Distance Oracles For Time-Dependent Road Networks

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Georgia Papastavrou

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN COMPUTER SCIENCE THEORY

University of Ioannina

February 2017

Examining Committee:

- **Σπύρος Κοντογιάννης**, Επίκουρος Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων (Επιβλέπων)
- **Σταύρος Δ. Νικολόπουλος**, Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων
- **Λουκάς Γεωργιάδης**, Επίκουρος Καθηγητής, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων

Dedication

Το έργο αυτό αφιερώνεται στην οικογένειά μου, που είναι πάντα δίπλα μου, και στον επιβλέποντα της εργασίας, κ. Σπύρο Κοντογιάννη, που με υποστήριξε με κάθε μέσο.

Acknowledgements

Η περάτωση της εργασίας αυτής και η απόκτηση του αντίστοιχου τίτλου δε θα ήταν εφικτή χωρίς την πολύτιμη συμβολή και καθοδήγηση του επιβλέποντος, κ. Σπύρου Κοντογιάννη, ο οποίος υπήρξε αδιάπαυστα υποστηρικτής της προσπάθειάς μου.

Επιπρόσθετα, θα ήθελα ιδιαίτερα να ευχαριστήσω τον συνάδελφο Ανδρέα Παρασκευόπουλο, για την άψογη συνεργασία μας και τον κ. Χρήστο Ζαρολιάγκη για την εμπιστοσύνη του.

Τέλος, ευχαριστώ τα μέλη της τριμελούς επιτροπής, τον κ. Σταύρο Δ. Νικολόπουλο και τον κ. Λουκά Γεωργιάδη, για τις εύστοχες παρατηρήσεις και συμβουλές τους.

Table of Contents

List of Figures

List of Tables

Abstract

Georgia Papastavrou, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, February 2017.
Distance Oracles For Time-Dependent Road Networks.
Advisor: Spyros Kontogiannis, Assistant Professor.

Urban road networks are represented as directed graphs, accompanied by a metric which assigns cost ***functions*** (rather than scalars) to the arcs, e.g. representing time-dependent arc-traversal-times. In this work, we present oracles for providing ***time-dependent*** min-cost route plans, and conduct their experimental evaluation in two real-world data sets of large-scale, in particular the road network for the metropolitan area of Berlin, and the national road network of Germany. Our oracles ***provably*** achieve two unique features: (i) ***subquadratic*** preprocessing time and space; (ii) ***sub-linear*** query time, in either the network size or the actual Dijkstra-Rank of the query at hand.

The first step towards a landmark-based oracle is the selection of a subset of vertices in the graph as landmarks. Then, our oracles are based on precomputing all landmark-to-vertex shortest travel-time functions, for properly selected landmark sets. The core of this preprocessing phase is based on a novel, quite efficient and simple one-to-all approximation method for creating approximations of shortest travel-time functions, called the Trapezoidal approximation method (TRAP). We then propose the `FLAT` oracle and three appropriate query algorithms, to efficiently provide min-cost route plan responses to arbitrary queries. Apart from the purely algorithmic challenges, we deal also with several implementation details concerning the digestion of raw traffic data, and we provide heuristic improvements of both the preprocessing phase and the query algorithms. We exploit parallelism and lossless compression to severely reduce preprocessing space and time requirements. We conduct an extensive, comparative experimental study. Our results are quite encouraging, achieving

remarkable speedups (at least by two orders of magnitude) and quite small approximation guarantees, over the time-dependent variant of Dijkstra's algorithm.

We also implement and experimentally evaluate a novel oracle (HORN), based on a landmark hierarchy. The implementation and experimental evaluation of HORN is based on a ***hierarchy*** of landmarks, from a few "global" landmarks possessing knowledge of the entire network towards (many more) "local" landmarks whose knowledge of the network is restricted to small neighborhoods around them. The advantage of HORN over FLAT is that it achieves query times sublinear, not just in the size of the network, but in the actual Dijkstra rank of the query at hand, be it long-range, mid-range, or short-range, while requiring asymptotically similar preprocessing space and time.

We present a third landmark-based oracle (CFLAT) for providing route plans in time-dependent road networks, which preprocesses time-varying combinatorial structures (collections of time-stamped shortest-path trees). Its main novelty is exactly that it preprocesses only shortest-path trees, ignoring entirely the actual travel-time values, except for assessing the quality of the currently achieved approximation guarantee (a.k.a. stretch) ensured by the preprocessed information. We further exploit the repetitive appearance of only a few patterns in the preprocessed information, in order to avoid duplicate records of the same data. This leads to a significant space reduction, compared with our previous oracles.

For this new kind of time-varying data structure, we also need a novel query algorithm (CFCA) that will exploit it.

As shown by our experimental evaluation, CFLAT achieves a significant improvement in preprocessing time and space requirements, better approximation guarantees and also comparable (if not better) query-response times, in comparison to previous state-of-art landmark-based oracles for time-dependent shortest paths. It also achieves comparable query-time performance and approximation guarantees, on both instances, compared to state-of-art speedup heuristic techniques for time-dependent road networks.

Finally, we implement and experimentally test a dynamic scheme to provide responsiveness to live-traffic reports of incidents with a small timelife (e.g., a temporary blockage of a road segment due to an accident). Our experiments also indicate that traffic information can be updated in seconds.

Εκτεταμενη Περιληψη

Γεωργία Παπασταύρου, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2017.
Χρησμοί Εύρεσης Βέλτιστων Διαδρομών σε Χρονοεξαρτώμενα Οδικά Δίκτυα.
Επιβλέπων: Σπύρος Κοντογιάννης, Επίκουρος Καθηγητής.

Για την αναπαράσταση των αστικών οδικών δικτύων χρησιμοποιούμε κατευθυνόμενα χρονοεξαρτώμενα γραφήματα, σε κάθε ακμή των οποίων αντιστοιχεί μια συνάρτηση η οποία αναθέτει στην ακμή το κόστος της, δηλαδή το χρόνο που χρειάζεται για τη διάσχισή της. Στην παρούσα εργασία, παρουσιάζουμε χρησμούς για την εύρεση χρονοεξαρτώμενων βέλτιστων διαδρομών σε οδικά δίκτυα. Η συντομότερη διαδρομή από έναν κόμβο-αφετηρία προς ένα κόμβο-προορισμό στα δίκτυα αυτά αλλάζει ανάλογα με τη χρονική στιγμή αναχώρησης από την αφετηρία. Αξιολογούμε πειραματικά τους χρησμούς που προτείνουμε σε δύο ρεαλιστικά στιγμιότυπα μεγάλης κλίμακας, συγκεκριμένα χρησιμοποιούμε το αστικό δίκτυο της πόλης του Βερολίνου και το εθνικό οδικό δίκτυο της Γερμανίας.

Οι χρησμοί που προτείνουμε επιτυγχάνουν δύο σημαντικά χαρακτηριστικά: i) το στάδιο της προεπεξεργασίας απαιτεί πολυπλοκότητα χώρου και χρόνου κάτω της τετραγωνικής, ii) ο χρόνος απόκρισης των αλγορίθμων εύρεσης της βέλτιστης διαδρομής για κάθε χρησμό είναι αποδοτικότερος του γραμμικού, τόσο ως προς το μέγεθος του δικτύου, όσο και ως προς το μέγεθος της εκάστοτε διαδρομής.

Το αρχικό συστατικό των χρησμών που μελετούμε είναι η επιλογή ενός υποσυνόλου των κόμβων του γραφήματος, τα οποία ονομάζονται landmarks. Τα σύνολα αυτά των κόμβων επιλέγονται με κατάλληλο τρόπο ώστε να αξιοποιηθούν αποτελεσματικά από τους χρησμούς. Στη συνέχεια, οι χρησμοί βασίζονται στην προεπεξεργασία των συναρτήσεων που αποτιμούν τη συντομότερη διαδρομή από κάθε landmark προς όλους τους πιθανούς προορισμούς του δικτύου. Υπολογίζουμε προσεγγιστικά τις συναρτήσεις αυτές, αφού ο ακριβής υπολογισμός τους είναι απαγορευτικός σε

απαιτήσεις χώρου και χρόνου σε οδικά δίκτυα μεγάλης κλίμακας με εκατομμύρια κόμβους και ακμές. Χρησιμοποιούμε την Τραπεζοειδή προσεγγιστική μέθοδο, την οποία εν συντομία ονομάζουμε TRAP, και που προσεγγιστικά υπολογίζει τις συναρτήσεις απόστασης μεταξύ ενός κόμβου του δικτύου προς όλους τους υπόλοιπους, με εφικτό τρόπο. Έπειτα, η εργασία παρουσιάζει τον πρώτο χρησμό, που ονομάζουμε FLAT oracle, και τρεις κατάλληλους αλγορίθμους, οι οποίοι με βάση τα landmarks και την πληροφορία της προεπεξεργασίας, παρέχουν απόκριση σε αυθαίρετα ερωτήματα βέλτιστης διαδρομής μεταξύ κόμβων του γραφήματος.

Η εργασία δεν αντιμετωπίζει μόνο τις θεωρητικές και αλγοριθμικές προκλήσεις, αλλά παρουσιάζει τις διάφορες λεπτομέρειες υλοποίησης και τις ευριστικές μεθόδους που χρησιμοποιήθηκαν στην πράξη για τη διαχείριση και τη βελτίωση του χώρου και χρόνου της προεπεξεργασίας. Περιλάμβάνεται, επίσης, εκτενής πειραματική αξιολόγηση όλων των μεθόδων και τεχνικών που μελετώνται. Στην πραγματικότητα, τα αποτελέσματα είναι αρκετά ενθαρρυντικά, αφού επιτυγχάνονται τόσο η επιτάχυνση του χρόνου της απόκρισης των αλγορίθμων όσο και η αξιοπιστία των προσεγγιστικών λύσεων, όταν αυτές συγκρίνονται με τον αντίστοιχο αλγόριθμο του Dijkstra σε χρονοεξαρτώμενα γραφήματα.

Επιπρόσθετα, υλοποιούμε και αξιολογούμε πειραματικά ένα νέο χρησμό, το HORN oracle, ο οποίος αξιοποιεί το υποσύνολο των landmarks όταν αυτό δημιουργείται ιεραρχικά. Συγκεκριμένα, η ιεραρχία ξεκινά στο πρώτο επίπεδο με κάποια λίγα (καθολικά) landmarks, που «γνωρίζουν» τις προσεγγιστικές συναρτήσεις απόστασης προς όλο το υπόλοιπο δίκτυο και καταλήγει στο τελευταίο επίπεδο, όπου περισσότερα σε πλήθος (τοπικά) landmarks «γνωρίζουν» κατά προσέγγιση τις αποστάσεις τους από προορισμούς που βρίσκονται σε μια μικρή γειτονιά γύρω τους. Το πλεονέκτημα του HORN έναντι του FLAT είναι το γεγονός ότι πετυχαίνει χρόνους απόκρισης με πολυπλοκότητα καλύτερη της γραμμικής, όχι μόνο ως προς το πλήθος των κόμβων του δικτύου, αλλά ως προς το ακριβές μέγεθος της διαδρομής, είτε το ερώτημα αφορά διαδρομές μεγάλου, μεσαίου ή μικρού μήκους, ενώ απαιτεί ασυμπτωτικά παρόμοια πολυπλοκότητα χώρου και χρόνου προεπεξεργασίας με το FLAT.

Παρουσιάζουμε ένα τρίτο χρησμό, που αξιοποιεί όπως και οι παραπάνω, ένα κατάλληλα επιλεγμένο υποσύνολο κόμβων-landmarks, τον οποίο ονομάζουμε CFLAT oracle. Ο χρησμός αυτός βασίζεται στην προεπεξεργασία χρονοεξαρτώμενων δέντρων συντομότερων διαδρομών, και όχι συναρτήσεων, χαρακτηριστικό που αποτε-

λεί την πρωτοτυπία του χρησμού αυτού.

Για την υλοποίησή του, καταφέρνουμε να μειώσουμε τον απαιτούμενο χώρο της προεπεξεργασίας, με τη χρήση ευριστικών τεχνικών, όπως για παράδειγμα η αναζήτηση και ανακάλυψη επαναλαμβανόμενων μοτίβων στη μορφή των συναρτήσεων που «κρατούν» τα landmarks (τις οποίες όμως δεν αποθηκεύουμε πια, παρά τα αντίστοιχα δέντρα), με τελικό σκοπό την εξοικονόμηση χώρου, αφού η πολυπλοκότητα των συναρτήσεων ή δέντρων μειώνεται.

Για αυτό το νέο τύπο χρησμού, ο οποίος προεπεξεργάζεται χρονο-μεταβαλλόμενες δομές δεδομένων (δέντρα συντομότερων διαδρομών), χρειαζόμαστε ένα κατάλληλο αλγόριθμο απόκρισης, τον CFCA, ο οποίος και αξιοποιεί τον χρησμό για την εύρεση της (προσεγγιστικά) συντομότερης διαδρομής σε οποιοδήποτε ερώτημα αφετηρίας-προορισμού στο γράφημα.

Όπως επιδεικνύει η πειραματική αξιολόγηση που διεξάγουμε στην παρούσα εργασία, ο χρησμός CFLAT επιτυγχάνει σημαντική βελτίωση όσον αφορά στον χώρο και χρόνο της προεπεξεργασίας, ξεκάθαρα καλύτερες προσεγγίσεις και επίσης συγκρίσιμους – αν όχι καλύτερους – χρόνους απόκρισης σε σχέση με αντίστοιχους σύγχρονους χρησμούς βέλτιστων διαδρομών σε χρονοεξαρτώμενα δίκτυα.

Η εργασία ακόμη μελετά την ανταπόκριση των αλγορίθμων σε ζωντανές, συχνά απροσδόκητες, αλλαγές στις κυκλοφοριακές συνθήκες των οδικών δικτύων, οι οποίες συνήθως έχουν μικρή διάρκεια (π.χ. μια προσωρινή διακοπή της κυκλοφορίας σε κάποιον δρόμο λόγω ατυχήματος). Στις περιπτώσεις αυτές, η πληροφορία της προεπεξεργασίας πρέπει να ενημερωθεί ώστε να ενσωματώσει τις απαιτούμενες αλλαγές στις συναρτήσεις απόστασης των landmarks (ή στα αντίστοιχα δέντρα συντομότερων διαδρομών) προς το υπόλοιπο δίκτυο. Στα πειράματα που παρουσιάζει η εργασία φαίνεται ότι οι ενημερώσεις αυτές μπορούν να γίνουν σε μερικά δευτερόλεπτα.

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Computing shortest paths in graphs is a core task in many real-world applications, such as route planning in transportation networks, routing in communication infrastructures, etc. Typically the underlying graph is accompanied with an arc-cost function, assigning a ***fixed*** cost value to every arc, representing average travel-time, distance, fuel consumption, etc. The path of a particular cost is then the aggregation of arc costs along it. The large-scale and real-time response challenges have been addressed in the last 15 years by means of a new algorithmic trend: the provision of ***distance oracles***. That is, succinct data structures encoding shortest path information among a carefully selected subset of pairs of vertices in a graph, which we call ***landmarks***. The encoding is done in such a way that the oracle can efficiently answer shortest path queries for arbitrary origin-destination pairs, exploiting the preprocessed data and/or local shortest path searches. A distance oracle is exact (resp. approximate) if the returned distances by the accompanying query algorithm

are exact (resp. approximate). The quality of an oracle is assessed by its preprocessing space and time requirements, the time-complexity of the query algorithm and the approximation guarantee (stretch).

A bulk of important work is devoted to constructing distance oracles for ***static*** (i.e., ***time-independent***), mostly undirected networks in which the arc-costs are fixed, providing trade-offs between the oracle's space and query time and, in case of approximate oracles, also of the stretch (maximum ratio, over all origin-destination pairs, between the distance returned by the oracle and the actual distance). For an overview of distance oracles for static networks, the reader is referred to [1] and references therein. Considerable experimental work on routing in large-scale road networks has also appeared in recent years, with remarkable achievements that have been demonstrated on continental-size road-network instances. The goal is again to preprocess the distance metric and then propose query algorithms (known as ***speedup techniques*** in this framework) for responding to shortest path queries in time that is several orders of magnitude faster than a conventional Dijkstra run. An excellent overview of this line of research is provided in [2]. Once more, the bulk of the literature concerns static distance metrics, with only a few exceptions (e.g., [3, 4, 5]).

However, in real-world applications the cost of each arc should not be considered as a fixed value, since it undergoes frequent updates. These updates may be instantaneous, unpredictable changes (e.g., due to a sudden change of weather conditions, or a car accident that blocks a road segment or junction), or anticipated updates due to periodic changes of the network characteristics over time. For example, the traversal-time of a road segment may depend on the real-time congestion upon traversal, and thus on the departure time from its tail: In rush hours it is anticipated that it will be much longer than the free-ride traversal-time which is usually valid only for particular departure times (e.g., during the weekend, or at night). Such networks in which the characteristics of the network change in a predictable fashion over time, are called ***time-dependent networks***.

The temporality of the network characteristics is often depicted by some kind of predetermined dependence of the metric on the actual time that each resource is used (e.g., traversal speed in road networks, packet-loss rate in IT networks, arc availability in social networks, etc). Perhaps the most typical application scenario, motivating also our work, is ***route planning in road networks*** where the travel-time for traversing an arc $a = uv$ (modeling a road segment) depends on the temporal traffic conditions

while traversing uv, and thus on the departure-time from its tail u.

In the present work we focus on such networks in which it is the behavior of the arc-cost functions that are described by time-dependent functions, whose exact shape comes from statistical analysis of historical traffic information. For example, the traversal time of a particular road segment may be sampled at particular times during a day from the historical traffic information, say per 5 minutes during rush hours and more rarely for the remaining periods of the day; the corresponding arc-cost function is then considered to be the (continuous) interpolant of all these sample points. So, we assume that the cost variation of each arc a is determined by a ***continuous, piecewise linear (pwl) and periodic function*** $D[a]$ of the time at which a is actually being traversed, as in [6, 7, 8, 9]. Simply taking a snapshot of the entire network (if possible) and solving the corresponding Static Shortest Path problem is clearly not the proper way to provide a route plan in this case. In the following we shall consider as arc-cost functions the traversal-time (or delay) functions when we start traversing them at particular times.

