Improved Oracles for Time-Dependent Road Networks*†

Spyros Kontogiannis[1], Georgia Papastavrou[2], Andreas Paraskevopoulos[3], Dorothea Wagner[4], and Christos Zaroliagis[5]

1 **Department of Comp. Science & Engineering, University of Ioannina, Ioannina, Greece; and Computer Technology Institute and Press "Diophantus", Patras, Greece**
kontog@cse.uoi.gr

2 **Department of Comp. Science & Engineering, University of Ioannina, Ioannina, Greece; and Computer Technology Institute and Press "Diophantus", Patras, Greece**
gioulycs@gmail.com

3 **Department of Comp. Eng. & Informatics, University of Patras, Patras, Greece; and Computer Technology Institute and Press "Diophantus", Patras, Greece**
paraskevop@ceid.upatras.gr

4 **Karlsruhe Institute of Technology, Karlsruhe, Germany**
dorothea.wagner@kit.edu

5 **Department of Comp. Eng. & Informatics, University of Patras, Patras, Greece; and Computer Technology Institute and Press "Diophantus", Patras, Greece**
zaro@ceid.upatras.gr

Abstract

A novel landmark-based oracle (CFLAT) is presented, which provides earliest-arrival-time route plans in time-dependent road networks. To our knowledge, this is the first oracle that preprocesses combinatorial structures (collections of time-stamped min-travel-time-path trees) rather than travel-time functions. The preprocessed data structure is exploited by a new query algorithm (CFCA) which also computes (and pays for it) the *actual connecting path* that preserves the theoretical approximation guarantees. To make it practical and tackle the main burden of landmark-based oracles (the large preprocessing requirements), CFLAT is extensively engineered. A thorough experimental evaluation on two real-world benchmark instances shows that CFLAT achieves a significant improvement on preprocessing, approximation guarantees and query-times, in comparison to previous landmark-based oracles. It also achieves competitive query-time performance compared to state-of-art speedup heuristics for time-dependent road networks, whose query-times in most cases do *not* account for path construction.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Time-dependent shortest paths, FIFO property, Distance oracles

Digital Object Identifier 10.4230/OASIcs.ATMOS.2017.4

* A full version of the paper is available at https://arxiv.org/abs/1704.08445.

† Partially supported by EU FP7/2007-2013 under grant agreements no. 609026 (project MOVESMART), no. 621133 (project HoPE), and by DFG grant WA 654/23-1 within FOR 2083.

17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017).
Editors: Gianlorenzo D'Angelo and Twan Dollevoet; Article No. 4; pp. 4:1–4:17

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The surge for efficient solutions (min-cost paths) in networks with temporal characteristics is a highly challenging research goal, due to both the large-scale and the time-varying nature of the underlying arc-cost metric. Along this line, the development of practical algorithms for providing earliest-arrival-time route plans in large-scale road networks accompanied with a time-dependent arc-travel-time metric (known as *Time-Dependent Route Planning* – TDRP), has received a lot of attention in the last decade. TDRP is a hard challenge, both theoretically and in practice. For certain tractable cases, there is an analogue of Dijkstra's algorithm (called *Time-Dependent Dijkstra* – `TDD`) to solve the problem in quasi-linear time, which is already too much for a route-planning application supporting real-time query responses in *large-scale* road networks. Time-dependence is also by itself a quite important degree of complexity, both in space and in query-time requirements. These two challenges have been tackled in the past either by oracles, or by speedup heuristics. An *oracle* is a preprocessed and succinctly stored data structure encoding min-cost path information for carefully selected pairs of vertices. This data structure is accompanied with a query algorithm, which responds to arbitrary queries in time *provably better* than the corresponding Dijkstra-time and, if approximate solutions are also an option, with a *provable* approximation guarantee (stretch). Analogously, a *speedup heuristic* preprocesses arc-cost metrics which are custom-tailored to road networks, and then uses a query algorithm for responding to (exact or approximate) min-cost path queries in time that is in practice *several orders of magnitude* faster than the running time of Dijkstra's algorithm.

Modeling Instances, Problem Statement & Related Work. We model road network instances by directed graphs in which every arc $a = uv$ depicts an uninterrupted portion of a road segment and is accompanied by an arc-travel-time *function* $D[a]$ determining the time to traverse a, given the departure-time from its tail u. These functions are assumed to be continuous, piecewise-linear (pwl), periodic with one-day period, and are succinctly represented as sequences of consecutive *breakpoints*, i.e., (departure-time,arc-travel-time) pairs. This model is typical in the literature when we seek for route plans for private cars (e.g., [8, 9, 5, 20, 6, 2, 11, 21, 18, 17, 14, 3, 15]). For an arbitrary pair (o, d) of origin-destination points, there are two main algorithmic challenges:

- **(i)** $TDRP(o, d, t_o)$ concerns the computation of a minimum travel-time od-path for a given departure-time t_o, i.e., the *evaluation* of the minimum-travel-time function $D[o, d](t_o)$ from o to d;
- **(ii)** $TDRP(o, d)$ concerns the construction and succinct representation of the entire function $D[o, d]$, for *all* possible departure-times (e.g., for future instantaneous evaluations).

A crucial property that makes $TDRP(o, d, t_o)$ tractable is the FIFO property, according to which delaying the departure-time from the tail of an arc cannot possibly cause an earlier arrival at its head (i.e., the arcs behave as FIFO queues). For FIFO-abiding instances, a time-dependent variant of Dijkstra's algorithm (`TDD`) running in quasi-linear time is known [10, 22]. Without the FIFO property the problem can become extremely hard, depending on the adopted waiting policy at the vertices of the network [22]. As for $TDRP(o, d)$, this is known to be hard even when the FIFO property holds [11]. Fortunately, if (good) upper-approximations $\overline{\Delta}[o, d]$ of the minimum-travel-time functions $D[o, d]$ are an option, then there exist polynomial-time and space-efficient *one-to-one* [4, 11, 21], or *one-to-all* [15, 17, 18] approximation algorithms.

As a quality measure, independent of the query at hand, the *relative error* is typically used, i.e., the *maximum absolute error* (MAE) divided by the optimal travel-time; the MAE is the worst-case difference of an optimal travel-time from the proposed (path's) travel-time.

Several speedup heuristics, with remarkable success in road networks possessing scalar arc-cost metrics, have been extended to the case of TDRP. Some of them [6, 7, 20] are based on (scalar) lower bounds of travel-time functions (e.g., free-flow travel-times) to orient the search for a good route. `TDCALT` [6] yields reasonable query-response times for $TDRP(o, d, t_o)$, and `TDSHARC` [5] provides in reasonable time solutions to $TDRP(o, d)$, even for continental-size networks. `TDCRP` [3] is currently one of the most successful speedup heuristics, whose main feature is *customizability*, i.e., almost real-time adaptation to changes in the arc-cost metric. `TCH` [2] also achieves remarkable query times, both for $TDRP(o, d, t_o)$ and for $TDRP(o, d)$, even for continental-size networks. All the above mentioned heuristics only compute (estimations of) erarliest-arrival-times, excluding the overhead for constructing the corresponding connecting path. The only heuristics that also account the path construction in their query-times are provided in [23], with quite competitive performances.

In parallel to speedup heuristics, there has been a recent trend to provide oracles for TDRP, with *provable* theoretical performance and approximation guarantees [17, 18], which have been experimentally evaluated on real-world instances [14, 15]. The most successful one, `FLAT` [15, 17], demonstrated in practice noticeable query times and relative errors, much better than the theoretical guarantees, thus being competitive to the aforementioned speedup heuristics, justifying further research on providing even better oracles for TDRP, for the additional reason that oracles also ensure scalability.

