For CRP, we partition the input graph into six levels, such that each cell has at most 6 553 600, 819 200, 102 400, 12 800, 1 600, 200, and 25 intersections on each level, respectively. The resulting partition has 11 145 338 cells in total, with 147 702 436 shortcuts. The graph partitioning algorithm is based on inertial flow [22] enhanced by some heuristics to locally improve the partition. Computing this partition takes 130 minutes, while customization takes 100 seconds. As mentioned above, both times are sequential, and both steps benefit greatly from parallelization. We do *not* use the microcode optimization [4], which accelerates customization in certain scenarios at the cost of a rather large increase in memory.

We are interested in evaluating the potential impact of our methods on a production system, where random queries are less important. According to US census data, the mean driving commute time in the US is roughly half an hour, and most often below one hour.[3] For this reason, if not stated otherwise, we perform random *one-hour queries*. We generate these by selecting a source node uniformly at random, performing a Dijkstra search from this source, and selecting as the target the first scanned node that is more than 60 minutes away. When reporting query times, we report the execution time in milliseconds, including path unpacking, averaged over 100 queries.

We use as ℓ the length function of the provided graph, which is said to model driving times of a car. Since the input also provides coordinates for all intersections, we also derive the (approximate) physical length

[1] Unfortunately, we cannot provide the source code we used for our experimental evaluation since it falls under IP restrictions.
[2] `http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php`
[3] `https://www.census.gov/hhes/commuting/data/commuting.html`

and average speed of each road segment. Unfortunately, for $\tilde{\ell}$, we are not aware of any open traffic feed, and we cannot use a real traffic feed to report results. Instead, we try to capture the following essential effect of traffic: it is sometimes beneficial to leave a congested but more important road in favor of a less-congested lower-class road. We simulate this effect of traffic as follows: Let S be the average speed (in kph) of a road without traffic. We reduce S to 5 kph for each road with $S > 30$ kph with a probablity of 0.5%. While this model is far from being perfect, it highlights several features of our algorithms.

5.1 Iterative Path Blocking

Table 1 reports the performance of iterative path blocking. We evaluate the average number of iterations, the average increase in path length (compared to optimizing $\tilde{\ell}$ without any constraints due to ℓ), and the average running time in milliseconds. For running times, we report the time spent in both phases of the algorithm separately.

We observe that the UBS computation by many-to-many queries is indeed very expensive (as mentioned above, these times could be reduced using hub labels), but for small ϵ, the exploration phase can become costly as well. The main reason for this is that the algorithm needs to maintain all blocked paths and backtrack often. Interestingly, the total number of iterations remains small, even for rather small values of epsilon. Most importantly, by using a shortest smooth path instead of the real shortest path, the overall increase in travel time is small. For example, for $\epsilon = 0.05$, exploration times are still reasonable, and with only 5% increase over the shortest $\tilde{\ell}$ path, the resulting paths do not contain any undesired local detours. Moreover, for this value, the smooth path is only 0.36% slower than the shortest $\tilde{\ell}$ path. Reducing ϵ further results in unreasonable query times, mostly due to the great increase in blocked paths.

Keep in mind though, that the choice of ϵ highly depends on the quality of the traffic feed. Also, the query times are not fast enough for an interactive system, but as mentioned above, we could use hub labels to determine the UBS, and introduce heuristic improvements to reduce the number of blocked paths. This would bring query times to a practical level.

5.2 Live Via Node Ranking

Next, we evaluate the performance of our via-node ranking approach, which can be extended to also provide alternates (another expected feature for driving navigation systems). We compute all via nodes according to ℓ, and rank the corresponding routes according to $\tilde{\ell}$. We evaluate the number of candidates we sort by $\tilde{\ell}$ on average, the quality of the routes (compared to the route found by the iterative approach), and the average execution times.

We observe that execution times are fast, and below 30 milliseconds for $\epsilon \leq 0.1$. Moreover, the additional increase over the route found by the iterative approach is small as well ($< 1.5\%$). Interestingly, the error first stays roughly the same with decreasing ϵ, but then goes up. A reason for this is that for very small ϵ, the number of valid candidates is small (only 2.2 on average for $\epsilon = 0.01$), which results in missing many reasonable routes. Still, a big advantage of the via-node approach is that it provides alternates for free, and $\epsilon = 0.01$ seems too strict of a choice. Keep in mind, however, that the quality of the produced routes by this approach gets worse if traffic gets worse: it gets more and more likely that we miss an interesting route because it cannot be constructed from two concatenated shortest ℓ paths. So, running the iterative approach in heavy traffic scenarios is an interesting option.