When providing route plans in time-dependent road networks, arc-costs are considered as ***arc-travel-times***, and time-dependent shortest paths as ***minimum-travel-time*** paths. The goal is then to determine the cost (***minimum-travel-time***) of a shortest path from an origin o to a destination d, as a function of the ***departure-time*** t_o from o. Due to the time-dependence of the arc-cost metric, the actual arc-cost value of an arc $a = uv$ is unknown until the exact time $t_u \geq t_o$ at which uv starts being traversed. In a ***time-dependent network model***, every arc uv comes with an arc-traversal-time function $D[uv]$, whereas each path-traversal-time function is simply the ***composition*** of the corresponding arc-traversal-time functions of its constituent arcs.

Thus, to compute a truly shortest path between an origin-vertex and a destination vertex in the network one has to take into account not only the departure time from the origin, but also the consequent departure time of any other arc that is to be used by a shortest path towards the destination. The problem was introduced in [10].

Formally, the ***Time Dependent Shortest Path*** (TDSP) problem concerns computing an $o-d$ path attaining the ***earliest arrival time*** at d, for an arbitrary triple (o, d, t_o) of an origin-destination pair of vertices $(o, d) \in V \times V$ and departure-time $t_o \in \mathbb{R}$ from the origin, in a time-dependent network model $(G = (V, A), (D[a] : \mathbb{R} \to \mathbb{R}_{>0})_{a \in A})$.

The shape of arc-travel-time functions and the waiting policy at vertices may considerably affect the tractability of the problem [9]. Regarding the waiting policy, a

crucial property that makes $TDSP(o, d, t_o)$ tractable (indeed quasilinear) is that each arc obeys the FIFO (a.k.a. non-overtaking) property, according to which the earliest-arrival-time function of an arc uv is an increasing function of its departure time t_u from the tail u, that is 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). Without the FIFO property the problem can become extremely hard [9]. Non-FIFO policies may lead to **NP**−hard cases [11]. On the other hand, in FIFO network models in which all the arc-travel-time functions possess the FIFO property, there is no need for waiting at either the origin or at intermediate nodes of the chosen path. Then, the problem can be solved in polynomial time by a straightforward variant of Dijkstra's algorithm (we call it `TDD`), which relaxes arcs by computing the arc costs "on the fly", when settling their tails [12].

For these reasons, we focus here on instances for which the FIFO property holds, as indeed is the case with most of past and recent works on $TDSP(o, d, t_o)$.

Two variants of the ***time-dependent shortest path*** problem have been considered in the literature: $TDSP(o, d, t_o)$ (resp. $TDSP(o, \star, t_o)$) focuses on the one-to-one (resp. one-to-all) determination of the ***scalar cost*** of a minimum-travel-time path to d (resp. for all d), when departing from the origin o at time t_o. $TDSP(o, d)$ (resp. $TDSP(o, \star)$) focuses on the one-to-one (resp., one-to-all) succinct representation of the time-dependent minimum-travel-time path ***function***(s) $D[o, d]$ from o to d (resp. towards all reachable d), and all departure-times from o (for future instantaneous evaluations).

The FIFO property may seem unreasonable in some application scenarios, e.g., when travellers at the dock of a train station wonder whether to take the very next slow train towards destination, or wait for a subsequent but faster train. However, our motivation in this work stems from ***route planning*** in urban-traffic road networks where the FIFO property seems much more natural, since all cars are assumed to travel according to the same (possibly time-dependent) average speed in each road segment, and overtaking is not considered as an option when choosing a route plan.

The classical shortest-path techniques (Dijkstra and Bellman-Ford) have their time-dependent variants [12, 9], in networks were the FIFO property is preserved on all arcs. However, these are not the choices one should consider as query algorithms, in case of a route planning service that has to reply in real-time to several dozens, or even hundreds, of queries within a large-scale road network, providing the customers

with fast route plans within milliseconds. In fact, apart from the challenge of scale, time-dependence is by itself also a quite important degree of complexity, both in space and in query-time requirements.

1.2 Related Work

In case of huge networks, as is the case for either continental road networks, metropolitan size urban networks, or social networks, it is rather impractical to use Dijkstra or a label-setting algorithm for every individual shortest path query, even in the stricter case of planar embedded graphs. For the time-independent case the issue has been tackled quite successfully both theoretically (using distance oracles) and in practice (using speed-up techniques). The main idea in both cases is to afford a costly preprocessing phase, that is nevertheless polynomial-time tractable, space efficient and amenable to relatively fast updates in case of dynamic changes in the graph, so that in real-time one can support extremely fast (in sub-linear / polylogarithmic / constant time theoretically, within microseconds in practice) arbitrary shortest path queries.

Both the theoretical and the practical approaches precompute distance-related information from / to specific subsets of nodes in the network. This precomputed information is stored and then used either as part of the direct shortest path calculations between arbitrary pairs of vertices, or in order to provide good lower bounds that are used to direct the search of a shortest path in a Dijkstra-like query algorithm. When applied to time-dependent instances, rather than storing shortest-path distances, one has to keep in memory the earliest-arrival-time / latest-departure-time functions from / to these particular nodes.

Until recently, most of the previous work on the time-dependent shortest path problem concentrated on computing an optimal origin-destination path providing the earliest-arrival time at destination when departing at a ***given*** time from the origin, and neglected the computational complexity of providing succinct representations of the entire earliest-arrival-time ***functions***, for ***all*** departure-times from the origin. Such representations, apart from allowing rapid answers to several queries for selected origin-destination pairs but for varying departure times, would also be valuable for the construction of ***distance summaries*** (a.k.a. ***route planning maps***, or ***search profiles***) from central vertices (e.g., ***landmarks*** or ***hubs***) towards other vertices in the network,

providing a crucial ingredient for the construction of distance oracles to support real-time responses to arbitrary queries $(o, d, t_o) \in V \times V \times \mathbb{R}$.

The complexity of succinctly representing earliest-arrival-time functions was first questioned in [13, 14, 6], but was solved only recently by a seminal work [8] which, for FIFO-abiding pwl arc-travel-time functions, showed that the problem of succinctly representing such a function for a ***single origin-destination pair*** has space-complexity $(1+K)\cdot n^{\Theta(\log n)}$, where n is the number of vertices and K is the total number of breakpoints (or legs) of all the arc-travel-time functions. Polynomial-time algorithms for constructing ***point-to-point*** $(1+\varepsilon)$-approximate distance functions are provided in [7, 8], delivering point-to-point travel-time values at most $1+\varepsilon$ times the true values. Such approximate distance functions possess ***succinct representations***, since they require only $\mathcal{O}(1+K)$ breakpoints per origin-destination pair. It is also easy to verify that K could be substituted by the number K^* of ***concavity-spoiling*** breakpoints of the arc-travel-time functions (i.e., breakpoints at which the arc-travel-time slopes increase).

Providing distance oracles for time-dependent networks with ***provably*** good approximation guarantees, small preprocessing-space complexity and sublinear query time complexity, has only been recently investigated in [15, 16]. In particular, the ***first*** approximate distance oracle for sparse directed graphs with time-dependent arc-travel-times was presented in [16], providing $(1+\sigma)-$approximate travel-times in query-time that is ***sublinear*** in the network size, and preprocessing time and space that are ***subquadratic*** in the network size, when the total number of concavity-spoiling breakpoints in the instance is sufficiently small, e.g. when $K^* \in \mathcal{O}(\text{polylog}(n))$. The oracle uses a novel ***one-to-all*** method (called ***Bisection*** – `BIS`) to produce $(1+\varepsilon)-$approximate landmark-to-vertex travel-time summaries, for a randomly selected landmark set. It also guarantees either constant approximation ratio (a.k.a ***stretch***) via the `FCA` query algorithm, or stretch at most $1+\sigma = 1+\varepsilon\frac{(1+\varepsilon/\psi)^{r+1}}{(1+\varepsilon/\psi)^{r+1}-1}$ via the `RQA` query algorithm, where ψ is a fixed constant depending on the characteristics of the arc-travel-time functions but is independent of the network size, and $r \in \mathcal{O}(1)$ is the recursion depth of `RQA`. In [15], another oracle is proposed, providing both constant and $(1+\sigma)-$approximate travel-times in query-time that is ***sublinear*** in the network size, and preprocessing time and space that are ***subquadratic*** in the network size, independently of the amount of disconcavity K^* in the network instance at hand. This is achieved by combining `BIS` with another ***one-to-all*** method (called

Trapezoidal – TRAP) to produce $(1+\varepsilon)-$approximate landmark-to-vertex travel-time summaries.

A few time-dependent variants of well-known speedup techniques for road networks have also appeared in the literature (e.g., [3, 4, 5]). All of them were experimentally evaluated on synthetic time-dependent instances of the European and German road networks, with impressive performances. For example, in [3] methods are provided that respond to arbitrary queries of the German road network (4.7 million vertices and 10.8 million arcs) in less than $1.5ms$ and preprocessing space requirements of less than 1GB. A point-to-point travel-time summary (a.k.a. search profile) can also be constructed in less than $40ms$, when the departure times interval is a single day. For point-to-point approximate travel-time summaries, with experimentally observed stretch at most 1%, the construction time is less than $3.2ms$. Their approach is based on the so-called ***time-dependent Contraction Hierarchies*** [17], along with several heuristic improvements both on the preprocessing step and on the query method.

1.3 Objectives and Contribution

The main challenge considered in this work is to provide TD-oracles that achieve: (i) subquadratic preprocessing requirements, for succinctly representing travel-time ***functions***, (ii) query-times sublinear, not just in the network size n, but in the number $\Gamma[o,d](t_o)$ (a.k.a. ***Dijkstra-Rank***) of settled vertices when executing $\texttt{TDD}(o,\star,t_o)$ until d is settled, and (iii) small (e.g., close to 1) approximation guarantee (stretch factor).

We engineer and experimentally evaluate three oracles for time-dependent road networks on two real instances corresponding to qualitatively different cases. The first one corresponds to a typical weekday of the metropolitan road network of Berlin (about $4.7K$ vertices and $1.13M$ arcs), while the second one corresponds to a typical weekday of the German road network (about $4.6M$ vertices and $11.18M$ arcs).

Our main contributions are summarized as follows: We propose three time-dependent distance oracles, whose preprocessing phase for computing landmark-to-vertex approximate travel-time summaries is based on a new approximation technique [15], the ***trapezoidal*** (TRAP) method. We speedup the preprocessing phase of computing approximate travel-time functions, from properly selected landmarks towards all

reachable destinations, by exploiting the inherent parallelism of the entire process. We significantly reduce the required preprocessing space, by applying advanced lossless compression techniques.

Our experimentation is based on several carefully chosen landmark sets and moreover on several ***refinements*** and ***hybrid combinations*** of them. We deal with the challenge of large-scale and thus, we present several heuristic improvements towards reducing the preprocessing space and time of our oracles. Based on TRAP, we propose the FLAT oracle. At a high level, our approach resembles the typical ones used in ***static*** and ***undirected*** graphs (e.g., [18, 19, 20]): Distance summaries from selected landmarks are precomputed and stored so as to support fast responses to arbitrary real-time queries by growing small distance balls around the origin and the destination, and then closing the gap between the prefix subpath from the origin and the suffix subpath towards the destination. However, it is not at all straightforward how this generic approach can be extended to ***time-dependent*** and ***directed*** graphs, since one is confronted with two highly non-trivial challenges: (i) handling directedness, and (ii) dealing with time-dependence, i.e., deciding the arrival-times to grow balls around vertices in the vicinity of the destination, because we simply do ***not*** know the earliest-arrival-time at destination – actually, this is what the original query to the oracle asks for. A novelty of our query algorithms, contrary to other approaches, is exactly that we achieve the approximation guarantees by growing balls only from vertices around the origin. Managing this was a necessity for our analysis since growing balls around vertices in the vicinity of the destination at the ***right*** arrival-time is essentially not an option. We propose three query algorithms for the FLAT oracle, called FCA, RQA and FCA$^+$.

Apart from FLAT we implement and experimentally evaluate the novel HORN oracle, which is based on a ***hierarchy*** of landmarks, from a few "global" landmarks possessing knowledge of the entire network towards (many more) "local" landmarks whose knowledge of the network is restricted to small neighborhoods around them. As was proved in [15], the advantage of HORN over FLAT is that it achieves query times ***sublinear***, not just in the size of the network, but in the actual Dijkstra rank of the query at hand, be it long-range, mid-range, or short-range, while requiring asymptotically similar preprocessing space and time. Our experiments on the (harder) Berlin instance indeed confirm the improved stretch factors, but also better speedups due to sophisticated early-stopping criteria, compared to the experimentation on FLAT for the

same subsets of "global" landmarks. For the HORN oracle, we propose the HQA query algorithm.

We propose a new approximation method, CTRAP. Its novelty is that, rather than approximating minimum travel-time functions, it samples and stores only the combinatorial structures (shortest-path trees) of the solutions at the sampled departure-times from carefully selected landmark vertices. We then proceed with the preprocessing phase of our new oracle, CFLAT, which is based on CTRAP. It guarantees a significant improvement in the space requirements, and it is much faster than the preprocessing of FLAT. We propose CFCA(N) (Section 3.3.2), the query algorithm of CFLAT, which, for an arbitrary query (o, d, t_o), grows a time-dependent Dijkstra (TDD) ball from o at time t_o until the N closest landmarks are settled. Then, it proceeds with a second step of marking 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 [21], but the choice of the relevant arcs is doomed to be done "on the fly", since this information is also time-dependent. A final step keeps growing the initial TDD ball within the subgraph of marked arcs, returning the minimum-travel-time od-path in this subgraph.

An experimental study demonstrates the excellent performance of our oracles in practice, achieving considerable memory savings and query-times about three orders of magnitude faster than the Time-Dependent Dijkstra TDD.

Finally, we implement and experimentally test a dynamic scheme to provide responsiveness to live-traffic reports of incidents with a small timelife (e.g., a temporary blockage of a road segment due to an accident). Our experiments also indicate that traffic information can be updated in seconds.

1.4 Structure of Thesis

The thesis is organized as follows:

Section 2 prepares the reader for studying the TD-oracles that we propose. It provides the necessary theoretical background, that is notation, assumptions, analysis of the importance of the FIFO property in time-dependent networks, description of landmarks selection and the preprocessing phase and the presentation of the ***trapezoidal method*** that we propose for approximating travel-time functions between landmarks

and destinations in the network, which we call TRAP. The main study of the thesis concerns sections 3 and 4, where the engineering. heuristic improvements and the experimental evaluation of the proposed oracles are described. In particular, Section 3 presents in detail the implementation of all three oracles, FLAT, HORN and CFLAT. It describes the appropriate query algorithms for each oracle, which provide the route responses, and presents in detail all the heuristic improvements we made towards the reduction of the preprocessing space and time. Section 4 is about the extensive experimentation that we performed for our oracles. The experimental effort is a strong component of our work, which proves in practice the performance of our oracles. Finally, in section 5 we summarize our results and we compare our work with the most successful state-of-the-art methods.

Chapter 2

Theoretical Background

2.1 Preliminaries and Notation

Consider a directed graph $G = (V, A)$, with nonnegative, continuous, piecewise linear (pwl) ***arc-delay*** functions $\forall a \in A, \overrightarrow{D}[a] : \mathbb{R} \to \mathbb{R}_{\geq 0}$ providing the arrival time at the destination $head[a]$ as a function of the departure time from the origin $tail[a]$. Such a function could for example be the interpolant of average arc-delays for particular departure times from a given time period $\Pi = [0, T]$, such that $\forall k \in \mathbb{Z}, \forall t_u \in \Pi, \forall a \in A, \overrightarrow{D}[a](t_u + k \cdot T) = \overrightarrow{D}[a](t_u)$. Since is periodic, continuous and pwl, it can be represented succinctly by a sequence of K_a breakpoints (i.e., pairs of departure-times and

arc-cost values) defining. An example of such an arc-delay function is the following:

$$\forall t_u \in \mathbb{R},\ \overrightarrow{D}[uv](t_u) \ = \ \begin{cases} \frac{4}{3}t_u + 1, & 0 \leq t_u \mod T \leq 3 \\ 5, & 3 \leq t_u \mod T \leq 5 \\ 2t_u - 5, & 5 \leq t_u \mod T \leq 7 \\ -\frac{8}{13}t_u + \frac{173}{13}, & 7 \leq t_u \mod T \leq 20 \\ 1, & 20 \leq t_u \mod T \leq 24 \end{cases}$$

For notational reasons we assume that $\forall t_u \in \Pi, \forall u \in V, \overrightarrow{D}[uu](t_u) = 0$ and $\forall uv \notin A \Rightarrow \overrightarrow{D}[uv](t_u) = +\infty$. Moreover, rather than defining the arc-delay functions as functions of ***departure-time from the tail***, we may also prefer to express them as functions of ***arrival-times at the heads***. We use the notation $\overleftarrow{D}[uv] : \mathbb{R} \to \mathbb{R}_{\geq 0}$ for these ***reverse arc-delay*** functions. For example, the reverse arc-delay function corresponding to the forward arc-delay described above is the following:

$$\forall t_v \in \mathbb{R},\ \overleftarrow{D}[uv](t_v) \ = \ \begin{cases} \frac{4}{7}t_v + \frac{3}{7}, & 1 \leq t_v \mod T \leq 8 \\ 5, & 8 \leq t_v \mod T \leq 10 \\ \frac{2}{3}t_v - \frac{5}{3}, & 10 \leq t_v \mod T \leq 16 \\ -\frac{8}{5}t_v + \frac{173}{5}, & 16 \leq t_v \mod T \leq 21 \\ 1, & 21 \leq t_v \mod T \leq 24 \vee 0 \leq t_v \mod T \leq 1 \end{cases}$$

Note 2.1. It is mentioned that for the reverse expression of the arc-delay function to exist, it must be the case that the original (forward) arc-delay does not have any leg of slope less or equal to -1. In particular, when this is the case, we can invert the (monotone in this case) arrival-time function $t_v = Arr[uv](t_u) = t_u + \overrightarrow{D}[uv](t_u)$ to get $Dep[uv] = (Arr[uv])^{-1}$ and then compute $\overleftarrow{D}[uv](t_v) = t_v - Dep[uv](t_v) = Arr[uv](t_u) - t_u = \overrightarrow{D}[uv](t_u)$. As we shall explain later, we indeed demand that all the slopes in any forward arc-delay function have value strictly greater than this value, and this is not only for the computation of the reverse arc delays.