Contributions and Outline. We present, engineer and experimentally evaluate `CFLAT` (Section 2), a novel *landmark-based* oracle for TDRP whose objective is to tackle the main burden of such oracles, the large preprocessing requirements, without compromising either the preprocessing scalability, the competitiveness of query-times, or the stretch. To our knowledge, `CFLAT` is the first oracle for time-dependent networks that preprocesses only time-evolving *combinatorial structures*: it maintains a carefully selected collection of time-stamped min-cost-path trees which can assure good approximation guarantees while minimizing the required space. Computing (and storing) less during preprocessing, unavoidably leads to more demanding work per query in real-time. Nevertheless, our novel query algorithm (`CFCA`) manages to achieve better query times and significantly improved practical performance compared to previous oracles, despite the fact that it accounts also the path construction[1]. Our specific contributions are threefold:

(i) We propose `CTRAP` (Section 2.2.1), a novel *approximation method* which stores only min-cost-path trees for carefully selected landmark vertices and sampled departure-times. Apart from the obvious economy of space due to omitting certain attributes (travel-time values), the novelty of this approach is that it exploits the fact that there are significantly fewer changes in the combinatorial structure, than in the functional description of the optimal solution. Moreover, we avoid multiple copies of the same preprocessed information, by organizing the destinations from a landmark into groups of (roughly) equidistant vertices, for which the common departure-times sequence is stored only once. We then proceed with the *landmark selection policies* (Section 3)

[1] Most of the existing oracles and speedup techniques in the literature only account for the estimation of a good upper bound on the minimum travel-time for the query at hand. Nevertheless, for time-dependent instances the path construction is not negligible, as is the case for static instances.

considered by CFLAT. Apart from the most successful ones in [15], we also consider new policies based on the betweeness-centrality measure. Due to the significant reduction in space requirements, we are able to select much larger landmark sets, which allows us to showcase the full scalability of CFLAT in trading smoothly preprocessing requirements with query performance (response time and stretch).

(ii) We propose CFCA(N) (Section 2.2.2), a novel *query algorithm* that exploits the preprocessed information of CFLAT: For a query (o, d, t_o), it starts by growing a TDD ball from o at time t_o, until the N closest landmarks are settled. It then marks a small subset of relevant arcs, using the N settled landmarks as "attractors" that orient the discovery of certain paths from d back to o. This is reminiscent of the ARCFLAGS algorithm for static metrics [12], but the choice of the relevant arcs is done "on the fly", since this information is also time-dependent. In the final step, it continues growing the initial TDD ball, but only within the subgraph of marked arcs, until d is settled within this subgraph. CFCA(N) achieves the same theoretical approximation guarantee with the query algorithm FCA(N) of FLAT; the observed stretch though, is in practice much better than the one of FCA(N).

(iii) We conduct a thorough *experimental evaluation* of CFLAT (Section 3), on two well established real-world instances, the urban area of Berlin and the national road network of Germany. Our findings are perceptible. For Berlin, the preprocessing requirements are 3.14sec and 0.7MB per landmark. Thus, if space is our primary concern, we can preprocess 250 random landmarks in about 13min, consuming 0.17GiB space, whereas the query performance (average query time and relative error) varies from 0.486msec and 0.02418 (for $N = 1$), to 2.758msec and 0.00136 (for $N = 6$). With 16K landmarks the query performance varies from 0.064msec and 0.00227 (for $N = 1$), to 0.214msec and 0.00019 (for $N = 6$). As for Germany, the preprocessing requirements are 26.052sec and 8.466MiB per landmark. For $4K$ landmarks, we achieve a query performance varying from 0.585msec and 0.0079 (for $N = 1$), to 3.434msec and 0.00047 (for $N = 6$).

Details omitted due to space limitations as well as further experiments can be found in the full version [16].

2 The CFLAT Oracle

A landmark-based oracle selects a set $L \subseteq V$ of *landmarks* and preprocesses travel-time information (*summaries*) between them and all (or some) reachable destinations. A query algorithm exploits these summaries for responding to earliest-arrival-time queries (o, d, t_o), from an origin o and departure-time t_o to a destination d, in time that is *provably* efficient (e.g., sublinear in the size of the instance). The oracle is also accompanied with a *theoretically proved* approximation guarantee (a.k.a. stretch) for the quality of the recommended routes.

In Section 2.2 we present our novel oracle, CFLAT. Before doing that, we recap in Section 2.1 FLAT, an oracle upon which CFLAT builds and achieves remarkable improvements.

2.1 Recap of FLAT

FLAT is, to date, the most successful oracle for TDRP in road networks, and was originally presented and analyzed in [17]. A variant of FLAT was implemented and experimentally evaluated in [15]. In this work, we consider (and refer to as FLAT) to that variant. Its main building block is the TRAP approximation method: Given a landmark ℓ, the period $[0, T)$ is split into intervals of an (arbitrarily chosen) length $3,200$sec. The endpoints of these intervals are used as sampled departure-times. The corresponding min-cost-path trees

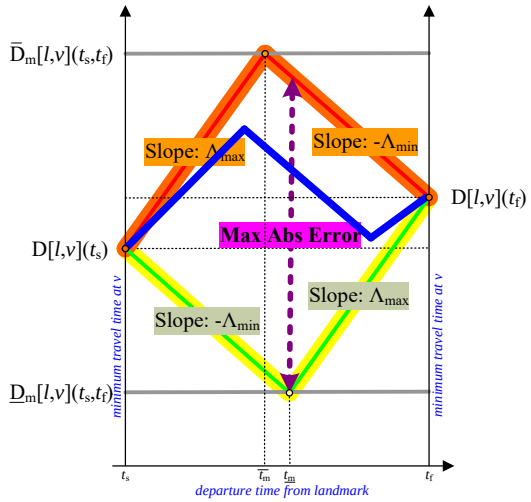


Figure 1 Upper-approximation $\overline{\delta}_k[\ell, v]$ (thick-orange) and lower-approximation $\underline{\delta}_k[\ell, v]$ (thick-green) of $D[\ell, v]$ (blue), within $[t_s, t_f)$.

rooted at ℓ are computed, producing travel-time values for all reachable destinations v. For each interval $[t_s, t_f)$, an upper-approximating function $\overline{\delta}$ is considered, which is the lower-envelope of a line of max slope ($\Lambda_{\max}$) passing via $\langle t_s, D[\ell, v](t_s)\rangle$ and a line of min slope ($-\Lambda_{\min}$) passing via $\langle t_f, D[\ell, v](t_f)\rangle$ (cf. Figure 1). Observe that $\overline{\delta}$ considers an *intermediate breakpoint* $\langle \overline{t}_m, \overline{D}_m\rangle$, the intersection of the two lines, which is *not* the outcome of an actual sampling. This intermediate breakpoint is only stored when v becomes deactivated (i.e., within this interval there is no need for further sample points, see next paragraph). A similar lower-approximating function $\underline{\delta}$ is considered, which is the upper-envelope of a min-slope line passing via $\langle t_s, D[\ell, v](t_s)\rangle$ and a max-slope line passing via $\langle t_f, D[\ell, v](t_f)\rangle$.

A closed-form expression of the worst-case error (*maximum absolute error* – MAE) is used to determine whether $\overline{\delta}$ is a sufficient upper-approximation of $D[\ell, v]$ within $[t_s, t_f)$, given a required approximation guarantee $\varepsilon > 0$. If this is the case, v becomes *deactivated* for this subinterval, meaning that no more sampled trees will be of interest for v within it. `TRAP` continues by choosing finer sampling intervals, first of length $1,600$sec, then 800sec, 400sec, etc., computing min-cost-path trees only for the new departure-time samples in each round, until eventually there is no active destination for any of subintervals of the currently chosen length. The concatenation of all the upper-approximations for the smallest active subintervals of v is considered by `TRAP` as the required $(1+\varepsilon)$-upper-approximation $\overline{\Delta}[\ell, v]$ (called a *travel-time summary*) of $D[\ell, v]$ within $[0, T)$. $\overline{\Delta}[\ell, v]$ is stored as a sequence of pairs of breakpoints, i.e., (departure-time,travel-time) pairs, in increasing order w.r.t. departure-times. During the preprocessing, `FLAT` calls `TRAP` to produce travel-time summaries, from a carefully selected set of landmark vertices towards all reachable destinations.