5.3 On-Demand Customization

Finally, we evaluate our new concept of on-demand customization. Table 3 reports the average query times when performing different types of queries for different types of customizations. Note that we here use ℓ as length function, and compute minimum-cost paths, including path unwinding. We compare computing shortcuts on demand, and precomputing them. We then report the average runtime for various numbers of queries, *including* the time to compute shortcuts. To show-case a key advantage of on-demand customization, we also perform *clustered* queries. For this, we use the bounding box of London (given by latitudes and

longitudes (51.3, -0.3) and (51.65, 0.15)) and perform queries only between nodes in that bounding box, resulting in average travel times of roughly 90 minutes. Note, however, that we still use the full European road network as input, so routes may leave and re-enter the bounding box. The motivation behind this is that in a production system we could shard queries such that a server gets queries from a specific region. Note that sharding queries instead of sharding the input allows flexible allocation of resources. Finally, we evaluate queries where we select source and target of the query uniformly at random, resulting in random long-range queries that span half of Europe on average. The average running times of Dijkstra's algorithm are approximately 300 ms for random one-hour queries, 2 seconds for queries in London, and 5 minutes for long-range queries, respectively. For all queries in this experiment, we size the LRU cache such that we do not evict shortcuts. The exact choice for the size of the LRU cache depends on how we shard queries.

We observe that on-demand customization performs really well for the clustered queries, which we are interested in most. It is faster than running Dijkstra's algorithm, and it is an order of magnitude faster than performing a full customization step. As already mentioned, the main motivation for this setup is for computing routes with a live traffic feed, which invalidates precomputed shortcuts regularly. During that timeframe, we can only run a limited number of queries, hence sharding queries by region is worth the effort. One potential disadvantage is that the first queries after a traffic update are rather slow, but this can be improved by using parallelization for customization. However, in production systems, we are not only interested in latency, but also throughput (how many requests can be handled per second). For this metric, on-demand customization gives the best numbers.

Even for random (long-range and 1-hour) queries, on-demand customization provides good results, but the advantages are less clear. For one-hour queries, only if the number of queries is high enough before cells are invalidated, it pays off to use CRP at all. For these queries, plain Dijkstra needs 300 ms to compute the minimum length path, which is fast enough. Another interesting observation is that running Dijkstra queries in London is much more expensive than random one-hour queries, which is likely due to the higher density of the road network in London. As a consequence, while random one-hour queries are fast enough with plain Dijkstra, performing random queries in London requires CRP.

For long-range queries, on-demand customization does not pay off. In fact, it is slightly slower than using precomputed shortcuts, mostly due to the overhead introduced by using an LRU cache for storing distances. Still, on-demand customization provides a good trade-off between customization effort and query performance.

6 Conclusion

We studied computing routes in traffic. Starting from the observation that computing shortest paths with a traffic-aware length function can lead to undesirable routes, we introduced the concept of shortest smooth paths. We introduced two techniques to compute such paths, and we evaluated them on realistic inputs. The first technique solves the problem to optimality, but may have exponential running times, whereas the second approach is roughly as performant as computing alternate routes with CRP, but at the cost of exactness. Experiments demonstrate that the second approach also finds good results. Moreover, we introduced the concept of on-demand shortcut computation for CRP, which allows running CRP queries *without* customization, enabling personalized route planning with CRP.

Regarding future work, we are interested in finding exact shortest smooth paths even faster, and we would like to use a time-dependent traffic-aware length function. However, we believe that for this, traffic models must become more accurate. We are also interested in other definitions of good routes in traffic, and more generally, a rigorous definition of route quality would be desirable. Finally, since on-demand shortcut computation (when used in a multi-threaded environment) enables truly personalized route planning, we are interested in making personalization practical.

Table 1: Performance of Iterative Path Blocking. We evaluate the increase over the shortest $\tilde{\ell}$ path, the number of iterations to find the smooth path, and the number of total blocked paths. The query times (in milliseconds) are reported separately for the time spent in finding paths, and the time for computing the UBS.

	increase			time [ms]		
ϵ	[%]	# it.	$\lvert B\rvert$	explor.	UBS	total
1.00	0.09	1.07	2	1.4	1 353	1 355
0.50	0.20	1.16	106	6.9	1 421	1 428
0.20	0.21	1.16	486	33.0	1 430	1 463
0.10	0.31	1.24	2 052	119.8	1 538	1 658
0.05	0.36	1.31	3 144	162.7	1 560	1 722
0.01	0.48	1.48	33 756	6 636.8	1 865	8 502

Table 2: Performance of Via Node Ranking. We evaluate the increase over the best smooth $\tilde{\ell}$ path found by our iterative approach, the number of candidates evaluated, and the average execution times (in milliseconds).

ϵ	# cand.	incr. [%]	time [ms]
1.00	347.0	0.92	172.09
0.50	247.4	0.86	107.01
0.20	125.3	0.85	47.99
0.10	65.8	0.84	23.42
0.05	31.2	0.84	11.34
0.01	2.2	1.47	2.42

Table 3: Average query times for computing and unpacking minimum-cost paths *including* the time to compute shortcuts for various customization types (on-demand and precomputed shortcuts) and different types of requests. All query times are given in milliseconds. The average running times of Dijkstra's algorithm are approximately 300 ms for random one-hour queries, 2 seconds for queries in London, and 5 minutes for long-range queries, respectively.