Analogously, $\overleftarrow{G} = (V, \overleftarrow{A})$ where $\overleftarrow{A} = \{vu \in V \times V : uv \in A\}$ is the graph produced by G if we reverse the directions of all the arcs in it.

For an arbitrary origin-destination pair of vertices, $(o, d) \in V \times V$, let $\mathcal{P}_{o,d}(G)$ be the set of all (directed) walks from o to d in G, while $\mathcal{P}(G) = \bigcup_{(o,d)\in V\times V} \mathcal{P}_{o,d}(G)$. For arbitrary vertices $u, v, z \in V$ and any walks $p \in \mathcal{P}_{u,v}$ and $q \in \mathcal{P}_{v,z}$, $p \oplus q \in \mathcal{P}_{u,z}$ is

the walk resulting as the concatenation of p and q at vertex v. Any walk $p \in \mathcal{P}(G)$ that does not repeat any vertex is a (sometimes redundantly called simple) ***path***. For sake of simplicity, we shall skip reference to the graph, when this is clear from the context. Any particular walk will mostly be considered as an ***ordered set*** of arcs such that for any pair of consecutive arcs, the head of the first arc is identical to the tail of the second arc. Occassionally we may want to declare a subwalk of p from (the first appearance in p of) a vertex $x \in V$ to (the fist appearance in p of) a vertex $y \in V$. This subwalk will be denoted by $p_{x \rightsquigarrow y}$.

For a walk (path) $p = \langle a_1, \cdots a_k \rangle \in \mathcal{P}_{o,d}$ and $\forall 1 \leq i \leq j \leq k$, let $p_{i,j}$ be the subwalk (subpath) of p starting with the i^{th} arc a_i and ending with the j^{th} arc a_j in the order. We define the ***walk/path-delay*** function of p recursively as a function of the departure time t_o from its own origin $tail(p) = tail(a_1)$, as follows:

$$\begin{array}{rl} \forall t_o \in \mathbb{R}, \forall 1 \leq i \leq k, & \overrightarrow{D}[p_{i,i}](t_o) = \overrightarrow{D}[a_i](t_o) \\ \forall t_o \in \mathbb{R}, \forall 1 \leq i < j \leq k, & \overrightarrow{D}[p_{1,j}](t_o) = \overrightarrow{D}[p_{1,i}](t_o) + \overrightarrow{D}[p_{i+1,j}]\left(t_o + \overrightarrow{D}[p_{1,i}](t_o)\right) \end{array} \tag{2.1}$$

We may also express a similar recursive definition of the reverse-path-delays:

$$\begin{array}{rl} \forall t_d \in \mathbb{R}, \forall 1 \leq i \leq k, & \overleftarrow{D}[p_{i,i}](t_d) = \overleftarrow{D}[a_i](t_d) \\ \forall t_d \in \mathbb{R}, \forall 1 \leq i < j \leq k, & \overleftarrow{D}[p_{i,k}](t_d) = \overleftarrow{D}[p_{j,k}](t_d) + \overleftarrow{D}[p_{i,j-1}]\left(t_d - \overleftarrow{D}[p_{j,k}](t_d)\right) \end{array} \tag{2.2}$$

Similarly, we define the ***arrival-time*** function of p at its end-vertex $head(p) = head(a_k)$, as a function of the departure-time $t_o \in \mathbb{R}$ from its start-vertex $tail(p) = tail(a_1)$:

$$\forall t_o \in \mathbb{R}, Arr[p](t_o) = t_o + \overrightarrow{D}[p](t_o) \tag{2.3}$$

It is easily seen that the path-arrival-time functions are indeed compositions of the corresponding arc-arrival-time functions of the arcs comprising them:

$$\begin{aligned} Arr[p_{1,k}](t_o) &= t_o + \overrightarrow{D}[p_{1,k}](t_o) = t_o + \overrightarrow{D}[p_{1,1}](t_o) + \overrightarrow{D}[p_{2,k}](t_o + \overrightarrow{D}[p_{1,1}](t_o)) \\ &= Arr[p_{2,k}]\left(Arr[p_{1,1}](t_o)\right) = \left(Arr[p_{2,k}] \circ Arr[p_{1,1}]\right)(t_o) = \cdots \\ &= \left(Arr[a_k] \circ \cdots \circ Arr[a_1]\right)(t_o) \end{aligned} \tag{2.4}$$

Analogously, the path-departure-time function of p from $tail(p) = tail(a_1)$, given the arrival-time $t_d \in \mathbb{R}$ at $head(p) = head(a_k)$, is defined as follows:

$$Dep[p](t_d) = t_d - \overleftarrow{D}[p](t_d) \tag{2.5}$$

Again, the path-departure-time functions are compositions of the corresponding arc-departure functions of the arcs comprising them:

$$\begin{aligned} Dep[p_{1,k}](t_d) &= t_d - \overleftarrow{D}[p_{1,k}](t_d) = t_d - \overleftarrow{D}[p_{k,k}](t_d) - \overleftarrow{D}[p_{1,k-1}]\left(t_d - \overleftarrow{D}[p_{k,k}](t_d)\right) \\ &= Dep[p_{1,k-1}]\left(Dep[p_{k,k}](t_d)\right) = \left(Dep[p_{1,k-1}] \circ Dep[p_{k,k}]\right)(t_d) = \cdots \\ &= \left(Dep[a_1] \circ \cdots \circ Dep[a_k]\right)(t_d) \end{aligned} \tag{2.6}$$

For any pair of vertices $(o, d) \in V \times V$, the ***earliest-arrival-time*** function from o to d is defined as follows:

$$\forall t_o \in \mathbb{R},\ Arr[o,d](t_o) = \min_{p \in \mathcal{P}_{o,d}} \{Arr[p](t_o)\} \tag{2.7}$$

The ***shortest-travel-time*** function from o to d is $D[o,d](t_o) = Arr[o,d](t_o) - t_o$. Finally, $SP[o,d](t_o)$ (resp. $ASP[o,d](t_o)$) is the set of shortest (resp., with stretch-factor at most $(1+\varepsilon)$) $od-$paths for a given departure-time t_o.

For any arc $a = uv \in A$ and any departure-times subinterval $[t_s, t_f) \subseteq [0, T)$, we consider the free-flow and maximally-congested travel-times for this arc, defined as follows:

- ***Free-flow*** arc-travel-time:

$$D_f[uv](t_s, t_f) := \min_{t_u \in [t_s, t_f)} D[uv](t_u)\,.$$

- ***Maximally-congested*** arc-travel-time:

$$D_c[uv](t_s, t_f) := \max_{t_u \in [t_s, t_f)} D[uv](t_u)\,.$$

We also denote D_c[uv] := D_c[uv](0,T) and D_f[uv] := D_f[uv](0,T). When $[t_s, t_f) = [0, T)$, we refer to the (static) ***free-flow*** and ***full-congestion*** travel-time metrics D_f and D_c, respectively. These definitions also extend naturally to path-travel-times and shortest-travel-times between arbitrary origin-destination pairs of vertices.

In case of huge networks, as is the case for either continental road networks, metropolitan-size urban networks, or social networks, it is rather impractical to use Dijkstra or a label-setting algorithm for every individual shortest path query, even in the stricter case of planar embedded graphs. For the time-independent case the issue has been tackled quite successfully both theoretically (using distance oracles) and in practice (using speed-up techniques). The main idea in both cases is to afford a costly

preprocessing phase, that is nevertheless polynomial-time tractable, space efficient and amenable to relatively fast updates in case of dynamic changes in the graph, so that in real-time one can support extremely fast (in sub-linear / polylogarithmic / constant time theoretically, within microseconds in practice) arbitrary shortest path queries.

Both the theoretical and the practical approaches precompute distance-related information from / to specific subsets of nodes in the network. This precomputed information is stored and then used either as part of the direct shortest path calculations between arbitrary pairs of vertices, or in order to provide good lower bounds that are used to direct the search of a shortest path in a Dijkstra-like query algorithm. When applied to time-dependent instances, one has to keep in memory the shortest travel-time function from/to particular nodes.

The succinct representation of $\overrightarrow{D}[a]$ is given by a collection (ordered list, by increasing time values) of triples:

$$\langle (\overrightarrow{\lambda}_i^a, \overrightarrow{\mu}_i^a, \overrightarrow{t}_i^a) : i \in \{1, \ldots, K_a\} \rangle$$

where the linear function describing the $i-$th leg of $\overrightarrow{D}[a]$ is $\overrightarrow{\lambda}_i^a \cdot t + \overrightarrow{\mu}_i^a$ and its valid interval is $[\overrightarrow{t}_{i-1}^a, \overrightarrow{t}_i^a]$ (we assume that $\overrightarrow{t}_0^a = 0$ and $\overrightarrow{t}_{K_a}^a = T$).

Note 2.2. For simplicity, we shall for now on refer to any forward travel-time function by the term $D[u, v](t_u)$, $\forall$ u, v $\in$ V.

As already mentioned, when the (strict) FIFO property holds, ***subpath optimality*** holds also in time-dependent instances. Simple variants of Dijkstra indeed work also for the computation of shortest $od-$paths and earliest-arrival-time values (for given departure time from origin) in any time-dependent network possessing the FIFO property [12]. We denote such a time-dependent variant by `TDD`. To avoid tricky situations in which the algorithm (even for static networks) might fail, we suppose that all the arc-delay functions are always ***non-negative***. Put it differently, we consider as the actual arc-delay to be the maximum of zero and the declared arc-delay function, for any departure time from the tail.

During the execution of `TDD` in a FIFO network with (non-negative) arc-delay functions, the delay value of every arc has to be estimated upon its (unique) relaxation, when its head is settled. When referring to the description ($\lambda-$ and $\mu-$ values) of an arc-delay function for the arc $a = uv$ that is currently being relaxed for a given departure time $t_u = Arr[o, u](t_o)$, the arc-delay evaluation operation is not constant

anymore, but costs either $\mathcal{O}(\log(K_a))$ (e.g., by maintaining a binary search tree of breakpoints) or even $\mathcal{O}(\log(\log(K_a)))$ if one employs more advanced data structures (e.g., fast tries of breakpoints) in order to determine the appropriate leg of the (pwl) arc-delay function $D[a]$ which is appropriate for t_u. K_a is the space-complexity (i.e., the number of breakpoints) of $D[a]$. Since every arc is relaxed at most once, in overall TDD will have time-complexity $\mathcal{O}(n\log(n) + m \cdot \log(\log(K_{\max})))$ to solve TDSP, where $K_{\max} = \max_{a\in A} K_a$.

A pair of continuous, pwl, periodic functions $\overline{D}[o,d]$ and $\underline{D}[o,d]$), with a (hopefully) small number of breakpoints, are $(1+\varepsilon)$-***upper-approximation*** and $(1+\varepsilon)$-***lower-approximation*** of $D[o,d]$, if $\forall t_o \geq 0$, $\frac{D[o,d](t_o)}{1+\varepsilon} \leq \underline{D}[o,d](t_o) \leq D[o,d](t_o) \leq \overline{D}[o,d](t_o) \leq (1+\varepsilon)\cdot D[o,d](t_o)$.

TDD Balls. For a point $(o,t_o) \in V \times [0,T)$ and $\beta \in \mathbb{N}$, let $B[o](t_o;\beta)$ be the set of the first β vertices settled by TDD, when growing a ball from (o,t_o). Analogously, $\underline{B}[o](\beta)$ and $\overline{B}[o](\beta)$ are the corresponding sets under the free-flow and fully-congested metrics $D_f f$ and $D_c g$, respectively. When we say that we ***"grow a TDD ball from*** (o,t_o)***"***, we refer to the execution of TDD from $o \in V$ at departure-time $t_o \in [0,T)$ for solving $TDSP(o,\star,t_o)$ (resp. $TDSP(o,d,t_o)$, for a specific destination d). Such a call, denoted as $\texttt{TDD}(o,\star,t_o)$ (resp. $\texttt{TDD}(o,d,t_o)$), takes time $\mathcal{O}(m + n\log(n)[1+\log\log(1+K_{\max})]) = \mathcal{O}(n\log(n)\log\log(K_{\max})])$, using predecessor search for evaluating continuous pwl functions [16]. The ***Dijkstra-Rank***, $\Gamma[o,d](t_o)$ of (o,d,t_o), is the number of settled vertices up to d, when executing $\texttt{TDD}(o,d,t_o)$.

2.2 FIFO Property in Time Dependent Networks

A fundamental property of time-dependent networks is the ***FIFO*** (a.k.a. ***non-overtaking***) property [9, 12, 8], which states the following:

$$\forall t_u, t'_u \in \mathbb{R}, \forall uv \in A, t_u > t'_u \Rightarrow Arr[uv](t_u) \geq Arr[uv](t'_u) \tag{2.8}$$

That is, all the arc-arrival-time functions in the network are non-decreasing. The following proposition is a characterization of the FIFO property for networks with continuous arc-delay functions:

Proposition 2.2.1. [*22*] ***Assume a graph*** $G = (V,A)$ ***with continuous arc-delay func-***

tions, satisfying the (strict) FIFO property. Then any arc-delay function must have left and right derivatives with values at least (greater than) -1.

It is also easy to verify that the FIFO property, only assumed for arc-arrival-time functions, also holds for arbitrary path-arrival-time functions, and earliest-arrival-time functions in the graph:

Proposition 2.2.2. [22] ***Assume a graph*** $G = (V, A)$ ***with continuous arc-delay functions, satisfying the FIFO property. Then, for any path*** $p = \langle a_1, \ldots, a_k \rangle \in \mathcal{P}(G)$ ***it holds that:***

$$\forall t_1 \in \mathbb{R}, \forall \delta > 0,\ Arr[p](t_1) \leq Arr[p](t_1 + \delta)$$

In case of strict FIFO property, the inequality is also strict. FIFO property holds also for every earliest-arrival-time function in G.

When moving from an origin to a destination in a time-dependent network, a traveler may possibly have the option to wait at a node for certain amounts of time, prior to traversing an arc emanating from it. We consider the following cases of waiting policies (see also [9]).

Unrestricted Waiting (UW) A traveler may wait at any node for an arbitrary amount of time, prior to traversing an arc emanating from it.

Forbidden Intermediate Waiting (FIW) A traveler may wait only at the origin, for an arbitrary amount of time, prior to starting the journey towards the destination (without any other waiting at a node).

Forbidden Waiting (FW) No waiting is allowed, at any node in the network.

The $TDSP(o, d)$ problem was proved to be **NP**$-$hard, if the FW-policy is adopted and arc-delays are allowed ***not*** to possess the FIFO property. It may even be the case that there is no optimal-waiting (i.e., one that minimizes the earliest-arrival-time at the destination) at a node [9]. On the other hand, it is well known [12] that $TDSP(o, d)$ is polynomial-time solvable when the UW-policy is adopted and the optimal-waiting time always exists for every node, independently of the shape of the arc-delay functions. Indeed, such a scenario is also known [9] to be equivalent to an appropriate FIFO network with the FW-policy.

For instances in which the FIFO property holds, the crucial property of subpath optimality holds:

Proposition 2.2.3. [22] ***Assume a graph with arc-delays satisfying the strict FIFO property. Then, for all vertices*** $u, v \in V$, ***any departure-time*** $t_u \in \mathbb{R}$ ***from*** u, ***and any optimal path***

$$p^* \in \arg\min_{p \in \mathcal{P}_{u,v}} \{Arr[p](t_u)\}$$

it holds that every subpath $q^* \in \mathcal{P}_{x,y}$ ***of*** p^* ***is a shortest path between its endpoints*** x, y ***for departure time from*** x ***equal to*** $t_x^* = Arr[p^*_{u \rightsquigarrow x}](t_u)$.

Therefore, both Dijkstra's label setting algorithm `TDD` and label-correcting algorithms for shortest $uv-$path computations in time-independent graphs, also work in time-dependent strictly FIFO networks, under the usual conventions that we consider for the static instances (positivity of arc-delays for the label setting, and inexistence of negative-delay cycles for the label correcting approaches).

2.3 Assumptions on the arc-cost metric.

The directedness and time-dependence of the TD-instance imply an asymmetric arc-cost metric, which also evolves with time. To achieve a smooth transition from static and undirected graphs towards time-dependent and directed graphs, we need a quantification of the degrees of asymmetry and evolution of our metric over time. These are captured via a set of parameters depicting the steepness of the minimum-travel-time functions, the ratio of minimum-travel-times in opposite directions, and the relation between graph expansion and travel-times. Next, we mention some assumptions on the values of these parameters, which seem quite natural for our main application scenario (route planning in road networks). The reader is refered to [15], where all assumptions were originally presented, exactly stated and validated on real-world road networks.

Assumption 2.3.1 (Bounded Travel-Time Slopes). **[23]** ***All the minimum-travel-time slopes are bounded in a given interval*** $[-\Lambda_{\min}, \Lambda_{\max}]$, ***for given constants*** $\Lambda_{\min} \in [0, 1)$ ***and*** $\Lambda_{\max} \geq 0$.

Assumption 2.3.2 (Bounded Opposite Trips). **[23]** ***The ratio of minimum-travel-times in opposite directions between two vertices, for any specific departure-time but not necessarily via the same path, is upper bounded by a given constant*** $\zeta \geq 1$.

Assumption 2.3.3 (Growth of Free-Flow Dijkstra Balls). **[15]** $\forall F \in [n]$, ***the free-flow ball*** $\underline{B}[v;F]$ ***blows-up by at most a polylogarithmic factor, when expanding its (free-flow) radius up to the value of the full-congestion radius within*** $\underline{B}[v;F]$.