Upon a query (o, d, t_o) `FLAT` calls `FCA(N)`[2], a query algorithm which grows a `TDD` ball from o with departure-time t_o, until either d or the first N landmarks are settled. It then returns either the exact route (when d is settled), or the best-of-N (w.r.t. the *theoretical guarantees*) *od*-path passing via one of the N settled landmarks and being completed (from ℓ to d) by exploiting the preprocessed summaries for d. Since `FCA(N)` does not need all summaries to

[2] In [15] it was called `FCA`$^+$, with a fixed number $N = 6$ of landmarks to settle.

be concurrently available in memory, the preprocessed data blocks representing travel-time summaries of FLAT were compressed, and only summaries of the landmarks required per query were decompressed on the fly. The zlib library was used for this purpose, leading to a reduction of 10% in the required space. More details on FLAT are provided in [15, 17].

2.2 Description of CFLAT

We now present CFLAT, which can be considered as the combinatorial analogue of FLAT. At a high level, CFLAT works as follows. In a preprocessing phase, it constructs and compactly stores min-cost-path trees at carefully sampled departure-times, rooted at each landmark $\ell \in L$. A query (o, d, t_o) is answered by first growing a TDD ball from o at time t_o, until either d or a small number of landmarks are settled. In the latter case, starting from d, a suitably small subgraph is constructed (consisting of certain paths going from d back to o, using the settled landmarks as "attractors"), until a settled vertex of the initial TDD ball is reached. Then, a continuation of growing the initial TDD ball on the resulted small subgraph returns an *od* path that turns out to approximate very well the optimal *od* path.

2.2.1 The Approximation Method CTRAP and CFLAT Preprocessing

CTRAP computes and stores only min-cost-path trees at carefully sampled departure-times, rather than actual breakpoints of the corresponding minimum-travel-time functions. The algorithm's pseudocode is provided in the full version of the paper [16]. We present here only a sketch of the main steps as well as the key new insights, compared to TRAP. CFLAT preprocessing consists simply in calling CTRAP(ℓ, ε) for each landmark $\ell \in L$.

procedure CTRAP(ℓ, ε)

STEP 1: Keep sampling finer departure-times from $[0, T)$, as in TRAP, until all destinations achieve relative error less than ε and become inactive.
1.1: Store (pruned at inactive nodes) min-cost-path trees from ℓ, for all departure-times.
1.2: Omit intermediate breakpoints.

STEP 2: Merge consecutive breakpoints with identical predecessors.

STEP 3: Avoid multiple copies of common departure-time sequences.

When executed from a landmark ℓ, CTRAP works as follows: Step 1 resembles TRAP, the only difference being that CTRAP keeps only the immediate predecessors (parents) per active destination v in the sampled min-cost-path trees. In particular, a pair of sequences is created, $PRED[\ell, v]$ for predecessors and $DEP[\ell, v]$ for the corresponding sampled departure-times, per landmark-destination pair $(\ell, v) \in L \times V$. Step 2 cleans up each pair of sequences, by merging consecutive breakpoints for which the predecessor is the same. Step 3 organizes the destinations from a landmark ℓ into groups with the same departure-times sequence, so that multiple copies of the same sequence are avoided. In the rest of this section, we describe in more detail the key new insights and algorithmic steps of CTRAP, compared to TRAP [15, 17].

Store min-cost-path trees. For each leg of $\overline{\Delta}[\ell, v]$, we store pairs $\langle t_\ell, PRED[\ell, v](t_\ell)\rangle$ of departure-times t_ℓ from ℓ and the predecessor of v in the corresponding min-cost-path tree rooted at (ℓ, t_ℓ), omitting the actual min-travel-time values $D[\ell, v](t_\ell)$. This modification makes the oracle aware only of the min-cost-path-tree structures created during the repeated sampling procedure. Additionally, rather than storing repeatedly the IDs of predecessors,

which would be space consuming in networks with millions of vertices, we only store the position of the corresponding arc in the list of incoming arcs to a vertex v. Since the maximum in-degree in the road instances we have at our disposal is at most 7, we only need to consume 1 byte per storage for a predecessor. We could even consume 3 bits per predecessor, which could then be packed into only two bytes containing also the corresponding departure-time value (by an appropriate discretization of the departure-time values). We prefer *not* to combine predecessors with departure-times in the same bit string, because we shall exploit later the extensive repetition of identical sequences of departure-times, which nevertheless would be lost for strings also containing the predecessors. It was observed in both benchmark instances that about one half of all possible destinations per landmark ℓ appear to have a *unique* predecessor throughout the entire period of departure-times, $[0, T)$. For them we store their unique predecessor only once. For the remaining destinations though, even with only two possible predecessors, we have to store the entire sequence of predecessor-changes.

Omit intermediate breakpoints. `TRAP` computes, and explicitly stores, intermediate breakpoints $(\bar{t}_m, \overline{D}_m)$ between consecutive sampled breakpoints of $D[\ell, v]$, as the intersection points of the two legs involved in the definition of $\bar{\delta}[\ell, v](t)$ (cf. Figure 1), for each pair (ℓ, v) and those intervals where the MAE is sufficiently small and v becomes deactivated. In `CTRAP` we choose *not* to keep these intermediate breakpoints and restrict the preprocessed information only to the actual samples. We let the query algorithm deal with the missing information, whenever needed. This way we avoid storing approximately 10M (for Berlin) and 100M (for Germany) of intermediate breakpoints per landmark.

Merge sequences of breakpoints with identical predecessors. `CTRAP`'s next algorithmic intervention is based on the observation that the vast majority of all destinations appear to have on average 2 alternating predecessors throughout the entire period $[0, T)$. To save space, we choose to merge *consecutive* sampled breakpoints for v of the form $\langle t_\ell, x = PRED[\ell, v](t_\ell)\rangle$ and $\langle t'_\ell, x = PRED[\ell, v](t'_\ell)\rangle$, i.e., possessing the same predecessor. This leads to a reduction in the number of breakpoints to store, but also has a negative influence on the similarities of the departure-times sequences, and thus on the repetitions that we could avoid (see next heuristic). However, there is still positive gain by applying both this heuristic and that for avoiding multiple copies of departure-times sequences.