	1-hour		**London**		**long range**	
# quer.	on-dem	full	on-dem	full	on-dem	full
10	1 164	9 736	838	9 762	11 026	10 313
100	503	978	88	975	1 167	1 063
1 000	107	99	10	99	152	138
10 000	12	11	2	11	53	49

References

[1] A. Orda and R. Rom, "Shortest-path and minimum delay algorithms in networks with time-dependent edge-length," *Journal of the ACM*, vol. 37, no. 3, pp. 607–625, 1990.

[2] H. Bast, D. Delling, A. V. Goldberg, M. Müller–Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," *Algorithm Engineering* 2016: pp. 19–80

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical hub labelings for shortest paths," in *Proceedings of the 20th Annual European Symposium on Algorithms (ESA)*, 2012, pp. 24–35.

[4] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable route planning in road networks," *Transportation Science*, vol. 51, no. 2, pp. 566–591, 2017.

[5] J. Dibbelt, B. Strasser, and D. Wagner, "Customizable contraction hierarchies," *ACM Journal of Experimental Algorithmics*, vol. 21, no. 1, pp. 1.5:1–1.5:49, April 2016.

[6] S. Funke and S. Storandt, "Personalized route planning in road networks," in *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2015.

[7] S. Funke, S. Laue, and S. Storandt, "Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees," in *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA)*, 2017, pp. 23:1–23:13.

[8] S. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis, "Analysis and experimental evaluation of time-dependent distance oracles," in *Proceedings of the 17th Meeting on Algorithm Engineering and Experiments (ALENEX)*, 2015, pp. 147–158.

[9] S. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis, "Engineering oracles for time-dependent road networks," in *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX)*, 2016.

[10] S. Kontogiannis and C. Zaroliagis, "Distance oracles for time-dependent networks," *Algorithmica*, pp. 1–31, 2015.

[11] S. Lim, C. Sommer, E. Nikolova, and D. Rus, "Practical route planning under delay uncertainty: Stochastic shortest path queries," in *Robotics: Science and Systems VIII*, 2012, pp. 249–256.

[12] S. Samaranayake, S. Blandin, and A. Bayen, "Speedup techniques for the stochastic on-time arrival problem," in *Proceedings of the 12th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, 2012, pp. 83–96.

[13] S. Erb, M. Kobitzsch, and P. Sanders, "Parallel bi-objective shortest paths using weight-balanced B-trees with bulk updates," in *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA)*, 2014, pp. 111–122.

[14] S. Funke and S. Storandt, "Polynomial-time construction of contraction hierarchies for multi-criteria objectives," in *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX)*, 2013, pp. 31–54.

[15] R. Geisberger, M. Kobitzsch, and P. Sanders, "Route planning with flexible objective functions," in *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2010, pp. 124–137.

[16] E. Machuca and L. Mandow, "Multiobjective heuristic search in road maps," *Expert Systems with Applications*, vol. 39, no. 7, pp. 6435–6445, June 2012.

[17] G. Tsaggouris and C. Zaroliagis, "Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications," *Theory of Computing Systems*, vol. 45, no. 1, pp. 162–186, June 2009.

[18] R. Geisberger, M. Rice, P. Sanders, and V. Tsotras, "Route planning with flexible edge restrictions," *ACM Journal of Experimental Algorithmics*, vol. 17, no. 1, pp. 1–20, 2012.

[19] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Alternative routes in road networks," *ACM Journal of Experimental Algorithmics*, vol. 18, no. 1, pp. 1–17, 2013.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[21] D. Delling, D. Fleischer, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck, "An exact combinatorial algorithm for minimum graph bisection," *Mathematical Programming*, vol. 153, no. 2, pp. 417–458, 2015.

[22] A. Schild and C. Sommer, "On balanced separators in road networks," in *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA)*, 2015, pp. 286–297.

[23] S. Knopp, P. Sanders, D. Schultes, F. Schulz, and D. Wagner, "Computing many-to-many shortest paths using highway hierarchies," in *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007, pp. 36–45.

[24] M. Mihalák, R. Šrámek, and P. Widmayer, "Approximately counting approximately-shortest paths in directed acyclic graphs," *Theory of Computing Systems*, vol. 58, no. 1, pp. 45–59, 2016.

[25] D. Delling and R. F. Werneck, "Customizable point-of-interest queries in road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 3, pp. 686–698, March 2015.

[26] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "A hub-based labeling algorithm for shortest paths on road networks," in *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA)*, 2011, pp. 230–241.

[27] D. Delling, A. V. Goldberg, and R. F. Werneck, "Hub label compression," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA)*, 2013, pp. 18–29.

[28] M. Baum, J. Dibbelt, T. Pajor, and D. Wagner, "Dynamic time-dependent route planning in road networks with user preferences," in *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA)*, 2016, pp. 33–49.

[29] D. Schultes and P. Sanders, "Dynamic highway-node routing," in *Proceedings of the 6th Workshop on Experimental Algorithms (WEA)*, 2007, pp. 66–79.

[30] D. Delling, M. Kobitzsch, and R. F. Werneck, "Customizing driving directions with GPUs," in *Proceedings of the 20th International Conference on Parallel Processing (Euro-Par)*, 2014, pp. 728–739.