Finally, we need to quantify the correlation between the arc-cost metric and the Dijkstra-Rank metric induced by it. For this reason, inspired by the notion of the doubling dimension (e.g., [24] and references therein), we consider some ***scalar*** $\lambda \geq 1$ and functions $f, g : \mathbb{N} \mapsto [1, \infty)$, such that the following hold: $\forall (o, d, t_o) \in V \times V \times [0, T)$, (i) $\Gamma[o, d](t_o) \leq f(n) \cdot (D[o, d](t_o))^{\lambda}$, and (ii) $D[o, d](t_o) \leq g(n) \cdot (\Gamma[o, d](t_o))^{1/\lambda}$. This property trivially holds, e.g., for $\lambda = 1$, $f(n) = n$, and $g(n) = \max_{a \in A} \{\overline{D}[a]\}$. Of course, our interest is for the smallest possible values of λ and at the same time the slowest-growing functions $f(n), g(n)$. Our last assumption quantifies the boundedness of this correlation by restricting λ, $f(n)$ and $g(n)$.

Assumption 2.3.4. **[15]** ***There exist*** $\lambda \in o\left(\frac{\log(n)}{\log\log(n)}\right)$ ***and*** $f(n), g(n) \in \text{polylog}(n)$ ***s.t. the following hold: (i)*** $\Gamma[o, d](t_o) \leq f(n) \cdot (D[o, d](t_o))^{\lambda}$, ***and (ii)*** $D[o, d](t_o) \leq g(n) \cdot (\Gamma[o, d](t_o))^{1/\lambda}$. ***Analogous inequalities hold for the free-flow and the full-congestion metrics*** $\underline{D}$ ***and*** $\overline{D}$.

Note that static oracles based on the doubling dimension demand a ***constant*** value for λ. We relax this by allowing λ to be even a (sufficiently slowly) growing function of n. We also introduce some additional slackness, by allowing divergence from the corresponding powers by polylogarithmic factors. In the rest of the paper we consider sparse TD-instances (i.e., $m \in \mathcal{O}(n)$), compliant with Assumptions 2.3.1, 2.3.2, 2.3.3, and 2.3.4.

2.4 Landmarks Selection Policies and Preprocessing of Distance Summaries

Ingredients of our oracles. A typical landmark-based oracle selects a set $L \subseteq V$ of ***landmarks*** and then preprocesses travel-time related information (called ***summaries***) between them and all (or some) reachable destinations. Consequently, a query algorithm exploits these summaries, in order to ***efficiently*** respond to queries (o, d, t_o), from an origin o and departure-time t_o, to a destination d. Typically the oracle provides also a theoretically proved approximation guarantee (stretch) of the provided

answers. In practice though, quite frequently the observed stretch is significantly smaller than the theoretical bound.

All our oracles start by selecting a subset $L \subset V$ of ***landmarks***. This can be done either randomly (e.g., by deciding for each vertex i.u.r with probability $\rho \in (0, 1)$ whether it belongs to L), or by exploiting the vertices in the cut sets provided by some graph partitioning algorithm. In this work we consider the KAHIP partitioning software.

After L is determined, a preprocessing phase is performed in which, $\forall \ell \in L$ and $\forall v \in V$, all ℓ-to-v $(1+\varepsilon)-$upper-approximating travel-time functions (we call them ***approximate travel-time summaries***) are computed and stored, based on the TRAP method, to be described later. Consequently, a query algorithm is used for providing in sublinear time ***guaranteed*** approximations of the actual shortest travel time values, for arbitrary queries $(o, d, t_o) \in V \times V \times [0, T)$. In a final step, a path-construction routine is run to provide an od-path with actual path-travel-time at most equal to the predicted one. In this section, we briefly review the above mentioned ingredients of our oracles.

Landmark Selection Policies. We now clarify the landmark selection policies that we consider:

◇ RANDOM (R): Landmarks are chosen independently and uniformly at random, with no repetitions.

◇ SPARSE-RANDOM (SR): A variant of R, where each newly chosen landmark excludes a small (free-flow) neighborhood of vertices around it from being selected as landmarks in the future.

◇ IMPORTANT-RANDOM (IR): Another variant of R, which moves each randomly selected landmark to its nearest important vertex within a free-flow neighborhood of size 100. For such a policy it is important that the underlying instance possesses a characterization of the road segments' significance. Only the instance of Berlin provides such information and we have considered as "important" those vertices which are incident to road segments of category at most 3.

◇ KAHIP (K): We choose either the tail or the head vertex of any boundary arc of a partition of the graph according to KAHIP partitioning algorithm. We used the KaFFPa algorithm of the KAHIP partitioning software (v1.00).

◇ HYBRID (H): This kind of landmark set consists of about the half vertices to be

on the boundary of the KAHIP partition and the remaining landmarks are chosen independently and uniformly at random, with no repetitions.

◇ SPARSE-KAHIP (SK): We consider the boundary vertices of a KAHIP partition of the graph as candidates for landmarks. We used the KaFFPa algorithm of the KAHIP partitioning software (v1.00), setting the parameters so that there are many more boundary vertices than the required number of landmarks. The actual landmarks are chosen sequentially and randomly. Each new landmark excludes from future selections a small free-flow neighborhood around it.

◇ KAHIP-CELLS (KC). Starting with a KAHIP partition, one landmark per cell of the partition is chosen uniformly at random, excluding a small neighborhood around it from future selections.

◇ BETWEENESS-CENTRALITY (BC): All the vertices are ordered in decreasing approximate betweeness centrality (ABC) values, which were computed according to [25]. Then, we select as landmarks the best vertices w.r.t. their ABC value, excluding a small free-flow neighborhood from each of them.

◇ KAHIP-BETWEENESS (KB): Starting from a KAHIP partition, we choose as landmark the vertex with the highest ABC value per cell of the partition, excluding a small free-flow neighborhood around it.

Finally, for the sake of the HORN oracle, we also construct hierarchical landmark sets, HIERARCHICAL-RANDOM (HR) and HIERARCHICAL-SPARSE-RANDOM (HSR) of $10,443$ and $20,886$ landmarks.

Preprocessing Phase. In the following, we shall deal with the problem of providing (the explicit representations of) shortest-travel-time functions in a time-dependent instance. In particular, we shall deal with the following problem:

Definition 2.4.1. Time Dependent Delay Approximation Functions [*22*]

INPUT : ***Directed graph*** $G = (V, A)$***.***
$(o, d) \in V \times V$***: An origin-destination pair of vertices.***
$O \subseteq V$***: A subset of potential origin vertices.***
$\forall a \in A, \overrightarrow{D}[a] : \mathbb{R} \to \mathbb{R}_{\geq 0}$***. The forward arc-delay (pwl) functions.***

SOTDDA : ***Provide (explicit representations of) approximate shortest-travel-time (delay) functions*** $\overline{D}[o, v]$ ***for all*** $v \in V$***, that will assure the following approximation guarantee:*** $\forall v \in V,\ \forall t_o \in [0, T], D[o, v](t_o) \leq \overline{D}[o, v](t_o) \leq (1 + \varepsilon) \cdot D[o, v](t_o)$.
Notation: The representation of the entire output will be denoted by $\overline{D}[o, \star]$***.***

The main criteria for the quality of solutions to the above mentioned problem are: (i) the polynomial-time construction of the required functions, and (ii) the required storage for the produced representations. Focusing only on (ii), one could have asymptotically optimal solutions, assuming prior knowledge (or, paying for the required computational cost and space to construct them) of the exact delay functions. This is not the case in our setting and it would probably be prohibitively expensive to construct the exact delay functions before space-optimally approximating them. Therefore, we have to be based only on (polynomial-time computable) samplings of the exact functions, in order to produce (as fast as possible) the required upper approximations, using as little space and assuring as good approximation guarantee as possible.

Since we wish to avoid computing the exact shape of $D[o, d]$ for a given od-pair, we need also a ***lower-bounding point-to-point approximate distance*** function for the same time-window:

$$\forall t_o \in [0, T], (1 - \varepsilon) \cdot D[o, d](t_o) \leq \underline{D}[o, d](t_o) \leq D[o, d](t_o) \tag{2.9}$$

Having two (guaranteed upper and lower) approximations $\overline{D}[o, d]$ and $\underline{D}[o, d]$ of $D[o, d]$ in the time-window of interest and an approximation guarantee between them:

$$\forall t_o \in [0, T],\ \overline{D}[o, d](t_o) \leq (1 + \varepsilon) \cdot \underline{D}[o, d](t_o) \tag{2.10}$$

would assure us that $\overline{D}[o, d] \leq (1+\varepsilon)D[o, d]$ is the required upper-approximation and $\underline{D}[o, d] \geq \left(1 - \frac{\varepsilon}{1+\varepsilon}\right) D[o, d]$ is the required lower-approximation of $D[o, d]$, without even

knowing (except for the explicitly sampled values) the actual shape of the function that is approximated.

2.5 The Trapezoidal (TRAP) Approximation Method

We present here the novel preprocessing step of our oracles which, based on the TRAP method, constructs $(1+\varepsilon)-$upper-approximations of shortest travel-time functions. The method is presented in [15].

After the landmark set L is determined, a preprocessing phase is performed in which all $(1+\varepsilon)-$upper-approximating travel-time functions (travel-time summaries) from landmarks $\ell \in L$ towards destinations $\forall v \in V$ are computed and stored, based solely on the TRAP method.

TRAP splits the entire period $[0, T)$ into small, consecutive subintervals of length $\tau>0$ each. It then provides a crude approximation of the unknown shortest-travel-time functions in each interval, solely based on Assumption 2.3.1 concecning the boundedness of the shortest travel-time slopes in the instance. As mentioned, a slight modification of Dijkstra that relaxes vertex labels according to the ***temporal*** arc-travel times of their incoming arcs, depending on the departure-times from their heads, works perfect in time-dependent networks possessing the FIFO property, for arbitrary (but given) departure-times t_o from the origin o. Eventually, the labels of the vertices reachable from o denote the earliest-arrival-time ***values***, when one departs from o at the given departure time t_o. After sampling the travel-time values (by running TDD) of each destination $v \in V$, for a given origin $u \in V$, we consider each pair of consecutive sampling times $t_s < t_f$ and the semilines with slopes Λ_{max} from t_s and $-\Lambda_{\text{min}}$ from t_f. The considered upper-approximating function $\overline{D}[u, v]$ within [t_s,t_f) is then (a refinement of) the lower-envelope of these two lines. Analogously, a lower-approximating function $\underline{D}[u, v]$ is the upper-envelope of the semilines that pass through t_s with slope $-\Lambda_{\text{min}}$, and from t_f with slope Λ_{max}. Depending on the value of the absolute error and the minimum possible value of $\underline{D}[u, v]$ in this interval, we can decide whether $\overline{D}[u, v]$ is a $(1+\varepsilon)$-upper-approximating function of $D[u, v]$. Any destination vertex that has such a $(1+\varepsilon)$-upper-approximating function for each subinterval of $[0, T)$, clearly has a $(1+\varepsilon)$-upper-approximating function for the entire period as well.

The problem with the trapezoidal approximation is that, by construction, it is not possible to provide $(1+\varepsilon)$-approximate travel-time functions for "nearby" destination vertices, which are too close to the origin. In [15] these "nearby" vertices of each landmark are either handled by the `BIS` method [16], or are left to be handled by local `TDD` searches "on the fly". Here we resolve this issue exclusively with `TRAP`, starting with a large subinterval length, and then recursively dividing by 2 the lengths of those subintervals containing vertices which have not been sufficiently approximated yet, until all landmark-to-vertex $(1+\varepsilon)$-approximate travel-time summaries have been successfully created. This proved to be extremely space- and time-efficient in practice.

More precisely, assume having a landmark vertex $\ell \in L$ and a departure-times subinterval [t_s, t_f = t_s + τ) $\subseteq$ [0,T), for some small departure-time difference value, compared to the entire period T of departure times. We provide a crude approximation, which we call `TRAP` (the trapezoidal method), for creating distance functions from any landmark $\ell \in L$ towards each posible destination $v \in V$. It is mentioned that, contrary to the approximation method `BIS` proposed in [16], no assumption is made on the shapes of the unknown distance functions to approximate within [t_s,t_f). In particular, no assumption is made on them being concave. `TRAP` will only exploit the fact that τ is indeed small, along with the ***Bounded Travel-Time Slopes*** Assumption (cf. Assumption 2.3.1). The approximation guarantee for each of these approximate distance functions actually varies, depending on the minimum (free-flow) travel-time from ℓ to each of the destinations. In particular, by Assumption 2.3.1, for any departure-time from ℓ, $t \in [t_s, t_f)$ and any destination vertex $v \in V$, the following inequalities hold [15]:

$$
\begin{aligned}
& -\Lambda_{\text{min}} \leq \frac{D[\ell,v](t) - D[\ell,v](t_s)}{t - t_s} \leq \Lambda_{\text{max}} \\
\Rightarrow \quad & -\Lambda_{\text{min}} \cdot (t - t_s) + D[\ell,v](t_s) \leq D[\ell,v](t) \leq D[\ell,v](t_s) + \Lambda_{\text{max}} \cdot (t - t_s) \\
\overset{/* \, \tau \geq t - t_s \, */}{\Rightarrow} \quad & \boxed{-\Lambda_{\text{min}} \cdot \tau + \text{D}[\ell,\text{v}](t_s) \leq \text{D}[\ell,\text{v}](\text{t}) \leq \text{D}[\ell,\text{v}](t_s) + \Lambda_{\text{max}} \cdot \tau}
\end{aligned}
$$

$$
\begin{aligned}
& -\Lambda_{\text{min}} \leq \frac{D[\ell,v](t_f) - D[\ell,v](t)}{t_f - t} \leq \Lambda_{\text{max}} \\
\Rightarrow \quad & -\Lambda_{\text{min}} \cdot (t_f - t) \leq D[\ell,v](t_f) - D[\ell,v](t) \leq \Lambda_{\text{max}} \cdot (t_f - t) \\
\overset{/* \, \tau \geq t_f - t \, */}{\Rightarrow} \quad & \boxed{\Lambda_{\text{min}} \cdot \tau + \text{D}[\ell,\text{v}](t_f) \geq \text{D}[\ell,\text{v}](\text{t}) \geq \text{D}[\ell,\text{v}](t_f) - \Lambda_{\text{max}} \cdot \tau}
\end{aligned}
$$

Combining the two inequalities we get the following bounds: $\forall v \in V, \forall t \in [t_s, t_f)$,

$$\min \left\{ \begin{array}{c} D[\ell, v](t_s) + \Lambda_{\max}\tau, \\ D[\ell, v](t_f) + \Lambda_{\min}\tau \end{array} \right\} \geq D[\ell, v](t) \geq \max \left\{ \begin{array}{c} D[\ell, v](t_s) - \Lambda_{\min}\tau, \\ D[\ell, v](t_f) - \Lambda_{\max}\tau \end{array} \right\} \quad (2.11)$$

Exploiting the fact that each shortest-travel-time function from ℓ to any destination $v \in V$ and departure time from $[t_s, t_f)$ respects the above mentioned upper and lower bounds, one could use a simple continuous, pwl approximation of $D[\ell, v]$ within this interval, which is the minimum of four linear functions:

$$\forall t \in [t_s, t_f),\ \overline{D}[\ell, v](t) = \min \left\{ \begin{array}{c} D[\ell, v](t_s) + \Lambda_{\max}\tau, \\ D[\ell, v](t_f) + \Lambda_{\min}\tau \\ \Lambda_{\max}t + D[\ell, v](t_s) - \Lambda_{\max}t_s, \\ -\Lambda_{\min}t + D[\ell, v](t_f) + \Lambda_{\min}t_f \end{array} \right\} \quad (2.12)$$

I.e., we consider the lines passing via the point $(t_s, D[\ell, v](t_s))$ with the maximum slope $\Lambda_{\max}$, until the upper bound in inequality 2.11 is reached, then follow a constant leg up to the point at which the line passing via $(t_s, D[\ell, v](t_s))$ with the minimum possible slope of $-\Lambda_{\min}$ is met.

Analogously, we construct a lower-bounding approximation of $D[\ell, v]$ within $[t_s, t_f)$.

$$\forall t \in [t_s, t_f),\ \underline{D}[\ell, v](t) = \max \left\{ \begin{array}{c} D[\ell, v](t_s) - \Lambda_{\min}\tau, \\ D[\ell, v](t_f) - \Lambda_{\max} \cdot \tau \\ \Lambda_{\max}t + D[\ell, v](t_f) - \Lambda_{\max} \cdot t_f, \\ -\Lambda_{\min}t + D[\ell, v](t_s) + \Lambda_{\min}t_s \end{array} \right\} \quad (2.13)$$

Figure 2.1 shows the (upper and lower) approximate distance summaries with respect to $D[\ell, v]$ within $[t_s, t_f)$.