Avoid multiple copies of common departure-time sequences. `CTRAP`'s next key insight is based on the fact that it is a repeated-sampling method which probes (at common departure-times for all destinations) min-cost-path trees from a landmark ℓ, starting from a coarse-grained sampling towards more fine-grained samples of the entire period $[0, T)$, until the MAE guarantee is satisfied for all reachable destinations from ℓ. A destination v may not care for all these departure-times, because the value of MAE may be satisfied at an early stage for it. This indeed depends on the actual minimum travel-time $\min\{D[\ell, v](t_s), D[\ell, v](t_f)\}$ at the endpoints of each given subinterval $[t_s, t_f)$. For each landmark-destination pair (ℓ, v), we store the sequences $DEP[\ell, v]$ of necessary departure-times and $PRED[\ell, v]$ of the corresponding predecessors. The crucial observation is that destinations which are (roughly) at the same distance from ℓ are anticipated to have the same sequence of sampled departure-times, possibly differing only in their sequences of predecessors. It is clearly a waste of space to store two identical sequences $DEP[\ell, v] = DEP[\ell, u]$ more than once, even if the corresponding sequences of predecessors differ. Thus, we store each departure-times sequence as soon as it first appears for some destination v, and consider v as the *representative*

of all other destinations u for which $DEP[\ell, u] = DEP[\ell, v]$. For each non-representative destination u, we store $PRED[\ell, u]$ and the corresponding representative v. Our next challenge is to efficiently compare departure-times sequences. To avoid a potential blow-up of the preprocessing time, we do not compare them point-by-point. Instead, we assign to every sampled departure-time t_ℓ two **iuar**[3] chosen floating-point numbers $w_1(t_\ell), w_2(t_\ell)$ from the interval $[1.0, 100.0]$. Each destination u adds the two values $w_1(t_\ell) \cdot t_\ell$ and $w_2(t_\ell) \cdot t_\ell$ to its own hash keys, i.e., $H_1[u] = H_1[u] + w_1(t_\ell) \cdot t_\ell$ and $H_2[u] = H_2[u] + w_2(t_\ell) \cdot t_\ell$, *only when* t_ℓ is indeed a necessary sample for u. Otherwise, the hash keys of u remain intact. At the end of the sampling process, we sort lexicographically the hash pairs of all destinations, in order to discover families of common departure-times sequences. We deduce that two destinations possess the same sequence when both their hash pairs match, in which case we verify this allegation by comparing them point by point. We observed that, for both benchmark instances, 80% of all destinations with at least two predecessors can be represented w.r.t departure-times by the remaining 20% of (representative) destinations.

Indexing preprocessed information. For retrieving efficiently the summaries from a landmark ℓ to each destination v, we maintain a vector of pointers per landmark, one pointer per destination, providing the address for the starting location of the summary for v. The pointers are in ascending order of vertex ID. The lookup time is $\mathcal{O}(1)$ and the required space for this indexing scheme is $\mathcal{O}(n \cdot |L|)$ additional bytes, where L is the chosen landmark set.

Speeding up preprocessing time. Handling only min-cost-path trees also has a collateral effect of speeding up the required preprocessing time. The reason for this is that we do not compute explicitly, each and every time that we sample travel-time values from ℓ, the exact shapes of the corresponding minimum-travel-time functions per destination. The travel-time summaries provided by `FLAT` were created based on this explicit computation of all the earliest-arrival *functions* per destination v, from each landmark ℓ. In contrast, the min-cost-path summaries of `CFLAT` are created without having to compute earliest-arrival functions. This leads to a reduction in the preprocessing time of more than 60%.

2.2.2 The Query Algorithm CFCA(N)

`CFCA(N)` is based on `FCA(N)` [15], but is fundamentally different from it in the sense that it exploits min-cost-path trees, and also considers the *od*-path construction as part of it, which was not the case for `FCA(N)`, and indeed for most of the query algorithms in the literature. N indicates the number of landmarks to be settled by `CFCA(N)` around the origin o. The pseudocode of the algorithm is presented in the next paragraph. `CFCA(N)` works as follows. In case that the destination d is already settled in Step 1, the resulting (exact) *od*-path can be computed by backtracking towards the origin, following the pointers to all predecessors. Otherwise, we proceed as follows. For each settled landmark ℓ, we have an optimal $o\ell$-path guaranteeing arrival-time $t_\ell = t_o + D[o, \ell](t_o)$ at ℓ. Since we do not have at our disposal travel-time values from ℓ towards d, or any other vertex, we are not able to compare ℓv-paths based on their (approximate) lengths. On the other hand, for the given departure-times t_ℓ and any vertex v, we can tell the predecessor(s) of v in the (at most two per landmark) most relevant min-cost-path trees, the ones at the consecutive sampled departure-times t_ℓ^- and t_ℓ^+ of each $DEP[\ell, v]$ for which it holds that $t_\ell \in [t_\ell^-, t_\ell^+)$.

[3] **iuar** = independently and uniformly at random, without repetitions.

procedure CFCA(N)

STEP 1: A TDD ball is grown from (o, t_o), until N landmarks are settled.
1.1: **if** d is already settled **then** **return** optimal solution.
1.2: For each settled landmark ℓ, $t_\ell = t_o + D[o, \ell](t_o)$.

STEP 2: An appropriate subgraph is recursively created from d.
2.1: $Q = \{ d \}$ /* Q is a FIFO queue */
2.2: **while** $\neg Q.Empty()$ **do** :
2.3: **if** $v = Q.Pop()$ is not explored from STEP 1's TDD ball **then** :
2.4: **for** each settled landmark ℓ of STEP 1 **do** :
2.5: Mark the arcs $\langle PRED[\ell, v](t_\ell^-), v\rangle$ and $\langle PRED[\ell, v](t_\ell^+), v\rangle$ leading to v, where $[t_\ell^-, t_\ell^+)$ is the unique interval in $DEP[\ell, v]$ containing t_ℓ.
2.6: $Q.Push(PRED[\ell, v](t_\ell^-))$; $Q.Push(PRED[\ell, v](t_\ell^+))$
2.7: **end for**
2.8: **end while**

STEP 3: **return** optimal od-path in the induced subgraph by (TDD ball of) STEP 1 and STEP 2.

CFCA(N) marks (per settled landmark ℓ) the connecting arcs from these most relevant predecessor(s) $PRED[\ell, v](^-_\ell)$ and $PRED[\ell, v](^+_\ell)$, towards v. All these discovered predecessors w.r.t. the N settled landmarks are inserted (if not already there) in a FIFO queue, which was initialized with d, so that, upon their extraction from the queue, they can provide in turn their own predecessors, etc. The recursive search for predecessors stops as soon as a vertex x in the explored area of the initial TDD ball of Step 1 is reached. CFCA marks then also the arcs of the corresponding short (not necessarily the shortest though, since x is explored but not necessarily settled) *ox*-path. This way we are guaranteed that in the subgraph of marked arcs there is already an *od*-path which has been oriented by (ℓ, t_ℓ) and passes via x. Step 2 of CFCA(N) terminates when the FIFO queue becomes empty, i.e., we no longer have to process intermediate vertices which are unexplored by Step 1. The actual path construction takes place in Step 3, which considers the subgraph induced by the marked arcs and continues growing the TDD ball from (o, t_o) within this subgraph. This path construction indeed leads to significantly smaller relative errors, since the resulting *od*-path is not only the best *prediction* among a given set of N paths induced by the N settled landmarks (as in FLAT), but actually the *optimal od*-path within the induced sugbgraph.

The worst-case approximation guarantee of CFCA(1) is $(1 + \varepsilon + \psi)$ (identical to that of FCA [17]), where ε is CTRAP's approximation guarantee and ψ is a constant depending on ε and the travel-time metric (but not on the size) of the network. Note that we could theoretically improve the stretch of CFCA(N) to $(1 + \sigma)$, for any constant $\sigma > \varepsilon$, and get a PTAS, by using in Step 1 the RQA algorithm [17]. We choose *not* to do so, because our previous experimental evaluation with FLAT [15] has shown that FCA(N) in practice dominates RQA.

3 Experimental Evaluation

Experimental Setup and Goal. Our algorithms were implemented in C++ (GNU GCC version 5.4.0) and Ubuntu Linux (16.04 LTS). All the experiments were conducted on a 6-core Intel(R) Xeon(R) CPU E5-2643v3 3.40GHz machine, with 128GB of RAM. We used 12 threads for the parallelization of the preprocessing phase. CFCA was always executed on a single thread. For the sake of comparison, we used the same set of 50,000 queries, **iuar** chosen from $V \times V \times [0, T)$ in each instance, for all possible landmark sets. The PGL library

[19] was used for graph representation and operations. Two benchmark instances were used, the first concerning the city of Berlin, and the second the national road network of Germany.

The main goal of our experimental evaluation was to investigate the scalability of `CFLAT`: how smoothly does it trade higher preprocessing requirements for better approximation guarantees and query-times. To demonstrate this, we aim at showcasing the performance of `CFCA(N)` for several types and sizes of landmark sets. We have also increased the typical size of the used landmark sets in our comparison of different landmark selection policies.

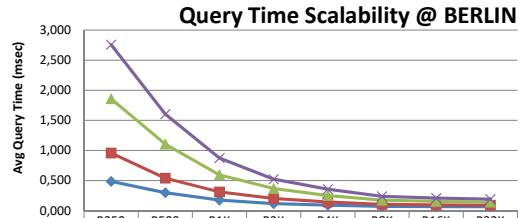
Landmark Selection Policies. Although the preprocessing requirements are proportional to $|L|$ (number of landmarks), they are essentially invariant of the landmark selection policy. However, as previous experimental evaluation indicated [15], the performance of the query algorithms has a strong dependence on the type of the landmarks. A key observation was that the *sparsity* of landmarks (not being too close to each other) as well as their *importance*, are crucial parameters. Therefore, in this work we insist in almost all cases (except for the RANDOM landmark sets which are used as baseline) on selecting the landmarks sparsely throughout the network. As for their importance, when such information is available, we also consider the selection of landmarks at junctions of an important road segment (as in [15]). Finally, we consider a new measure of vertex significance, the (approximate) betweeness-centrality measure. In particular, we consider the following landmark selection policies:

- ◇ RANDOM (R): **iuar** choice of landmarks.
- ◇ SPARSE-RANDOM (SR): Incremental **iuar** choice of landmarks, where each chosen landmark excludes a free-flow neighborhood of vertices around it from future landmark selections.
- ◇ IMPORTANT-RANDOM (IR): A variant of R which moves each random landmark to its nearest important vertex within a free-flow neighborhood of size 100. This policy is only applicable for the instance of Berlin which provides road-segment importance information.
- ◇ SPARSE-KAHIP (SK): We use the `KaFFPa` algorithm of the KAHIP partitioning software (v1.00) [13], setting the parameters so that there are many more boundary vertices than the required number of landmarks. The landmarks are incrementally and **iuar** chosen among the boundary vertices. Each landmark excludes a free-flow neighborhood from future selections.
- ◇ KAHIP-CELLS (KC). Starting with a KAHIP partition, one landmark per cell is incrementally and **iuar** chosen, excluding a free-flow neighborhood from future selections.
- ◇ BETWEENESS-CENTRALITY (BC): Vertices are ordered in non-increasing *approximate betweeness-centrality* (ABC) values [1]. Landmarks are selected incrementally according to ABC values, excluding a free-flow neighborhood from future selections.
- ◇ KAHIP-BETWEENESS (KB): For a KAHIP partition, incrementally choose as landmark the vertex with the highest ABC value in a cell, excluding a neighborhood from future selections.

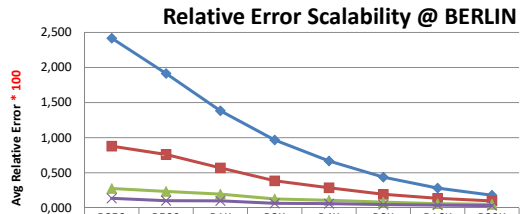
We finally consider the following systematic naming of the landmark sets. Each set is encoded as XY, where $X \in \{R, SR, IR, SK, KC, BC, KB\}$ determines the type of landmark set, and $Y \in \{250, 500, 1K, 2K, 3K, 4K, 8K, 16K, 32K\}$ determines its size.

3.1 Evaluation of CFLAT @ Berlin

For Berlin we have considered all types of landmarks. For each of them, we have used as baseline the size $Y = 4K$. $\{R, SR, IR, SK\}$ were considered also in [15] (but for smaller sizes), whereas $\{KC, BC, KB\}$ are new types. Especially for R we tried all possible values

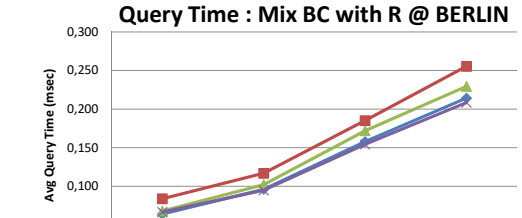


	R250	R500	R1K	R2K	R4K	R8K	R16K	R32K
CFCA(1)	0,486	0,299	0,176	0,116	0,091	0,070	0,067	0,065
CFCA(2)	0,958	0,540	0,311	0,202	0,144	0,104	0,095	0,090
CFCA(4)	1,859	1,105	0,593	0,367	0,250	0,174	0,154	0,141
CFCA(6)	2,758	1,605	0,873	0,523	0,356	0,237	0,209	0,191

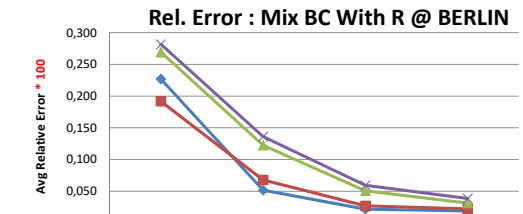


	R250	R500	R1K	R2K	R4K	R8K	R16K	R32K
CFCA(1)	2,418	1,915	1,383	0,967	0,668	0,438	0,282	0,180
CFCA(2)	0,880	0,760	0,570	0,385	0,287	0,193	0,136	0,098
CFCA(4)	0,276	0,234	0,196	0,127	0,108	0,082	0,059	0,050
CFCA(6)	0,136	0,102	0,100	0,064	0,060	0,047	0,038	0,032

Figure 2 Performance of `CFCA(N)` in Berlin, for random landmarks and 50,000 *random queries*. In the graph of relative errors, all values are multiplied by 100.



	N=1	N=2	N=4	N=6
BC16K	0,064	0,096	0,158	0,214
BC8K + R8K	0,084	0,117	0,185	0,256
BC4K + R12K	0,068	0,102	0,172	0,230
R16K	0,067	0,095	0,154	0,209

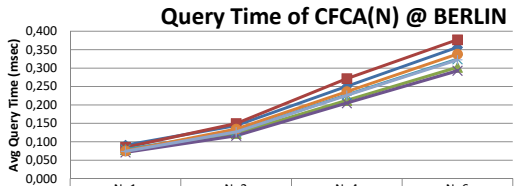


	N=1	N=2	N=4	N=6
BC16K	0,227	0,052	0,022	0,019
BC8K + R8K	0,192	0,068	0,027	0,022
BC4K + R12K	0,269	0,123	0,051	0,031
R16K	0,282	0,136	0,059	0,038

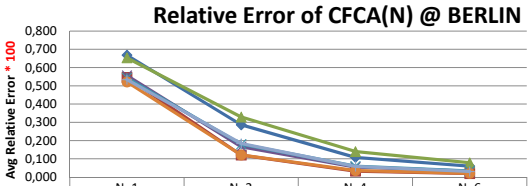
Figure 3 Performance of `CFCA` for mixtures (BC- and R-landmark types) of 16K landmarks in Berlin, and a query set of 50,000 *random queries*. All relative errors are multiplied by 100.

for Y, in order to showcase the scalability of `CFLAT` and its smooth trade-off of preprocessing requirements, query-times and stretch factors. Concerning *vertex-importance* (only available in Berlin), we considered as important those vertices which are incident to roads of category at most 3. As for *sparsity*, we set the sizes of the excluded free-flow ball per selected landmark to 150 vertices for SR, 100 for IR, 50 for SK, 20 for KC, 150 for BC, and 20 for KB. For KAHIP based landmark sets (SK, KC and KB) we used the following parameters: The number of cells to partition the graph was set to 4,000, having 13,256 boundary vertices in total. For SK we chose randomly 4,000 boundary vertices as landmarks. For KC and KB we chose one landmark per cell.