Let $(\underline{t}_m, \underline{D}_m)$ be the intersection point of the two non-constant legs involved in the definition of $\underline{D}[\ell, v]$. Then it is easy to observe that:

$$\begin{aligned} \underline{t}_m &= \frac{D[\ell, v](t_s) - D[\ell, v](t_f)}{\Lambda_{\min} + \Lambda_{\max}} + \frac{\Lambda_{\min}t_s + \Lambda_{\max}t_f}{\Lambda_{\min} + \Lambda_{\max}} \\ \underline{D}_m &= \frac{\Lambda_{\max}D[\ell, v](t_s) + \Lambda_{\min}D[\ell, v](t_f)}{\Lambda_{\min} + \Lambda_{\max}} - \frac{\Lambda_{\min} \cdot \Lambda_{\max}}{\Lambda_{\min} + \Lambda_{\max}} \cdot (t_f - t_s) \end{aligned}$$

Similarly, let $(\overline{t}_m, \overline{D}_m)$ be the intersection point of the two non-constant legs involved in the definition of $\overline{D}[\ell, v]$. Then:

$$\begin{aligned} \overline{t}_m &= \frac{D[\ell, v](t_f) - D[\ell, v](t_s)}{\Lambda_{\min} + \Lambda_{\max}} + \frac{\Lambda_{\min}t_f + \Lambda_{\max}t_s}{\Lambda_{\min} + \Lambda_{\max}} \\ \overline{D}_m &= \frac{\Lambda_{\max}D[\ell, v](t_f) + \Lambda_{\min}D[\ell, v](t_s)}{\Lambda_{\min} + \Lambda_{\max}} + \frac{\Lambda_{\min}\Lambda_{\max}}{\Lambda_{\min} + \Lambda_{\max}}(t_f - t_s) \end{aligned}$$

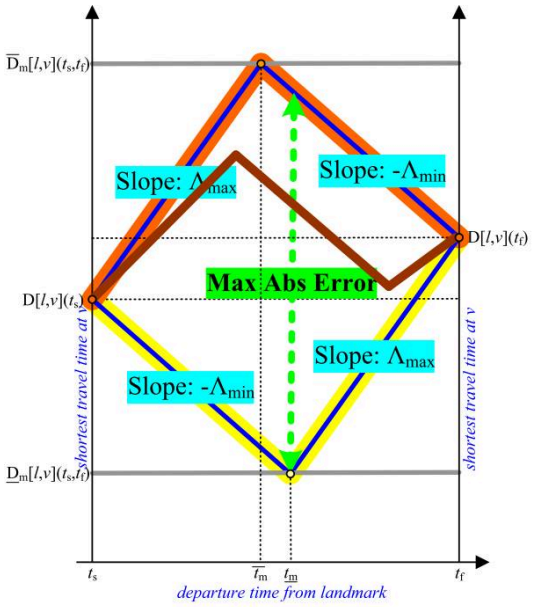


Figure 2.1: The upper-approximating function $\overline{D}[\ell, v]$ (thic orange, upper pwl line) of the unknown distance function $D[\ell, v]$ within the interval $[t_s, t_f)$. The lower-approximating function (thic yellow, lower pwl line), of the unknown distance function within the interval.

The worst-case maximum (additive) error guaranteed for $\overline{D}[\ell, v]$ within $[t_s, t_f]$ is given by the following closed form:

$$\begin{aligned} MAE(t_s, t_f) &= \max_{t \in [t_s, t_f)} \left\{ \overline{D}[\ell, v](t) - \underline{D}[\ell, v](t) \right\} \\ &= \max_{t \in \{\overline{t}_m, \underline{t}_m\}} \left\{ \overline{D}[\ell, v](t) - \underline{D}[\ell, v](t) \right\} \end{aligned} \tag{2.14}$$

The following lemma correlates the value of the maximum absolute error with the minimum possible distance from ℓ, within $[t_s, t_f)$.

Lemma 2.1. [15] ***For a given landmark vertex*** $\ell \in L$, ***any destination vertex*** $v \in V$ ***and a given departure-time subinterval*** $[t_s, t_f) \subseteq [0, T)$, ***the following hold:***

1. $MAE[\ell, v](t_s, t_f) \leq \Lambda_{\max} \cdot (t_f - t_s)$.

2. ***If at least one of the following conditions hold, then the trapezoidal method guarantees that*** $\overline{D}[\ell, v]$ ***is a*** $(1+\varepsilon)-$***approximation of*** $D[\ell, v]$ ***within*** $[t_s, t_f)$.

   (i) $D[\ell, v](t_s) \geq \left(\Lambda_{\min} + \frac{\Lambda_{\max}}{\varepsilon}\right)(t_f - t_s)$.

   (ii) $D[\ell, v](t_f) \geq \left(\Lambda_{\max} + \frac{\Lambda_{\max}}{\varepsilon}\right)(t_f - t_s)$.

Chapter 3

Engineering Oracles for Time-Dependent Shortest Paths

3.1 The FLAT Oracle

The main rationale of the FLAT oracle, presented and analysed in [15] is the following: we somehow select a small subset of landmark vertices L. Then, we compute ***travel-time summaries***, i.e., $(1+\varepsilon)-$approximations of minimum-travel-time ***functions***, from landmarks towards all reachable vertices. Finally, we use properly designed query algorithms that exploit these travel-time summaries in order to provide approximately minimum-travel-time ***values*** for arbitrary queries $(o, d, t_o) \in V \times V \times [0, T)$.

Preprocessing Phase. After L is determined, a preprocessing phase is performed in which all $(1+\varepsilon)-$upper-approximating travel-time functions (travel-time summaries) from landmarks $\ell \in L$ towards destinations $\forall v \in V$ are computed and stored, based solely on the TRAP method.

Query Algorithms. We consider three query algorithms FCA, RQA, andFCA$^+$. The first two were introduced in [16], while the third one was introduced in [26]. All algorithms can be fine-tuned to run in $o(n)$ time.

FCA grows a ball $B_o \equiv B[o](t_o) = \{x \in V : D[o,x](t_o) \leq D[o,\ell_o](t_o)\}$ from (o,t_o), by running TDD until either d or the closest landmark $\ell_o \in \arg\min_{\ell \in L}\{D[o,\ell](t_o)\}$ is settled. It then returns either the exact travel-time value, or an $(1+\varepsilon+\psi)$-approximate travel-time value via ℓ_o, where ψ is a constant depending on ε, ζ, and $\Lambda_{\max}$, but not on n.

RQA is a PTAS, providing an approximation guarantee of $1+\sigma = 1+\varepsilon\cdot\frac{(1+\varepsilon/\psi)^{r+1}}{(1+\varepsilon/\psi)^{r+1}-1}$, by exploiting carefully a number $r \in \mathbb{N}$ (called the ***recursion budget***) of recursive accesses to the preprocessed information, each of which produces (via calls to FCA) additional candidate $od-$paths sol_i. RQA works as follows. As long as the destination vertex within the explored area around the origin has not yet been discovered, and there is still some remaining recursion budget, it "guesses" (by exhaustively searching for it) the next vertex w_k of the boundary set of touched vertices (i.e., still in the priority queue) along the unknown shortest $od-$path. Then, it grows an outgrowing TDD ball from the new center $(w_k, t_k = t_o + D[o,w_k](t_o))$, until it reaches the closest landmark ℓ_k to it, at travel-time $R_k = D[w_k,\ell_k](t_k)$. This new landmark offers an alternative $od-$path sol_k by a new application of FCA.

FCA$^+(N)$ is a variant of FCA that keeps growing a TDD ball from (o,t_o) until either d, or a given number N of landmarks is settled, and then returns the smallest via-landmark approximate travel-time value (among all N settled landmarks). The approximation guarantee is the same as that of FCA, but its practical performance is impressive (in most cases even better than RQA).

A more detailed presentation of FCA and RQA, along with the proofs of correctness and their time complexities, are provided in [16]. As for the approximation guarantee of FCA$^+$, it is straightforward to observe that, at least theoretically, it is as small as that of FCA, whereas its time complexity is comparable to that of RQA.

3.1.1 Constant-approximation Query Algorithm

A distance oracle needs a fast query algorithm providing constant approximation to the shortest-travel-time values of arbitrary queries $(o,d,t_o) \in V \times V \times [0,T)$. The proposed query algorithm, called ***Forward Constant Approximation*** (FCA), grows an

outgoing ball

$$B_o := B[o](t_o) = \{x \in V : D[o,x](t_o) \leq \min\{D[o,d](t_o), D[o,\ell_o](t_o)\}\}$$

from (o, t_o), by running TDD until either d or the closest landmark $\ell_o \in \arg\min_{\ell \in L}\{D[o,\ell](t_o)\}$ is settled. We call $R_o = \min\{D[o,d](t_o), D[o,\ell_o](t_o)\}$ the ***radius*** of B_o. If $d \in B_o$, then FCA returns the ***exact*** travel-time $D[o,d](t_o)$; otherwise, it returns the approximate travel-time value $R_o + \overline{D}[\ell_o, d](t_o + R_o)$ via ℓ_o.

The algorithm was first presented and validated by S. Kontogiannis and C. Zaroliagis in [16].

Figure 3.1 gives an overview of the whole idea. Figure 3.2 provides the pseudocode.

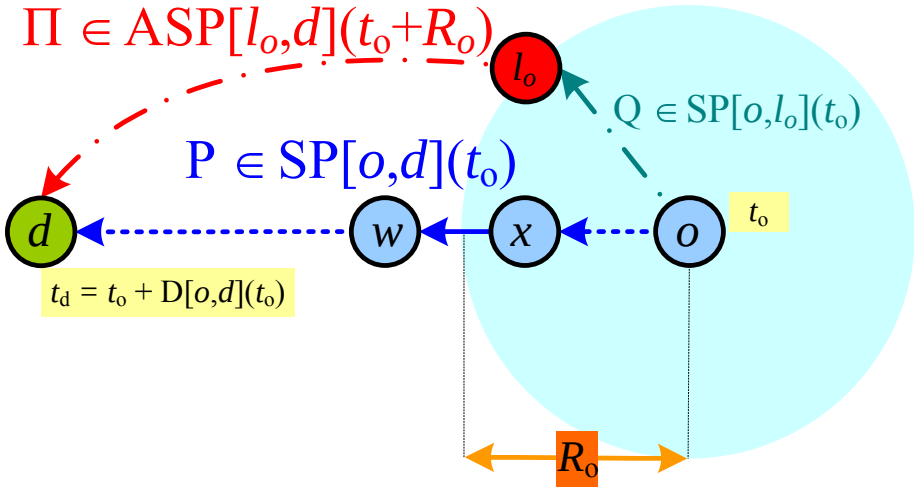


Figure 3.1: The rationale of FCA. The dashed (blue) path P is a shortest $od-$path for (o, d, t_o). The dashed-dotted (green and red) path $Q \bullet \Pi$ is the via-landmark $od-$path indicated by the algorithm, if the destination vertex is out of the origin's TDD ball.

3.1.2 $(1+\sigma)-$approximate Query Algorithm

The ***Recursive Query Algorithm*** (RQA) improves the approximation guarantee of the chosen $od-$path provided by FCA, by exploiting carefully a number of recursive calls of FCA, based on a given bound – called the ***recursion budget*** r – on the depth of the recursion tree to be constructed. Each of the recursive calls accesses the preprocessed information and produces another candidate $od-$path. The crux of our approach

FCA(o, d, t_o)
1. **if** $o \in L$ **then return** $\left(\overline{D}[o, d](t_o)\right)$ /* $(1+\varepsilon)-$approximate answer */
2. $B_o =$ TDD-ball around (o, t_o) until either d or the first landmark is settled
3. **if** $d \in B_o$ **then return** $(D[o, d](t_o))$ /* exact answer */
4. $\ell_o = B_o \cap L; R_o = D[o, \ell_o](t_o);$
5. **return** $\left(R_o + \overline{D}[\ell_o, d](t_o + R_o)\right)$ /* $(1+\varepsilon+\psi)-$approximation */

Figure 3.2: The pseudocode describing FCA.

is the following: We ensure that, unless the required approximation guarantee has already been reached by a candidate solution, the recursion budget must be exhausted and the sequence of radii of the consecutive balls that we grow from centers lying on the unknown shortest path, is lower-bounded by a ***geometrically increasing*** sequence. We prove that this sequence can only have a ***constant*** number of elements until the required approximation guarantee is reached, since the sum of all these radii provides a lower bound on the shortest-travel-time that we seek.

A similar approach was proposed for ***undirected*** and ***static*** sparse networks [18], in which a number of recursively growing balls (up to the recursion budget) is used in the vicinities of ***both*** the origin ***and*** the destination nodes, before eventually applying a constant-approximation algorithm to close the gap, so as to achieve improved approximation guarantees.

In our case the network is both directed and time-dependent. Due to our ignorance of the exact arrival time at the destination, it is difficult (if at all possible) to grow incoming balls in the vicinity of the destination node. Hence, our only choice is to build a recursive argument that grows outgoing balls in the vicinity of the origin, since we only know the requested departure-time from it. This is exactly what we do: As long as we have not discovered the destination node within the explored area around the origin, and there is still some remaining recursion budget $r - k > 0$ ($k \in \{0, \ldots, r\}$), we "guess" (by exhaustively searching for it) the next node w_k along the (unknown) shortest $od-$path. We then grow a new out-ball from the new center $(w_k, t_k = t_o + D[o, w_k](t_o))$, until we reach the closest landmark-vertex ℓ_k to it, at distance $R_k = D[w_k, \ell_k](t_k)$. This new landmark offers an alternative $od-$path $sol_k = P_{o,k} \bullet Q_k \bullet \Pi_k$ by a new application of FCA, where $P_{o,k} \in SP[o, w_k](t_o)$, $Q_k \in$

$SP[w_k, \ell_k](t_k)$, and $\Pi_k \in ASP[\ell_k, d](t_k + R_k)$ is the approximate suffix subpath provided by the distance oracle. Observe that sol_k uses a ***longer*** optimal prefix-subpath P_k which is then completed with a shorter approximate suffix-subpath $Q_k \bullet \Pi_k$.

The algorithm was first presented and validated by S. Kontogiannis and C. Zaroliagis in [16].

Figure 3.3 provides an overview of RQA's execution.

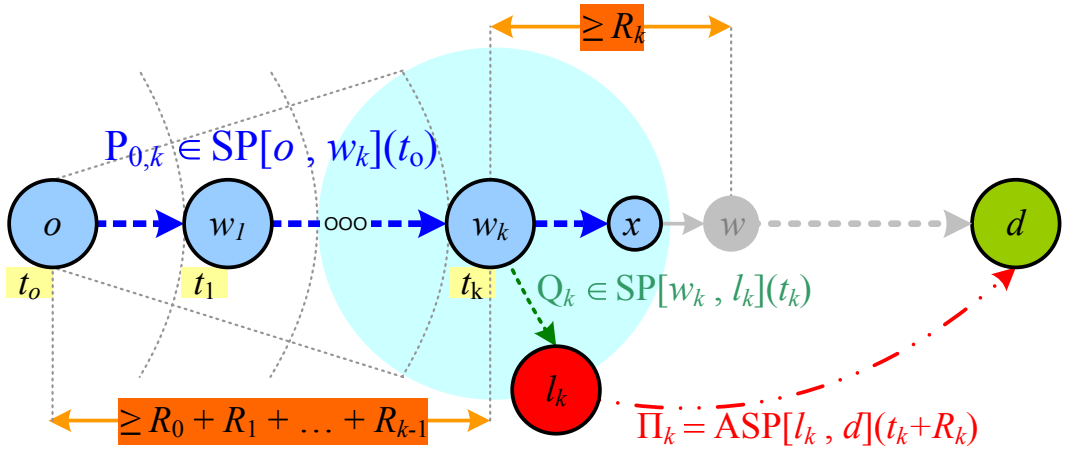


Figure 3.3: Overview of the execution of RQA.

Figure 3.4 provides the pseudocode of RQA.

3.1.3 The Path Reconstruction

Finally, we describe the ***Path Reconstruction*** method followed for the generation of the calculated ***o-d*** path, as a sequence of arcs. In the case that the shortest path returned by the query algorithm is exact, i.e. the destination was discovered during the TDD-search (which is actually quite possible to occur), the path is constructed by simply following the predecessors from the destination to the origin, which the TDD ball provided. However, if the algorithm decides that the destination is to be reached via an appropriate landmark ℓ, we need a method to retrieve the unknown sequence of predecessors from the destination up to the landmark, corresponding to the approximate ℓ-***d*** path. For this purpose, we exploit the preprocessed information. For every possible destination ***d*** and for any departure time from ℓ, the travel-time function contains the immediate predecessor for ***d***, valid for a specific time interval, where TRAP performed its sampling. Based on this information and some heuristic ideas, the path reconstruction method works as follows.

Let v denote every node that belongs on the path we want to construct. We start with $v = d$. We then obtain the immediate (approximate) predecessor, ***approxPred(v)***, given by the approximate travel-time function $D[\ell,d](t)$, when departing from ℓ at time $t_l = Arr[o,\ell](t_o)$, which denotes the arrival time set by the Dijkstra-ball. We then mark node v as ***visited***, we set $v =$ ***approxPred(v)*** and repeat. The procedure terminates when we reach the landmark-node ℓ, i.e. $v = \ell$. The retrieval of all predecessors is done by searching the preprocessed data.

In practice we observed the following phenomena which we tackled accordingly. Firstly, as we reversely approach the landmark ℓ, the sequence of nodes v at some point enters the area explored by the query algorithm. We decided to collect all those explored nodes v and calculate the total travel-time $D[v,d](t_v)$, exploiting the ***reverse*** arc-travel-time functions on all arcs connecting all nodes v up to that point. When the main loop of our method terminates, we check which explored node on the path we constructed (including the landmark ℓ) gives the minimum $D[o,v](t_o) + D[v,d](t_v)$. This means that there can be several cases in which we construct the approximate path via an appropriate explored node v by the Dijkstra-ball rather than via the proposed landmark ℓ.

Next, we observed that a predecessor given by the preprocessed distance-summaries can be already ***visited***, which means that a cycle is created. This can be expected since we search into different approximate functions for each vertex v, departing from ℓ. The TRAP method samples the exact travel-time function in different subintervals for each destination. We choose to face this case as follows. The path reconstruction method returns to its initial step, where $v = d$. Instead departing from the landmark ℓ at the exact departure time t_l, we seek for the closest departure time to t_l, contained as a breakpoint in all approximate distance functions of the predecessors involved to an approximate path up to ℓ. This safely means that this departure time is a sampling time for all those destinations and thus, they all belong to the very same shortest-path tree, creted by the TRAP method during the preprocessing. To avoid the cycle (which in practice is a rare case), we consider the sequence of vertices created, considering the above departure time from ℓ, which is usually very close to the actual.

After the sequence of predecessors has been constructed, the method simply walks on the edges connecting them an the ℓ-v path-travel-time value is provided by the (actual) path-travel-time function, when departing from ℓ at its actual arrival-time, set by the query algorithm. The resulting value is at most the approximate one. In

practice, we obtain a much better travel-time response. The last step is to connect the constructed ℓ-v path with the exact o-ℓ path, leading to a total o-d route-response, which is actually very close to the exact one.

3.1.4 Compressing Preprocessing Space

Due to the criticality of the preprocessing space, our goal is to achieve an efficient storage of the constructed travel-time summaries, while keeping a sufficient precision. The key is that some specific features can be exploited in order to reduce the required space.

Contraction of the road network. The preprocessing space and time can be reduced if we only focus on a subgraph of the underlying graph representing the road network. Towards this direction, we have chosen to "contract" all the vertices which do not depict junctions of road segments (e.g., intermediate stops along a road segment). We consider these vertices as ***inactive*** (only for the preprocessing phase), and we do not consider them during the subsequent preprocessing of travel-time related information, since they do not provide actual alternatives along a route using them, unless they are indeed endpoints of the query at hand. It is emphasized though, that the queries are conducted in the original graph, not just the contracted subgraph, meaning that we can query also for contracted origin-destination pairs and the returned paths do not contain shortcuts but actual road segments.

In more detail, in the ***instance-contraction phase*** we seek for maximal w.r.t. the number of arcs (possibly bidirectional) paths which have no "vertical" intersections, i.e., all the intermediate vertices connect only with their neighboring vertices along the path. Each such path is substituted with a ***shortcut*** (arc) connecting its endpoints, which is equipped with an arc-travel-time function equal to the corresponding exact path-travel-time function. In fact, multiple paths with no intermediate intersections may connect the same active endpoints. In that case, a single shortcut represents more than one contracted paths, i.e. the arc-travel-time function of the shortcut is computed by applying the minimization operator on the path-travel-time functions corresponding to each of the contracted paths. If there exists an original arc connecting two active endpoints, which are to be connected with a shortcut, we choose not to insert an additional shortcut, but to update accordingly the arc-travel-time of the

already existing arc which now plays the role of a shortcut as well. The original arcs involved in the contracted paths are also considered as ***inactive***. All contracted vertices are ignored during the landmark-preprocessing and therefore the number of reachable destinations from a landmark is smaller. At the query phase, the contracted paths can be easily recovered, by exploiting the appropriate information kept on all shortcuts and the corresponding contracted vertices.

Almost Constant Legs. The original TRAP approximation method [27] introduced at least one intermediate breakpoint per interval that does not yet meet the required approximation guarantee. This is certainly unnecessary for small intervals in which the actual shortest-travel-time functions are constant. To avoid the blow-up of the required preprocessing space, we heuristically make a "guess" that we have to deal with a constant shortest-travel-time function $D[\ell, v]$ within a given interval $[t_s, t_f = t_s + \tau)$ with sufficiently small length τ, whenever the following holds: $D[\ell, v](t_s) = D[\ell, v](t_f) = D[\ell, v]\left(\frac{t_s+t_f}{2}\right)$. This is justified by the fact that $D[\ell, v]$ is a continuous pwl function and it is unlikely that three different departure times within a small interval would give the same value, unless the function is indeed constant. Of course, one could easily construct artificial examples for which this criterion is violated, e.g. by providing a properly chosen periodic function with period $\tau/2$. On the other hand, one can easily tackle this by considering a ***randomly perturbed*** sampling period $\tau + \delta$, for some arbitrarily small but positive random variable δ. Since we engineer oracles for real-world road-networks, having three colinear points which do not belong to a leg of the sampled travel-time function is quite unlikely, therefore we choose not to randomly perturb the sampling period.

Fixed Range. For a one-day time period, departure-times and arrival-times have a bounded value range. The same also holds for travel times which are at most one-day for any query within a country area such as Germany. Therefore, when the considered precision of the traffic data is within seconds, we handle time-values as integers in the range $\{0\ ,\ 1\ ,\ \ldots\ ,\ 86,399\}$, for milliseconds as integers in $\{0\ ,\ 1\ ,\ \ldots\ ,\ 86,399,999\}$, etc.

Any (real) time value within a single-day period, represented as a floating-point number t_f, can thus be converted to an integer t_i with fewer bytes and a given unit of measure. For a unit measure (or scale factor) s, the resulting integer is $t_i =$

$\left\lceil \frac{t_f}{s} \right\rceil$, requiring $\left\lceil \frac{\log_2(t_f/s)}{8} \right\rceil$ bytes for its storage. The division $\frac{t_f}{s}$ has quotient π and remainder υ s.t., $t_f = s \cdot \pi + \upsilon$, and $t_i = \left\lceil \frac{s \cdot \pi + \upsilon}{s} \right\rceil = \pi + \left\lceil \frac{\upsilon}{s} \right\rceil \in \{\pi, \pi + 1\}$, since $0 \leq \upsilon \leq s - 1$. Therefore, by storing t_i we actually consider the upper-approximating time $t'_f = s \cdot t_i$ of t_f, which causes an absolute error of at most s (i.e., one unit of measure): $t'_f - t_f < s \cdot (\pi + 1) - s \cdot \pi = s$. In our experiments, for storing the time values involved in the approximate shortest-travel-time functions, we have considered a 1.32***sec*** resolution, corresponding to the appropriate scale factor $s = 1.318359375$ (when originally counting time in seconds), that requires 2 bytes per time-value.

Bucketing. The number of breakpoints of the arc-travel-time functions is a major factor of space increase on the resulting minimum-travel-time functions. A way to deal with this is by merging consecutive breakpoints having absolute difference in travel-time values less than few seconds, in each arc travel time function. In this manner, to preserve the upper bound error, each resulted breakpoint gets the largest travel time among the breakpoints which take part in merge. Depending on the bucketing parameter c, we can decrease the number of breakpoints, sacrificing part of accuracy. In our experiments, the bucketing led to the highest reduction (about 86%) in space requirements.

Piecewise Composition. Many shortest od-paths typicaly contain at least one arc with pwl travel-time function, making $D[o, d](t)$ also a pwl function. To avoid the space increase from storing breakpoints unrestrictedly, we analyse any such shortest od-path into two subpaths o-p-d. p is selected so that the pd-subpath is the maximum subpath with no arc having pwl travel-time function. Such pd-subpaths exist because the number of constant arc-travel-time functions is much larger than the number of the pwl ones. Thus, for $D[o, d](t)$ we only store a "predecessor" pointer to $D[o, p](t)$ and the ***constant*** travel-time offset $D[p, d]$ i.e., $D[o, d](t) = D[o, p](t) + D[p, d]$. In our experiments, this method lead to 40% reduction of the space requirements.

Delay Shifts. There are many shortest od-paths with travel-time $D[o, d](t)$ with delay variation. Thus we further reduced the required space as follows. The delay fluctuates around a constant value. By taking the minimum delay value, the leg-delays can be represented as small shifts from this value. Those small shifts, belonging to a smaller value range, can be stored even in 1 byte. This conversion led to more than 5%

reduction of space requirements.

Compression. Since there is no need for all landmarks to be concurrently active, we can compress their data blocks. We used the library $zlib$ for this compression, which led to 10% reduction in space.

Hierarchy of subintervals Another improvement that we adopt is that, rather than splitting the entire period $[0, T)$ in a flat manner into ***equal-size*** intervals, we start with a coarse partitioning based on a large length and then in each inteval and for each destination vertex we check for the provided approximation guarantee by TRAP. All the vertices which are already satisfied by this guarantee with respect to the current interval, become inactive for this and all its subsequent subintervals. We then proceed by splitting in the middle every subinterval that contains at least one still active destination vertex, and repeating the check for all active vertices within the new subintervals.

Indexing Travel-Time Summaries. For retrieving efficiently the required minimum travel-time function $D[\ell, d](t)$ from a landmark ℓ to a destination-node d, we need also to store an index. Depending on the oracle, we used two types of indices.

We maintain a vector of pointers per landmark, one pointer equals per destination. The pointer of destination v provides the address of the $D[\ell, v](t)$ data. The pointers are in ascending order of node ID. The search time is $\mathcal{O}(1)$ and the required space is $\mathcal{O}(n|L|)$.

Required space. For Berlin, the required space was limited to an average size of less than 14MB per landmark. For storing the time-values of approximate travel-time summaries, we considered 2.64sec as resolution, corresponding to a scale factor $s = 1.32$ (when counting time in seconds), which requires 2 bytes per time-value.

RQA(o, d, t_o, r)

1. **if** $o \in L$ **then return** $\big(ASP[o,d](t_o), \overline{D}[o,d](t_o)\big)$ /* $(1+\varepsilon)-$approximation */
2. $B[o](t_o) :=$ TDD-ball from (o, t_o) until either d or a landmark is settled
3. **if** $d \in B_o$ **then return** $(D[o,d](t_o))$ /* exact suffix-subpath */
4. $\ell_0 \in B[o](t_o) \cap L;\ R_0 = D[o, \ell_0](t_o)$
5. $sol_0 = \big(Q_0 \bullet \Pi_0\ ,\ \overline{D}[sol_0](t_o) = R_0 + \overline{D}[\ell_0, d](t_o + R_0)\big)$ /* via-ℓ_o approximation */
6. $k := 0;\ t_k = t_o;$
7. **while** $k < r$ **do**

7.1. "guess" the first vertex $w_{k+1} \in SP[w_k, d](t_k) \setminus B[w_k](t_k)$ /* exhaustive search */

7.2. $t_{k+1} = t_k + D[w_k, w_{k+1}](t_k);$

7.3. **if** $w_{k+1} \in L$

7.4. **then return** $\big(P_{0,k+1} \bullet \Pi[w_{k+1}, d](t_{k+1}), t_{k+1} - t_0 + \overline{D}[w_{k+1}, d](t_{k+1})\big)$
/* approximate answer via w_{k+1} */

7.5. $B[w_{k+1}](t_{k+1}) :=$ TDD-ball until d or a landmark is settled

7.6 **if** $d \in B[w_{k+1}](t_{k+1})$ **then**

7.7. **then** $sol_{k+1} = \left(\begin{array}{c} P_{0,k+1} \bullet P_{k+1,d}, \\ \overline{D}[sol_{k+1}](t_o) = t_{k+1} - t_o + D[w_{k+1}, d](t_{k+1}) \end{array} \right)$

7.8. **else**

7.8.1 $\ell_{k+1} \in L \cap B[w_{k+1}](t_{k+1});\ R_{k+1} = D[w_{k+1}, \ell_{k+1}](t_{k+1})$

7.8.2 $sol_{k+1} = \left(\begin{array}{c} P_{0,k+1} \bullet Q_{k+1} \bullet \Pi_{k+1}, \\ \overline{D}[sol_{k+1}](t_o) = t_{k+1} - t_o + R_{k+1} \\ \qquad + \overline{D}[\ell_{k+1}, d](t_{k+1} + R_{k+1}) \end{array} \right)$
/* approximate answer via ℓ_{k+1} */

7.9. $k = k + 1$

8. **endwhile**
9. **return** $\min_{0 \le k \le r} \{sol_k\}$

Figure 3.4: The recursive algorithm RQA providing $(1+\sigma)-$approximate time-dependent shortest paths. $Q_k \in SP[w_k, \ell_k](t_k)$ is the shortest path connecting w_k to its closest landmark w.r.t. departure-time t_k. $P_{0,k} \in SP[o, w_k](t_o)$ is the prefix of the shortest $od-$path that has been already discovered, up to vertex w_k. $\Pi_k = ASP[\ell_k, d](t_k + R_k)$ denotes the $(1+\varepsilon)-$approximate shortest $\ell_k d-$path precomputed by the oracle.

3.2 The HORN Oracle

The novelty of the HORN oracle [15] is to create a hierarchy of landmark sets, whose range of "preprocessed destinations" gradually ranges from a few "nearby" vertices up to all reachable vertices (in the last level), in order to serve each query (o, d, t_o) only with relevant landmarks with respect to its own Dijkstra rank $\Gamma[o, d](t_o)$. This way, we aim at achieving speedups similar to those of FLAT for long-range queries, to all possible ranges of queries, while increasing the space requirements only by a small factor. Our goal is to "guess" the order of the Dijkstra Rank $\Gamma[o, d](t_o)$ for (o, d, t_o). The guessing is achieved in a way that is typical in online algorithms that have to deal with an unknown parameter: Starting from a small value (say, $\mathcal{O}(\sqrt{n})$), we keep growing a ball from (o, t_o), increasing appropriately the value of the guess as the ball grows, until the very first time at which a successful completion of a proper variant of RQA is very likely to occur (exactly because we "guessed right" the actual Dijkstra rank). The travel-time returned is that of the best possible od-path among all the successfully discovered approximate od-paths so far, via "informed" landmarks that possess travel-time summaries for d. The crux is in organizing the preprocessed information in such a way that it is indeed possible for the query algorithm to successfully complete its execution as soon as the "guess" asymptotically matches the value of $\Gamma[o, d](t_o)$.

The ***Hierarchical Query Algorithm*** (HQA) for (o, d, t_o) proceeds as follows: a single ball grows from (o, t_o), until either d is reached, or an ***Early Stopping criterion*** (ESC) is fulfilled, or the ***Appropriate Level of Hierarchy*** (ALH) of landmarks is reached (whichever occurs first). If d is settled by the ball from (o, t_o), an exact solution is returned. If ESC causes HQA to terminate, then the value $D[o, \ell_o](t_o) + \overline{\Delta}[\ell_o, d](t_o + D[o, \ell_o](t_o))$ is reported, because it is already a very good approximation. Otherwise, HQA, due to ALH, considers being at the right level-i of the hierarchy and continues executing the corresponding variant of RQA, call it RQA_i, which uses as its own landmark set $M_i = \cup_{j=i}^{4} L_j$. Observe that RQA_i may now fail constructing approximate shortest paths via certain landmarks in M_i that it settles, since they may not possess a travel-time summary for d. HQA terminates by returning the best od-path that has been discovered so far, via ***all*** settled landmarks which are "informed" (i.e., they have d in their coverage), either by the very first ball from (o, t_o) or by RQA_i. HQA uses some parameters: a is the degree of sublinearity in the query time, compared to the targeted Dijkstra rank; β is related to the approximation guarantee achieved upon exit due to

ESC; γ has to do with the number of levels that we create in the hierarchy; and ξ is the amount of slackness that we introduce in the size of the area of coverage. We set these parameters here to the values $a = 1,\ \beta = 1,\ \gamma = 1.88,\ \xi = 0.1$. A more detailed explanation of HQA, as well as of its parameters, is provided in [15].

In the following, we describe and analyze in detail the ***Hierarchical ORacle for time-dependent Networks*** (HORN), whose query algorithm is highly competitive against TDD, not only for long-range queries (i.e., having Dijkstra-Rank proportional to the network size) but also for medium- and short-range queries, while ensuring ***sub-quadratic*** preprocessing space and time [15]. The main idea of HORN is to preprocess: many landmarks, each possessing summaries for a few destinations around them, so that all short-range queries can be answered using only these landmarks; fewer landmarks possessing summaries for more (but still not all) destinations around them, so that medium-range queries be answered by them; and so on, up to only a few landmarks (those required by FLAT) possessing summaries for all reachable destinations. The ***area of coverage*** $C[\ell] \subset V$ of ℓ is the set of its nearby vertices, for which ℓ possesses summaries. ℓ is called ***informed*** for each $v \in C[\ell]$, and ***uninformed*** for each $v \in V \setminus C[\ell]$. The landmarks are organized in a hierarchy, according to the sizes of their areas of coverage. Each level L_i in the hierarchy is accompanied with a ***targeted Dijkstra-Rank*** $N_i \in [n]$, and the goal of HORN is to assure that L_i should suffice for RQA to successfully address queries (o, d, t_o) with $\Gamma[o, d](t_o) \leq N_i$, in time $\mathrm{o}(N_i)$. The difficulty of this approach lies in the analysis of the query algorithm. We want to execute a variant of RQA which, based on a ***minimal*** subset of landmarks, would guarantee a $(1 + \sigma(r))$-approximate solution for any query (o, d, t_o) (as in TRAPONLY and FLAT), but also time-complexity sublinear in $\Gamma[o, d](t_o)$. We propose the ***Hierarchical Query Algorithm*** (HQA) which grows an initial TDD ball from (o, t_o) that stops only when it settles an informed landmark ℓ w.r.t. d which is at the "right distance" from o, given the density of landmarks belonging to the same level with ℓ. HQA essentially "guesses" as appropriate level-i in the hierarchy the level that contains ℓ, and continues with the execution of RQA with landmarks having coverage at least equal to that of ℓ.

Initialization of HORN. We use the following parameters for the hierarchical construction: (i) $k \in \mathcal{O}(\log\log(n))$ determines the number of levels (minus one) comprising the hierarchy of landmarks. (ii) $\gamma > 1$ determines the actual values of the targeted Dijkstra-Ranks, one per level of the hierarchy. For example, as γ gets closer to 1,

the targeted Dijkstra-Ranks accumulate closer to small- and medium-rank queries. (iii) $\delta \in (0,1)$ is the parameter that quantifies the sublinearity of the query algorithm (HQA), in each level of the hierarchy, compared to the targeted Dijkstra-Rank of this level. In particular, if N_i is the targeted Dijkstra-Rank corresponding to level-i in the hierarchy, then HQA should be executed in time $\mathcal{O}\big((N_i)^{\delta}\big)$, if only the landmarks in this level (or in higher levels) are allowed to be used.

Preprocessing of HORN. $\forall i \in [k]$, we set the targeted Dijkstra-Rank for level-i to $N_i = n^{(\gamma^i-1)/\gamma^i}$. Then, we construct a randomly chosen level-i landmark set $L_i \subset_{\mathbf{uar}(\rho_i)} V$, where $\rho_i = N_i^{-\delta/(r+1)} = n^{-\delta(\gamma^i-1)/[(r+1)\gamma^i]}$. Each $\ell_i \in L_i$ acquires summaries for all (and only those) $v \in C[\ell_i]$, where $C[\ell_i]$ is the smallest ***free-flow*** ball centered at ℓ_i containing $c_i = N_i \cdot n^{\xi_i} = n^{(\gamma^i-1)/\gamma^i+\xi_i}$ vertices, for a sufficiently small constant $\xi_i > 0$. The summaries to the $F_i = c_i^{\chi}$ nearby vertices around ℓ_i are constructed with BIS; the summaries to the remaining $c_i - F_i$ faraway vertices of ℓ_i are constructed with TRAP, where $\chi = \frac{\theta}{\nu} = \frac{1+\alpha}{2+\alpha\nu} \in \left[\frac{1}{2}, \frac{2}{2+\nu}\right]$ is an appropriate value determined to assure the correctness of FLAT w.r.t. the level-i of the hierarchy. An ultimate level $L_{k+1} \subset_{\mathbf{uar}(\rho_{k+1})} V$ of landmarks, with $\rho_{k+1} = n^{-\frac{\delta}{r+1}}$, assures that HORN is also competitive against queries with Dijkstra-Rank greater than $n^{(\gamma^k-1)/\gamma^k}$. We choose in this case $c_{k+1} = N_{k+1} = n$, $F_{k+1} = n^{\chi}$ and $C[\ell_{k+1}] = V$, $\forall \ell_{k+1} \in L_{k+1}$.