We first conducted an experiment to test the scalability of `CFCA`'s performance as a function of N and the number of landmarks, always for R-type landmarks. As is evident from Figure 2, the average errors decrease linearly and the query-times decrease quadratically, as we double the number of landmarks. Additionally, notable "quick-and-dirty" answers are possible with only 250 landmarks, which require space 0.17GiB; cf. [16]. In particular, the query performance (average query time and relative error) varies from 0.486msec and 0.02418 ($N = 1$), to 2.758msec and 0.00136 ($N = 6$). If query time is the main goal then, for R32K, the query performance of `CFCA` varies from 0.065msec and 0.0018 ($N = 1$), to 0.191msec and 0.00032 ($N = 6$). The best performance (cf. Figure 3) is achieved by BC16K, varying from 0.064msec and 0.00227 (for $N = 1$), to 0.214msec and 0.00019 (for $N = 6$). Since the average



	N=1	N=2	N=4	N=6
R4K	0,091	0,144	0,250	0,356
SR4K	0,085	0,150	0,271	0,377
IR4K	0,079	0,123	0,212	0,302
SK4K	0,071	0,116	0,205	0,293
KC4K	0,079	0,128	0,227	0,324
BC4K	0,075	0,134	0,236	0,338
KB4K	0,074	0,127	0,226	0,322



	N=1	N=2	N=4	N=6
R4K	0,668	0,287	0,108	0,060
SR4K	0,546	0,121	0,033	0,019
IR4K	0,653	0,329	0,140	0,078
SK4K	0,557	0,166	0,055	0,033
KC4K	0,544	0,181	0,060	0,033
BC4K	0,521	0,121	0,036	0,021
KB4K	0,534	0,184	0,058	0,031

Figure 4 Performance of `CFCA(N)` in Berlin, for $4K$ landmarks and $50,000$ *random queries*. In the graph of relative errors, all values are multiplied by 100.

query-time for `TDD` is 110.02msec[4], the speedup of `CFCA(1)` with BC16K is $1,719$.

Our next experiment compares landmark types of size 4K each (cf. Figure 4). Concerning query-times, the best curve is that of SK4K. As for relative errors, SR4K and BC4K are clear winners. Further experiments are reported in [16]. In comparison with `FLAT`, the query-performance of `CFCA(1)` for BC4K (0.075msec and 0.00521) dominates that of `FCA(1)` (0.081msec and 0.00771, cf. [15]). It is worth mentioning that, with only one fourth of the required space (e.g. using SR4K) we can achieve the best observable average error (0.00019) for Berlin. Of course, the query time deteriorates from 0.214msec (for BC16K) to 0.377msec. It is also observed that mixing BC-landmarks with R-landmarks is not indeed a good idea. For example, the pure BC16K and R16K landmark sets dominate all the mixed landmark sets we have tried, both w.r.t. the error and the query time (cf. Figure 3).

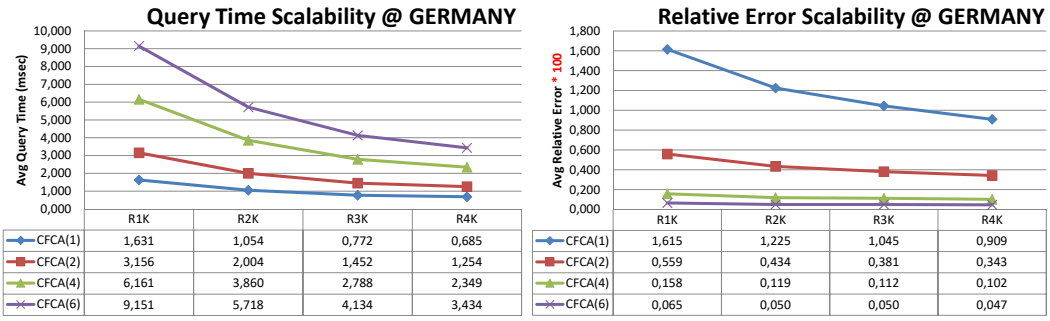
3.2 Evaluation of CFLAT @ Germany

We considered R-landmark sets of sizes from 1K to 4K. The rest of the landmark sets were of size 3K, with excluded neighborhood size $1,200$ vertices for SR3K, 350 for SK3K, and $1,000$ for BC3K. We started again with a demonstration of the scalability of `CFCA` on R-landmark sets, as a function of the number of landmarks (cf. Figure 5). The relative errors decrease linearly and the running times decrease quadratically, as we increase the number of landmarks. Relative errors of 0.00065 are achieved for `CFCA(6)` even with 1K landmarks which require 8.3GiB space, with query-time 9.151msec. Moreover, a "quick-and-dirty" answer of error at most 0.01615 is returned in only 1.631msec. As for the query performance of R4K, `CFCA(1)` achieves 0.685msec and 0.00909, and `CFCA(6)` has 3.434msec and 0.00047.

We proceeded next with a comparison of various landmark types of size 3K each (cf. Figure 6). For Germany we have a clear winner, SR3K, w.r.t. relative errors. As for query times, BC3K is preferable for $N \geq 4$ and SR3K is better for $N \leq 2$.

The best query performance for `CFCA` (see Table 1) is achieved for SR4K, and varies from 0.585msec and 0.007913 (for $N = 1$), to 3.572msec and 0.000177 (for $N = 6$). Thus, the best achieved speedup over `TDD` (whose average running time is $1,1190.873$msec) is more than 2035 in this case. As for Berlin, mixing BC-landmarks with R-landmarks does not

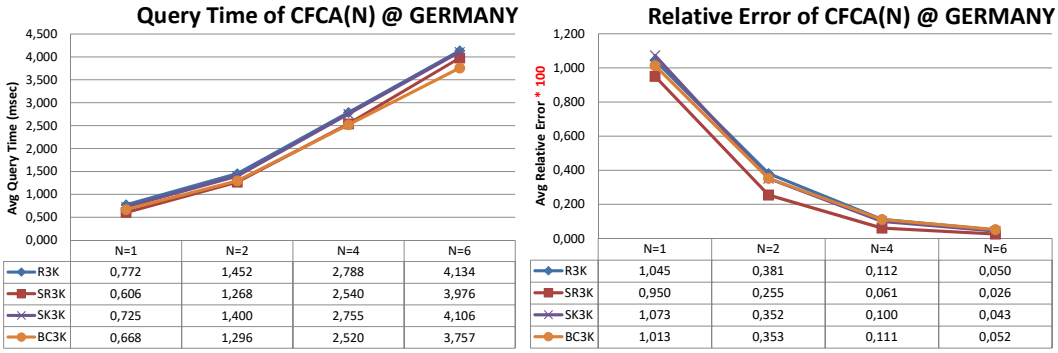
[4] `TDD` is executed here on the original instance, even before the vertex contraction. In [15] it was executed on the contracted graph, hence the slightly smaller execution times of `TDD` in that work. Nevertheless, we believe that this is the appropriate measurement to make for `TDD`, for sake of comparison with other works, and also since the contraction of degree-2 vertices is part of the preprocessing phase.



	R1K	R2K	R3K	R4K
CFCA(1)	1,631	1,054	0,772	0,685
CFCA(2)	3,156	2,004	1,452	1,254
CFCA(4)	6,161	3,860	2,788	2,349
CFCA(6)	9,151	5,718	4,134	3,434

	R1K	R2K	R3K	R4K
CFCA(1)	1,615	1,225	1,045	0,909
CFCA(2)	0,559	0,434	0,381	0,343
CFCA(4)	0,158	0,119	0,112	0,102
CFCA(6)	0,065	0,050	0,050	0,047

Figure 5 Performance of `CFCA(N)` in Germany, for random landmarks and 50,000 *random queries*. In the graph of relative errors, all values are multiplied by 100.