Description of HQA. A TDD ball from (o, t_o) is grown until d is settled, or the ***(ESC)***-criterion or the ***(ALH)***-criterion is fulfilled (whichever occurs first):

$\diamond$ **Early Stopping Criterion (ESC)**: $\ell_o \in L = \cup_{i\in[k+1]} L_i$ is settled, which is informed ($d \in C[\ell_o]$) and, for $\varphi \geq 1$, $\frac{\overline{\Delta}[\ell_o,d](t_o+D[o,\ell_o](t_o))}{D[o,\ell_o](t_o)} \geq (1+\varepsilon)\cdot\varphi\cdot(r+1)+\psi-1$.

$\diamond$ **Appropriate Level of Hierarchy (ALH)**: For some level $i \in [k]$ of the hierarchy, the first landmark $\ell_{i,o} \in L_i$ is settled such that: ***(i)*** $d \in C[\ell_{i,o}]$ ($\ell_{i,o}$ is "informed"); and ***(ii)*** $\frac{N_i^{\delta/(r+1)}}{\ln(n)} \leq \Gamma[o,\ell_{i,o}](t_o) \leq \ln(n)\cdot N_i^{\delta/(r+1)}$ ($\ell_{i,o}$ is at the "right distance"). In that case, HQA concludes that i is the "appropriate level" of the hierarchy to consider. Observe that the level-$(k+1)$ landmarks are always informed. Thus, if no level-$(\leq k)$ informed landmark is discovered at the right distance, then the first level-$(k+1)$ landmark that will be found at distance larger than $\ln(n)\cdot N_k^{\delta/(r+1)}$ will be considered to be at the right distance, and then HQA concludes that the appropriate level is $k+1$.

If d is settled, an exact solution is returned. If (ESC) causes termination of HQA, the value $D[o,\ell_o](t_o) + \overline{\Delta}[\ell_o,d](t_o + D[o,\ell_o](t_o))$ is reported. Otherwise, HQA stops the initial

ball due to the (ALH)-criterion, considering $i \in [k+1]$ as the appropriate level, and then continues executing the variant of RQA, call it RQA_i, which uses as its landmark set $M_i = \cup_{j=i}^{k+1} L_j$. Observe that RQA_i may fail constructing approximate solutions via certain landmarks in M_i that it settles, since they may not be informed about d. Eventually, HQA returns the best od-path (w.r.t. the approximate travel-times) among the ones discovered by RQA_i via ***all*** settled and informed landmarks ℓ. Theorem 3.1 [15] summarizes the performance of HORN.

Theorem 3.1. [*15*] ***Consider any TD-instance with*** $\lambda \in \mathrm{o}\Big(\sqrt{\frac{\log(n)}{\log\log(n)}}\Big)$ ***and*** $g(n), f(n) \in \mathrm{polylog}(n)$ ***(cf. Assumption 2.3.4). For*** $\varphi = \frac{\varepsilon\cdot(r+1)}{\psi\cdot(1+\varepsilon/\psi)^{r+1}-1}$ ***and*** $k \in \mathcal{O}(\log\log(n))$, ***let*** $\xi_i \in \Big(\Big[(1+\lambda)\cdot\log\log(n) + \lambda\log\Big(1+\frac{\zeta}{1-\Lambda_{\min}}\Big)\Big] / \log(n)\ ,\ 1-\gamma^{-i}\Big)$, ***for all*** $i \in [k]$. ***Then, for any query*** (o,d,t_o) ***s.t.*** $N_{i^*-1} < \Gamma[o,d](t_o) \leq N_{i^*}$ ***for some*** $i^* \in [k+1]$, ***any*** $\delta \in (\alpha, 1)$, $\beta > 0$, ***and*** $r = \Big\lfloor \frac{\delta}{\alpha}\cdot\frac{(2/\nu+\alpha)(1-\gamma)}{\beta\cdot(2/(\alpha\nu)+1)+2/\nu-1}\Big\rfloor - 1$, HORN ***achieves*** $\mathbb{E}\{Q_{\mathsf{HQA}}\} \in (N_{i^*})^{\delta+o(1)}$, $P_{\mathsf{HORN}}\ ,\ S_{\mathsf{HORN}} \in n^{2-\beta+o(1)}$ ***and stretch*** $1+\varepsilon\frac{(1+\varepsilon/\psi)^{r+1}}{(1+\varepsilon/\psi)^{r+1}-1}$, ***with probability at least*** $1-\mathcal{O}\big(\frac{1}{n}\big)$.

Horn Index. For each node v we maintain a vector of pointers. The number of pointers equals to the number of landmarks, from any partition level, which have travel time to the corresponding node v as destination. Obviously, there exist at least one landmark from the highest partition level. The pointer of the associated landmark ℓ provides the address of the $D[\ell, v](t)$ data. The pointers are stored in ascending order of node ID. The search time is $\mathcal{O}(\log(|L|))$ and the required space is $\mathcal{O}(n|L|)$.

3.3 The CFLAT Oracle

The currently most successful oracle is FLAT [27, 15]. However, despite their remarkable query response times, oracles suffer from high preprocessing space and time requirements.

In this section we present our novel oracle, CFLAT, which can be considered as the combinatorial analogue of FLAT, aiming mostly at a ***significant*** reduction in the required space without compromising the query-time, and also achieving smaller stretch factors.

This oracle is based on the CTRAP approximation mathod, which computes and stores only shortest path trees at carefully sampled departure-times, rather than actual breakpoints of the corresponding minimum-travel-time functions as TRAP does.

We present here an overview of the main steps, as well as the major differences compared to TRAP, which allow the significant reduction in preprocessing space (and time) requirements. In a nutshell, CTRAP, when executed from a landmark ℓ, works as follows:

procedure CTRAP(ℓ)
STEP 1: Keep sampling departure-times from $[0, T)$, until all the destinations achieve a desired approximation guarantee. Only predecessors of time-stamped shortest-path trees, rooted at ℓ, are stored per destination. **STEP 2**: Cleanup each sequence of predecessors, by merging consecutive records with the same predecessor. **STEP 3**: Look for destinations with identical sequences of departure-times (time-stamps) and write this common sequence only once, for a representative destination, whereas the rest (non-representative) destinations keep only the sequences of the corresponding predecessors.

STEP 1 resembles TRAP, the only difference being that CTRAP keeps the immediate predecessors (parents) per active destination v in the sampled shortest-path trees. In particular, CTRAP maintains a pair of sequences, $PRED[\ell, v]$ for predecessors and $DEP[\ell, v]$ for the corresponding sampled departure-times, per destination vertex v, given ℓ.

The algorithm's pseudocode is provided below.

The CTRAP approximation algorithm (pseudocode) Now, we present a more detailed description of CTRAP. The pseudocode is the following:

procedure CTRAP(ℓ)

1: **for** $v \in V$ **do** $\{\ ACTIVE[\ell, v](0, T) = TRUE\ \}$; $\tau_{old} = T;\ \tau = 3200$ /* initialization */
2: **while** $\exists v \in V, \exists k \in [0, T) :\ ACTIVE[\ell, v](k\tau_{old}, (k+1)\tau_{old}) == TRUE$ **do**
3: Sample (not already sampled) shortest-path trees rooted at ℓ, for all departure times
$k\tau \in [0, T)$ /* $PRED[\ell, v](k\tau)$ indicates v's parent in the tree for $(\ell, k\tau)$... */
4: **for** $v \in V \wedge k : k\tau \in [0, T)$ **do** /* looking for still active destinations... */
5: **if** $ACTIVE[\ell, v](k\tau, (k+1)\tau) == TRUE$ **then**
6: **if** $DEP[\ell, v].NotInSequence(kt)$ **then**
7: $position = DEP[\ell, v].SortedInsertion(k\tau)$;
8: $PRED[\ell, v].Insertion(parent[\ell, v](k\tau), position)$
9: **end if**
10: **if** $MAE[\ell, v](k\tau, (k+1)\tau) == TRUE$ **then** $\{\ ACTIVE[\ell, v](k\tau, (k+1)\tau) = FALSE\ \}$
11: **end if**
12: **end for**
13: $\tau = \tau/2;\ \tau_{old} = 2\tau$
14: **end while**
15: **for** $v \in V$ **do**
16: **repeat** /* merge intervals with the same predecessor... */
17: **for** ***consecutive*** records $(PRED[\ell, v](t), t)$ and $(PRED[\ell, v](t'), t')$ such that $PRED[\ell, v](t) == PRED[\ell, v](t')$ **do**
18: $PRED[\ell, v].Delete(PRED[\ell, v](t'))$
19: $DEP[\ell, v].Delete(t')$
20: **end for**
21: **until** $PRED[\ell, v]$ does not have identical consecutive records.
22: **end for**
23: **for** $u, v \in V : DEP[\ell, v] == DEP[\ell, u]$ **do**
24: Store the departure-times sequence $DEP[\ell, v]$ only once, for the (unique) representative v, and store the sequence $PRED[\ell, u]$ of corresponding predecessors for all non-representative destinations u.
25: **end for**

We now comment on the data types used. For a given landmark vertex ℓ, a destination vertex v, and a subinterval $[t_s, t_f) \subseteq [0, T)$, the flag $ACTIVE[\ell, v](t_s, t_f)$ declares whether the upper-approximation $\overline{\delta}[\ell, v]$ considered by CTRAP (cf. Figure 2.1) is satisfactory, given the required approximation guarantee that we consider. The variable τ determines the current step of the sampled departure-times from ℓ. $PRED[\ell, v]$ and $DEP[\ell, v]$ are the sequences of predecessors and (corresponding) departure-times from ℓ, w.r.t. the destination vertex v. We assure that $DEP[\ell, v]$ is always ordered in increasing departure-time values. This is done by assuming the operation $DEP[\ell, v].SortedInsertion(x)$ which places x in the right position, which is then returned by the procedure. As for $DEP[\ell, v]$, we consider the insertion of a new element u at an arbitrary position pos, $DEP[\ell, v](u, pos)$. It is mentioned at this point that these operations have been implemented in a rather straightforward manner (essentially performing linear scans on the queues), leaving for the future the consideration of more sophisticated implementations. The boolean function $MAE[\ell, v](t_s, t_f)$ determines whether the maximum-absolute-error test is satisfied for v, in the interval $[t_s, t_f)$. In particular, since we already have sampled all the travel-times at t_s and t_f, for a given approximation guarantee $\varepsilon > 0$ we perform the following test, which is a sufficient condition for $\overline{\delta}[\ell, v]$ being a $(1+\varepsilon)$-upper-approximation of $D[\ell, v]$ within $[t_s, t_f)$:

procedure MAE$[\ell, v](t_s, t_f)$

1: **if** $\min\{D[\ell, v](t_s), D[\ell, v](t_f)\} \geq \left(1 + \frac{1}{\varepsilon}\right) \Lambda_{\max}$ **then** **return** (TRUE)
2: **else** **return** (FALSE)

3.3.1 Preprocessing space and time reduction

Next, we describe in more detail the major algorithmic improvements of CTRAP towards a significant reduction in the required preprocessing space and time.

Omit intermediate breakpoints between consecutive sampled breakpoints. TRAP computes "intermediate" breakpoints $(\overline{t}_m, \overline{D}_m)$ between consecutive (sampled) breakpoints of $D[\ell, v]$, as the intersection points of the two non-constant legs involved in the definition of the upper-approximating function $\overline{\delta}[\ell, v](t)$, for each landmark ℓ and destination v. All these intermediate points are stored by FLAT, for each interval between consecutive sampled points where the MAE is satisfied. 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, per landmark we avoid the storage of approximately 10M for Berlin, and 100M intermediate breakpoints for Germany.

Store trees, rather than functions. For each leg of the travel-time summary $\overline{\Delta}[\ell, v]$, we choose in CTRAP to store pairs $\langle t_\ell, PRED[\ell, v](t_\ell)\rangle$ of departure times t_ℓ from ℓ and the immediate predecessor of v in the corresponding shortest-path tree rooted at (ℓ, t_ℓ), omitting the actual min-travel-time values $D[\ell, v](t_\ell)$ of the sampling procedure. This modification makes the oracle aware only of the shortest-path-tree structures created during the repeated sampling procedure, instead of the travel-time summaries stored by TRAP (as sequences of breakpoints).

Additionally, rather than storing repeatedly the IDs of predecessors, which would be space consuming in a network of millions of vertices, CTRAP stores only the position of the corresponding arc in the list of incoming arcs to a vertex v. Since the maximum in-degree in a road network 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. E.g., by choosing a scaling factor $s = 10.547$sec, each departure-time value would require 13 bits for its own representation. 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 those nodes, we simply 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.

Avoid duplicate records of common departure-time sequences. We now exploit the fact that CTRAP is a repeated-sampling method which probes (at common departure-times for all destinations) shortest-path trees from a given 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 immediate predecessors. The crucial observation is that destinations which are (roughly) at the same distance from ℓ would be expected 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 choose to store each sequence $DEP[\ell, v]$ as soon as it first appears, for some destination vertex v, and consider v as the ***representative destination*** of all other destinations u for which $DEP[\ell, u] = DEP[\ell, v]$. For each non-representative destination u, we only store $PRED[\ell, u]$ and their representative v for which it holds that $DEP[\ell, u] = DEP[\ell, v]$.

A challenging aspect is how to efficiently compare a newly created departure-times sequence with the already stored ones. In order to avoid a potential blow-up of the preprocessing time, we do not compare these subsequences point-by-point, in order to discover whether they are identical or not. Instead, we assign to every sampled departure-time t_ℓ two independently and uniformly at random 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, 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.

Merge sequences of breakpoints with identical predecessors. Our next algorithmic intervention in `CTRAP` is based on our 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 break-

points 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 significant reduction in the number of breakpoints to store, but also has a negative influence (reduction of similarities) on the departure-times sequences, and thus on the amount of repetitions that we could exploit. However, there is still a significant gain by applying both the aforementioned heuristic and that of Section 3.3.1.

In overall, all the above mentioned algorithmic improvements resulted in a reduction of 88% in the preprocessing space requirements of CFLAT, compared to the requirements of FLAT reported in [27].

Indexing preprocessed information. For retrieving efficiently the required preprocessed information (summary) 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 consumes $\mathcal{O}(n \cdot L)$ additional bytes.

Speeding up preprocessing time. Handling shortest-path trees, rather than minimum-travel-time functions, 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 shortest-path summaries of CTRAP are created without having to compute earliest-arrival functions. This leads to a reduction in the preprocessing time of approximately 35%, compared to FLAT.

3.3.2 The Query Algorithm

We now present our novel query algorithm CFCA(N) for responding to arbitrary queries $(o, d, t_o) \in V \times V \times [0, T)$, which efficiently exploits the time-dependent shortest-path trees data structure computed during the preprocessing phase. The algorithm is based on FCA$^+(N)$ presented in [27], but is fundamentally different from it in the sense that it now has to exploit shortest-path trees (rather than travel-time functions)

and also considers the construction of the proposed od-path to be part of it, which was not accounted for in the performance of $\mathsf{FCA}^+(N)$, and indeed of most of the shortest-path query algorithms in the literature. The parameter N indicates the number of landmarks to be settled by the query algorithm around the origin o, before specifying the particular od-path to recommend. In particular, $\mathsf{CFCA}(N)$ consists of three main steps:

procedure $\mathsf{CFCA}(N)$

STEP 1: A TDD ball is grown from (o, t_o), until N landmarks are settled.

STEP 2: Starting from d, recursively mark arcs from their predecessors to all the intermediate vertices, along od-paths indicated per pair of a settled landmark and its corresponding arrival-time at it, until explored (but not necessarily settled, yet) vertices from STEP 1 are reached.

STEP 3: Return an od-path by running TDD on the subgraph induced by all the marked arcs.

Recall that, at the end of STEP 1, we do not have at our disposal travel-time summaries, from a settled landmark ℓ towards d, or any other vertex. As a consequence, we are not able to compare ℓv-paths based on their (approximate) lengths. We only possess time-dependent information about predecessors in the appropriate shortest-path trees from these landmarks. On the other hand, for each departure-time t_ℓ from ℓ, and any vertex v, we can tell the predecessor(s) of v in the (at most two) most relevant shortest-path trees.

3.3.3 Detailed description of the actual path construction

In case that the destination was discovered in the first step, the resulting (exact) od-path can be computed by a simple backtracking towards the origin, following the pointers to all predecessors. Otherwise, the algorithm takes into account the preprocessed information provided by the N settled landmarks as follows. Given a settled landmark ℓ, for which we already have a shortest $o\ell$-path that guarantees arrival-time $t_\ell = D[o, \ell](t_o)$ at ℓ, we repeatedly ask the oracle for predecessors of intermediate vertices v in the most relevant preprocessed shortest-path tree(s) determined at the endpoints of the corresponding to the leg containing t_ℓ, starting with $v = d$. The goal is to eventually mark a small subset of vertices that contain ***some*** od-path, not

necessarily passing by ℓ, but being "oriented" towards ℓ.

Clearly, the predecessor of a vertex v, w.r.t. ℓ may vary with time. Since the departure-time t_ℓ from ℓ is a continuous variable in $[0, T)$, at most two different predecessors of v may be proposed by the oracle w.r.t. (ℓ, t_ℓ), depending on the subinterval $[t_s, t_f)$ to which t_ℓ belongs.

CFCA(N) marks (per landmark) the connecting arcs from the most relevant (one or two) predecessor(s) of v to v. All the discovered predecessors w.r.t. the N settled landmarks are inserted in a FIFO queue, which was initiated with d, so that, upon their extraction from the queue, they can provide in turn their own predecessors (to be added to the queue, if they have not been processed already).