	N=1	N=2	N=4	N=6
R3K	0,772	1,452	2,788	4,134
SR3K	0,606	1,268	2,540	3,976
SK3K	0,725	1,400	2,755	4,106
BC3K	0,668	1,296	2,520	3,757

	N=1	N=2	N=4	N=6
R3K	1,045	0,381	0,112	0,050
SR3K	0,950	0,255	0,061	0,026
SK3K	1,073	0,352	0,100	0,043
BC3K	1,013	0,353	0,111	0,052

Figure 6 Performance of `CFCA(N)` in Germany, for 3K landmarks and 50,000 *random queries*. In the graph of relative errors, all values are multiplied by 100.

really make a difference in the Germany instance. This and further experiments are reported in [16].

3.3 Exploring Outliers in Relative Errors

The purpose of our next experiment was to delve into the details of the relative error of `CFCA(N)`. We study the quantiles of the relative error for serving 50,000 random queries, for BC16K at Berlin, and for SR4K at Germany. Figure 7 presents the results of this experimentation.

It is worth mentioning for Berlin that, with the BC16K-landmark set, we had 99.47% of queries (i.e., only 265 out of the 50K queries exceeded it) with error less than 0.01. Moreover, 97.81% of queries have error less than 0.001. The maximum observed error for SR4K at Berlin was 0.1728.

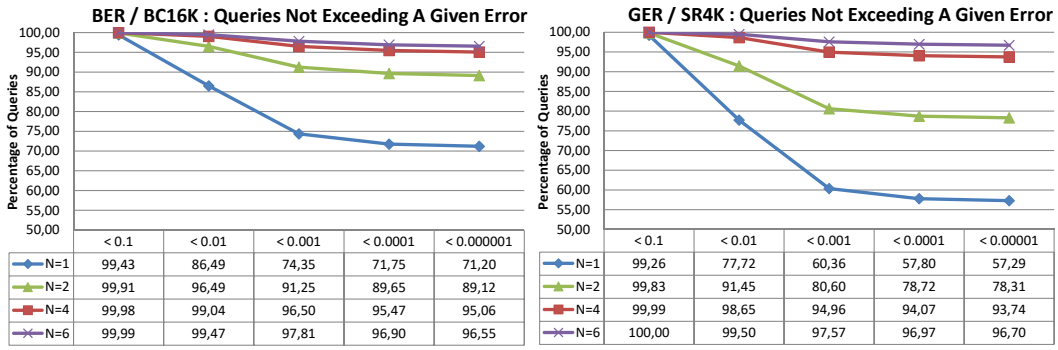
The picture is analogous also for Germany: With the SR4K-landmark set we can have 99.5% of the queries answered with an error less than 0.01, i.e., only 250 of the 50K queries exceeded it. Moreover, 97.57% of them with error less than 0.001. The maximum observed error for SR4K at Germany was 0.079821.

3.4 Comparison with State-of-Art

Table 1 provides a comparison of `CFLAT` with the most prominent oracles and speedup techniques for the two benchmark instances. In particular, we compare the performances of the following algorithms, on the instances of Berlin and Germany:

Table 1 Comparison with State-of-Art; [⋆]: evaluated in this work on exactly the same benchmark instances and for the same sets of 50K **iuar** chosen queries.

	Algorithm		Preprocessing Performance			Query Performance			
	Name [ref.]	Param	Time h:m (#cores)	Work h:m	Space B/node	Path N/Y	Time msec	Relative Error avg	Relative Error max
GER (4,692,091 nodes; 10,805,429 arcs)	TDD [⋆]	–	–	–	–	•	1,190.873	0	0
	inex.TCH [2]	(0.1)	06:18 (8)	50:24	286	○	0.70	0.02	0.10
	inex.TCH	(1.0)			214		0.69	0.27	1.01
	inex.TCH	(2.5)			172		0.72	0.79	2.44
	inex.TCH	(10.0)			113		1.06	3.84	9.75
	KaTCH [⋆]	–	00:05 (6)	00:26	881	○	0.84	0	0
	TDCRP [3]	(1.0)	00:13 (16)	03:28	77	○	1.17	0.68	3.60
	FreeFlow [23]	–	< 00:02 (16)	00:24	n/r	•	0.12	0.14	12.4
	TD-S+4 [23]		< 00:06 (16)	01:34			0.97	0.002	2
	TD-S+9 [23]		< 00:17 (16)	04:23			2.09	0.001	2
	DijFF [⋆]	–	–	–	–	•	905.31	0.134	11.7
	FLAT FLAT [15]	SR2K, N=1	42:42 (6)	256:12	11,625	○	1.275	0.01444	n/r
		SR2K, N=6					9.952	0.00662	
	FLAT	SK2K, N=1	44:06 (6)	264:36			1.269	0.01534	
		SK2K, N=6					9.689	0.00676	
	CFLAT [⋆] CFLAT	SR4K, N=1	28:57 (6)	173:42	7,387	•	0.585	0.0079	0.918
		SR4K, N=6					3.572	0.000177	0.079821
BER (478,089 nodes – 1,126,468 arcs)	TDD [⋆]	–	–	–	–	•	110.02	0	0
	KaTCH [⋆]	–	< 00:01 (6)	< 00:04	851	○	0.339	0	0
	TDCRP [3]	(1.0)	00:02 (16)	00:28	67	○	0.28	1.47	2.69
	FreeFlow [23]	–	< 00:01 (16)	00:07	n/r	•	0.09	0.0012539	15.574
	TD-S [23]						0.23	0.0000153	1.851
	TD-S+A [23]						3.01	0.00000584	1.029
	DijFF [⋆]	–	–	–	–	•	80.32	0.365	21.67
	FLAT [15]	SR2K, N=1	05:12 (6)	31:12	61,198	○	0.081	0.00771	n/r
		SR2K, N=6					0.586	0.00317	
		SK2K, N=1	05:42 (6)	33:12			0.083	0.00781	
		SK2K, N=6					0.616	0.00227	
	CFLAT [⋆]	BC4K, N=1	03:44 (6)	22:23	6,353	•	0.075	0.00521	0.4115
		BC4K, N=6					0.338	0.0002	0.3148
		BC16K, N=1	14:42 (6)	88:12	26,900		0.064	0.0023	0.3855
		BC16K, N=6					0.214	0.00019	0.1728



	< 0.1	< 0.01	< 0.001	< 0.0001	< 0.000001
N=1	99,43	86,49	74,35	71,75	71,20
N=2	99,91	96,49	91,25	89,65	89,12
N=4	99,98	99,04	96,50	95,47	95,06
N=6	99,99	99,47	97,81	96,90	96,55

	< 0.1	< 0.01	< 0.001	< 0.0001	< 0.00001
N=1	99,26	77,72	60,36	57,80	57,29
N=2	99,83	91,45	80,60	78,72	78,31
N=4	99,99	98,65	94,96	94,07	93,74
N=6	100,00	99,50	97,57	96,97	96,70

Figure 7 Tails of the error percentages of `CFCA(N)`, for 50,000 randomly chosen queries in the instance of Berlin with the BC16K landmark set, and for the instance of Germany with the SR4K landmark set.

(1) `TDCRP`, tested on a 16-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64GB of DDR3-1600 RAM, 20 MB of L3 and 256 KB of L2 cache. The reported numbers are from [3];

(2) `FreeFlow`, `TD-S` and `TD-S+A`, tested on a 16-core Intel Xeon E5-1630 v3 clocked at 3.70GHz with 128GB of 2133GHz DDR4 RAM. The reported numbers are from [23];

(3) `inex.TCH`, tested on an 8-Core Intel i7, clocked at 2.67 GHz, with 64 GB DDR4 RAM. The reported numbers are from [3];

(4) an open-source version of `TCH` (`KaTCH`[5]), tested (with compilation parameters -O3 and -DNDEBUG, and its default values) on our machine;

(5) our own implementation of the `FreeFlow` heuristic (called `DijFF`), tested on our machine (it is a static-Dijkstra execution on the Free Flow instance, with no exploitation of any speedup heuristic, and then computation of the time-dependent travel-time along the chosen path); and

(6) `FLAT` and `CFLAT`, which were tested on our machine. The reported numbers for `FLAT` are from [15].

All the reported times are *unscaled* (i.e., as they have been reported) and include both metric-independent and metric-dependent preprocessing of the instances. *Work* is measured as the product of the running time with the number of cores. The "path" column indicates whether the explicit construction of a connecting path is accounted for in the reported query times: ○ is a NO-answer, ● means YES. It is worth noting at this point that, despite the fact that path construction takes a negligible fraction of the execution time in the static case, this is not true for time-dependent instances. This is due to the fact that, as we move backwards from the destination towards the origin, we have to deal with evaluations of functions (rather than just labels). An additional complication is that time is continuous, therefore it is not always clear which is the most appropriate parent to consider. In certain cases one needs to choose more than one parents, in order to avoid cycling. Thus, the path construction is *not* a negligible fraction of the overall effort of a query algorithm. For example, in our case this task (consisting of Steps 2 and 3 in our query algorithm) consumes more than 30% of the overall computational effort (cf. [16]). "n/r" means that a particular value has not been reported.

[5] `https://github.com/GVeitBatz/KaTCH`, with checksum 70b18ad0791a687c554fbfe9039edf79bc3a8ff3.

As is shown in the table, CFLAT dominates the performance of FLAT [15] for both instances. We therefore focus on the comparison of CFLAT with state-of-art speedup heuristics. In particular, we consider the speedup heuristics inex.TCH [2] (only for Germany), TDCRP [3], KaTCH, FreeFlow [23], TD-S [23], and TD-S+A [23]. The algorithms TDD, KaTCH, DijFF and CFLAT, marked in Table 1 with [⋆], were evaluated in the present work, on exactly the same benchmark instances and for the same sets of 50K **iuar** chosen queries. For the other algorithms we report (unscaled) the measurements of the original experimentation by their authors. For the sake of comparison and a posteriori verification, we provide the two random query sets that we have used in http://150.140.143.218:8000/public/. CFLAT is certainly significantly more demanding in preprocessing requirements than the other state-of-art techniques, which is typically anticipated by any landmark-based technique. Nevertheless, it is noted that, if one considers dynamic updates of the instance, e.g. unforeseen road blockages due to unforeseen incidents, CFLAT is remarkably fast in updating the preprocessed information. For example, if one considers 15-minute road blockages, then the procedure for updating the affected landmarks' summaries requires less than 10sec on both instances [16].

Concerning the observed relative errors, first note that KaTCH is an exact heuristic achieving essentially optimal solutions in all cases (although when it is used, it also reports its relative error with TDD; using our implementation of TDD, the reported max relative errors were indeed negligible: $1.4 \cdot 10^{-6}$ for Germany and $2.4 \cdot 10^{-5}$ for Berlin; these errors probably have to do with numerical precision issues). The reported errors for TD-S are noticeable. The average and worst-case errors of CFLAT are higher than those of KaTCH on both instances, but are better (e.g., for SR4K, N=6), almost by an order of magnitude, than those of TD-S for Germany. They are higher than those of TD-S for Berlin[6]. It is mentioned though that the achieved errors of CFLAT are of the same order for both instances, which seems not to be the case for TD-S.

Concerning the (unscaled) query times, CFLAT is the fastest technique for Berlin (e.g., for BC16K and $N = 1$) and the second fastest technique for Germany (e.g., for SR4K and N=1). Of course, absolute running times on different machines are hard to compare. Nevertheless, since all the used machines are essentially of comparable computational capabilities, this makes CFLAT a highly competitive route planning algorithm for time-dependent instances.

Finally, we note that we have also conducted preliminary experimentation on the time-dependent synthetic instance of Central Europe that is typically used for experimentation of the state-of-art techniques. Our findings are quite encouraging: Using only 800 random landmarks, we observed average query times of 15.4msec and average relative error 0.01339. More details about this experiment will appear in the journal version of this paper.

Acknowledgements. The authors wish to thank G. Veit Batz, Julian Dibbelt and Ben Strasser for valuable and fruitful discussions.

References

1 D. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. *Algorithms and Models for the Web-Graph (WAW)*, pages 124–137, 2007.

2 G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, 2013.

[6] The numbers of TD-S for Berlin were taken from the ArXiv report of [23].

3 M. Baum, J. Dibbelt, T. Pajor, and D. Wagner. Dynamic time-dependent route planning in road networks with user preferences. *Experimental Algorithms (SEA)*, LNCS(9685):33–49, 2016.

4 F. Dehne, M. T. Omran, and J. R. Sack. Shortest paths in time-dependent FIFO networks. *Algorithmica*, 62(1–2):416–435, 2012.

5 D. Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, 2011.

6 D. Delling and G. Nannicini. Core routing on dynamic time-dependent road networks. *INFORMS Journal on Computing*, 24(2):187–201, 2012.

7 D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. *Experimental Algorithms (WEA)*, LNCS(4525):52–65, 2007.

8 D. Delling and D. Wagner. Time-dependent route planning. *Robust and Online Large-Scale Optimization*, LNCS(5868):207–230, 2009.

9 U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. A case for time-dependent shortest path computation in spatial networks. *SIGSPATIAL Advances in Geographic Information Systems (GIS)*, pages 474–477, 2010.

10 S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations Research*, 17(3):395–412, 1969.

11 L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, 68(4):1075–1097, 2014.

12 M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, AMS 74:41–72, 2009.

13 KaHIP – Karlsruhe High Quality Partitioning, May 2014.

14 S. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis. Analysis and experimental evaluation of time-dependent distance oracles. *Algorithm Engineering and Experiments (ALENEX)*, SIAM:147–158, 2015.

15 S. Kontogiannis, G. Michalopoulos, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis. Engineering oracles for time-dependent road networks. *Algorithm Engineering and Experiments (ALENEX)*, SIAM:1–14, 2016.

16 S. Kontogiannis, G. Papastavrou, A. Paraskevopoulos, D. Wagner, and C. Zaroliagis. Improved oracles for time-dependent road networks. *CoRR abs/1704.08445 (arxiv:1704.08445)*, 2017.

17 S. Kontogiannis, D. Wagner, and C. Zaroliagis. Hierarchical oracles for time-dependent networks. *Algorithms and Computation (ISAAC)*, LIPICS 64(47):1–13, 2016.

18 S. Kontogiannis and C. Zaroliagis. Distance oracles for time-dependent networks. *Algorithmica*, 74(4):1404–1434, 2016.

19 G. Mali, P. Michail, A. Paraskevopoulos, and C. Zaroliagis. A new dynamic graph structure for large-scale transportation networks. *Algorithms and Complexity (CIAC)*, LNCS(7878):312–323, 2013.

20 G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional A* search on time-dependent road networks. *Networks*, 59:240–251, 2012.

21 M. Omran and J. R. Sack. Improved approximation for time-dependent shortest paths. *Computing and Combinatorics (COCOON)*, LNCS(8591):453–464, 2014.

22 A. Orda and R. Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990. doi:10.1145/79147.214078.

23 B. Strasser. Intriguingly Simple and Efficient Time-Dependent Routing in Road Networks. CoRR abs/1606.06636 (arxiv:1606.06636v2). URL: https://arxiv.org/abs/1606.06636.