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 by 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 STEP1.

The path construction takes place in STEP 3, which considers the subgraph induced by the marked arcs and grows a TDD ball from (o, t_o) in this subgraph. Indeed, we continue growing the TDD ball from STEP 1, which already possesses all the explored nodes reached so far, but only considering the marked arcs from now on. This path construction indeed leads to significantly smaller relative errors, since the resulting shortest-path is not only the best choice among a given set of N paths induced by the N settled landmarks (as in FLAT), but the actual shortest od-path within the induced sugbgraph.

Chapter 4

Experimental Evaluation

4.1 Experimental Setup

All algorithms were implemented using C++ (gcc version 4.8.2).

To support graph-operations we used the PGL library [28]. This graph structure consists of three packed-memory arrays, one for the nodes and two for the edges of the graph (viewed as either outgoing or incoming) with pointers associating them. The two edge arrays are copies of each other, with the edges sorted as outgoing or incoming in each case. PGL supports the following features: i) Compactness: ability to efficiently access adjacent nodes or edges, a requirement of all speed-up techniques based on Dijkstra's algorithm, ii) Agility: ability to change and reconfigure its internal layout in order to improve the locality of the elements, according to a given algorithm, and iii) Dynamicity: ability to efficiently insert or delete nodes and edges.

The experiments were executed on an Intel(R) Xeon(R) CPU E5-2643v3 3.40GHz using 128GB of RAM and Ubuntu 14.04 LTS. We used 6 threads for the parallelization of the preprocessing phase and the adaptation of the preprocessed information to live traffic incidents. The query algorithms were executed on a single thread.

The experimental evaluation of FLAT, HORN and CFLAT was conducted on two real-world instances, one concerning the metropolitan area of the city of Berlin, and the other concerning the road network in Germany.

4.2 Benchmark Instances

We report here the details of the instances of Berlin and Germany, on which we have conducted the experimental evaluation of our oracles.

Berlin Instance The instance of Berlin consists of $473,253$ nodes and $1,126,468$ arcs. The contraction of the road network created $183,468$ shortcuts. Whenever more than one contracted paths shared the same endpoints, we added only one shortcut representing all these contracted paths. There were 914 such cases in the Berlin instance. The contracted paths that could be represented by an original arc in the graph, are $11,398$ in total. In overall, the contraction of Berlin led to a graph of $292,356$ active vertices and $752,362$ active arcs.

Germany Instance The instance of Germany consists of $4,692,091$ nodes and $11,183,060$ arcs. After the instance-preprocessing phase we got an instance with $3,431,213$ active vertices and $11,554,840$ active arcs. The total number of the added shortcuts was $4,595,148$. We avoided the insertion of additional shortcuts in $106,464$ cases, where $6,816$ of them correspond to "parallel" shortcuts and the $99,648$ correspond to the existence of actual arcs connecting the endpoints of contracted paths.

4.3 Experimental Evaluation of FLAT and HORN.

We report here the outcome of our experiments on the instance of Berlin.

4.3.1 FLAT @ Berlin.

Tables 4.1 and 4.2 summarize the performance of the basic query algorithms of FLAT with respect to absolute running times and Dijkstra rank values, respectively, for landmark sets of various sizes. The best performance is indicated by highlighted

table cells. The last four lines in each table are for the sake of comparison of FLAT with HORN (see Section 4.3.2). As for the query algorithms, we used recursion budget 1 for RQA and we let FCA$^+$ settle the 6 closest landmarks, which is roughly the average number of settled landmarks by RQA as well.

Landmark Selection Policies.

R is for uniform and random landmark selection. K is for selecting the boundary vertices of a KAHIP partition as landmarks. We have used the version v0.71 of the KAHIP partitioning software, exploiting the KaFFPa algorithm, with the following parameters: The number of blocks to partition the graph was set to 178, so that we get (slightly more than) $2,000$ landmarks. H is for a hybrid partition that initially creates a KAHIP partition (with half the landmarks) and then randomly chooses additional landmarks within each cell of the partition. IR indicates a variant of R that moves each randomly selected landmark to its closest important node. We have considered as "important" those nodes in the Berlin instance which are incident to road segments of category at most 3. SR indicates another variant of randomly selected landmarks, where each newly chosen random landmark excludes its closest 300 nodes (under the free-flow metric) from being landmarks in the future.

4.3.2 HORN @ Berlin.

Due to large space requirements, we could handle landmark hierarchies with up to $21,000$ landmarks for the Berlin instance, which seems to be harder than that of Germany[1]. The average size per landmark in the hierarchy is 2.1MBytes. For a hierarchy of $10,443$ landmarks the preprocessing of HORN took 5.1 hours, for $20,886$ landmarks it took 10.3 hours, or at most 1.8sec per landmark in either case. The landmarks in each level of the hierarchy were chosen by the RANDOM (HR) and SPARSE-RANDOM (HSR) methods. We consider 4 levels of the hierarchy, according to the sizes of the landmarks' ***areas of coverage***, i.e. the number of "nearby" destinations for which they possess travel-time summaries. The area of coverage for landmarks of

[1]We observed that the speedups are significantly better in Germany, despite the fact that we consider the same number of landmarks in a larger, by an order of magnitude, network. This is probably due to stronger correlation of time-dependence among different road segments in an urban environment, rather than in a nationwide road network.

level 4 is actually the entire graph. These are exactly the "global" landmarks which the corresponding variant of FLAT would also consider. The landmarks of the other levels have significantly smaller areas of coverage.

In overall, we created four distinct hierarchies, with $10,443$ and $20,886$ landmarks, based on HR and HSR landmark selection methods (cf. Table 4.5). The corresponding variants for FLAT possess the same (274 and 548, respectively) "global" landmarks. We created travel-time summaries using only the TRAP approximation method, within the subgraph induced by each landmark's area of coverage (extended, as in Assumption 2.3.3).

The experimental results of FCA for R_{274}, SR_{274}, R_{548} and SR_{548} are shown in Tables 4.1 and 4.2. Table 4.4 summarises the experimental evaluation of HQA. Interestingly,its performance with HR-landmarks is better than that with HSR-landmarks, probably because the ESC criterion seems more effective in the former case. Table 4.6 summarizes the comparison of HORN with FLAT. The results for HR_{10443}, HSR_{10443}, HR_{20886} and HSR_{20886} are compared with the corresponding results of FCA for R_{274}, H_{274}, R_{548} and H_{548} (i.e., with the same number of global landmarks), respectively. There is a significant improvement in query performance (e.g., more than 41% w.r.t. Dijkstra ranks), but also in quality of the produced solution (by more than 29%), at the cost of increasing the space requirements by a factor of 6.34 at most.

4.3.3 Detailed Experimental Results

	TDD		FCA		FCA$^+(6)$		RQA	
	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %	Time (msec)	Rel.Error %
R_{2000}	73.402	0	0.130	0.964	0.692	0.448	0.677	0.669
K_{2000}			0.158	1.076	0.513	0.361	0.410	0.554
H_{2000}			0.154	0.907	0.714	0.331	0.636	0.631
IR_{2000}			0.105	0.894	0.583	0.346	0.561	0.585
SR_{2000}			0.085	0.717	0.574	0.321	0.446	0.604
R_{548}			0.384	1.838	2.365	0.736	2.348	1.629
SR_{548}			0.553	1.545	2.456	0.562	2.498	1.529
R_{274}			0.943	2.309	4.008	0.843	4.929	2.348
SR_{274}			0.729	2.091	4.288	0.747	4.271	2.256

Table 4.1: Performance of FCA, FCA$^+(6)$ and RQA, w.r.t. the ***running times*** and ***relative errors***, at 2.64sec resolution, for a query set of $10,000$ ***random queries*** in Berlin.

	TDD		FCA		FCA$^+(6)$		RQA	
	Rank	Speedup	Rank	Speedup	Rank	Speedup	Rank	Speedup
R_{2000}	147,904	1	155	954.219	904	163.610	971	152.321
K_{2000}			191	774.366	858	172.382	665	222.412
H_{2000}			168	880.381	880	168.072	829	178.413
IR_{2000}			135	1,095.585	830	178.197	851	173.800
SR_{2000}			121	1,222.347	957	154.549	788	187.695
R_{548}			548	269.898	3,250	45.509	3,449	42.883
SR_{548}			604	244.874	3,670	40.300	3,843	38.486
R_{274}			1,174	125.983	6,367	23.229	6,835	21.639
SR_{274}			1,184	124.919	7,045	20.994	7,359	20.098

Table 4.2: Performance of FCA, FCA$^+(6)$ and RQA, w.r.t. ***Dijkstra ranks***, at 2.64sec resolution, for a query set of $10,000$ ***random queries*** in Berlin.

Level	Size of Levels		Area of coverage	Excluded Ball Size (for HSR)	
	$\|L\| = 10,443$	$\|L\| = 20,886$		$\|L\| = 10,443$	$\|L\| = 20,886$
L_1	$7,832$	$15,664$	$1,292$	35	15
L_2	$1,630$	$3,260$	$29,841$	150	80
L_3	707	$1,414$	$158,535$	350	180
L_4	274	548	$299,693$	800	400

Table 4.3: Landmark hierarchies for HORN, based on HR and HSR landmark selection methods, for the Berlin instance.

	TDD				HQA			
	Time (msec)	Rel.Error %	Rank	Speedup	Time (msec)	Rel.Error %	Rank	Speedup
HR_{10443}	73.402	0	147,904	1	0.532	1.606	664	222.747
HSR_{10443}					0.550	1.432	686	215.603
HR_{20886}					0.241	1.179	314	471.032
HSR_{20886}					0.304	1.096	355	416.631

Table 4.4: Performance of HQA, w.r.t. the ***running times***, ***relative errors*** and ***Dijkstra ranks***, at 2.64sec resolution, for a query set of $10,000$ ***random queries*** in Berlin.

Level	Size of Levels		Area of coverage	Excluded Ball Size (for HSR)	
	$\|L\| = 10,443$	$\|L\| = 20,886$		$\|L\| = 10,443$	$\|L\| = 20,886$
L_1	$7,832$	$15,664$	$1,292$	35	15
L_2	$1,630$	$3,260$	$29,841$	150	80
L_3	707	$1,414$	$158,535$	350	180
L_4	274	548	$299,693$	800	400

Table 4.5: Landmark hierarchies for HORN, based on HR and HSR landmark selection methods, for the Berlin instance.

Change in:	Query Time (%)	Error (%)	Dijkstra Rank (%)	Space (in times)
R_{274} vs HR_{10443}	43.58	30.44	43.44	6.057
H_{274} vs HSR_{10443}	24.55	31.51	42.06	6.333
R_{548} vs HR_{20886}	37.23	35.85	42.70	6.298
H_{548} vs HSR_{20886}	45.02	29.06	41.22	6.283

Table 4.6: Comparison of HORN and FLAT oracles for the instance of Berlin.

4.3.4 Live Traffic Reporting with FLAT.

In a server-side routing service that responds to several queries in real-time, various disruptions may occur "on the fly" (e.g., the abrupt and unforeseen congestion, or even blockage of a road segment for half an hour due to a car accident) and have to be taken into account for the affected route plans that have already been suggested or will be suggested in the near future. We thus consider dynamic scenarios where there is a stream of live-traffic reports about abnormal delays on certain road segments (arcs), along with a time-window $[r_s, r_e]$, of typically small duration, in which the disruption occurs.

Our update step involves the recomputation of travel-time summaries for a subset of landmarks in the vicinity of the disruption. In particular, for a disrupted arc $a = uv$ of disruption duration $[r_s, r_e]$, we run a `Backward-Dijkstra` from u under the free-flow metric, with travel time radius of at most $r_e - r_s$. The limited travel time radius is used to trace only the nearest landmarks that may actually be affected by the disruption, leaving unaffected all the "faraway" landmarks. The goal is to update as soon as possible the recommendations for the drivers who are close to the area of disruption. For each affected landmark ℓ, we consider a ***disruption-times window*** $[t_s, t_e]$, containing the latest departure times from ℓ for arriving at the tail u at any time in the interval $[r_s, r_e]$ in which the disruption occurs. We then compute ***temporal*** travel-time summaries for each affected landmark and disruption-times window. This computation is conducted as in the preprocessing phase. Using a 15min radius for the disruptions, we ran $1,000$ live-traffic updates for the instance of Berlin, with $2,000$ SR-landmarks, using 6 threads. The average number of affected landmarks was 86 and the corresponding update time for their preprocessed data was 32.376sec.

4.4 Experimental Evaluation of CFLAT.

In the following, we present the extensive experimental evaluation of the improved distance oracle `CFLAT`.

4.4.1 Evaluation of CFLAT @ Berlin

Landmark Selection Policies.

We start with a systematic naming of the chosen landmark sets that we consider. Each set is encoded as XY, where $X \in \{R, SR, IR, SK, KC, BC, KB\}$ determines the type of landmark set, and $Y \in \{250, 500, 1K, 2K, 4K, 8K, 16K\}$ determines its size. For all types we have considered $Y = 4K$ meaning that we chose $4,000$ landmarks. Especially for R we have tried all possible values for Y, in order to showcase the scalability of CFLAT and its smooth trade-off of preprocessing requirements, query-times and stretch factors. We now proceed with a clarification of the considered types of landmark sets. $\{R, SR, IR, SK\}$ were also considered in [27], whereas $\{KC, BC, KB\}$ are new landmark types. Additionally, 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 the SK, KC and KB landmark sets we used the following parameters for KAHIP: 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, either randomly or according to importance, one landmark per cell.

Preprocessing Requirements.

We start with the presentation of the preprocessing requirements for the construction of shortest-path summaries for CFLAT, for various sizes of random (R) landmark sets (cf. Figure 4.1). The requirements for other landmark types are analogous. For this preprocessing, we have used 6 parallel threads. It is worth mentioning that FLAT [27] required ***uncompressed*** preprocessing space 43GB, or equivalently, ***compressed*** size of 14MB per landmark, and 33h to preprocess $R2K$. CFLAT preprocesses $R16K$ in 26h consuming 41.4GB, and $R2K$ in 3.5h consuming only 5.2GB.

Performance of CFCA(N).

We measured the performance of CFCA(N), for $N \in \{1, 2, 4, 6\}$, on a set of $50,000$ queries (o, d, t_o) generated independently and uniformly at random from $V \times V \times [0, T)$.

We first conducted an experiment with various sizes of R-landmark sets, to test the scalability of our oracle's query-time and relative error, as a function of the

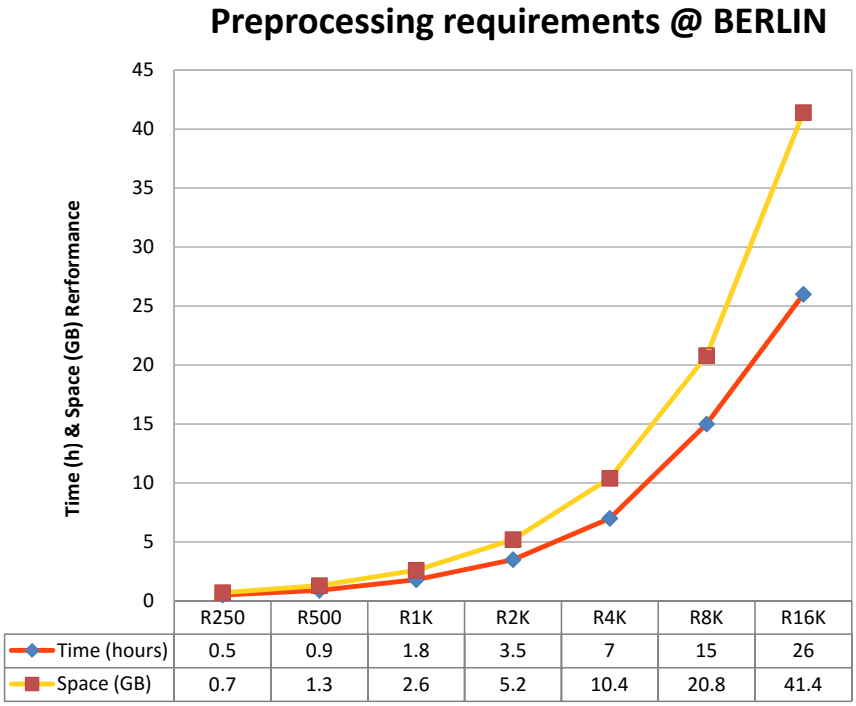
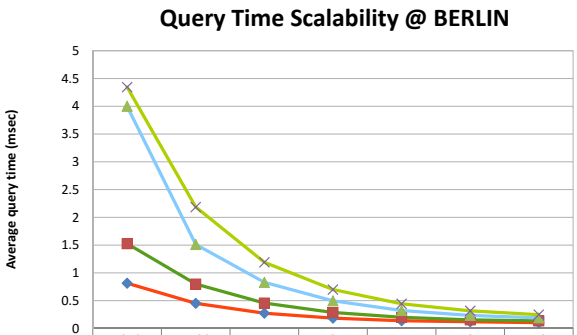


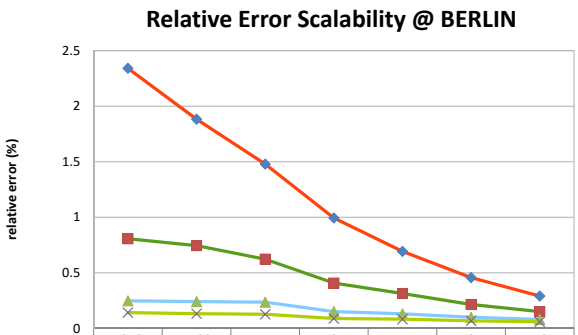
Figure 4.1: Preprocessing requirements for Berlin.

number of landmarks. Figure 4.2 demonstrates the results of this experiment. As is evident from this figure, the relative errors decrease linearly and the running times decrease quadratically, as we double the number of landmarks. It is also mentioned that remarkable relative errors of less than 0.142% are achieved for $\texttt{CFCA}(6)$ even with only 250 landmarks which require only 700MB, with running time less than 4.346msec. Moreover, a "quick-and-dirty" answer of average error at most 2.341% is returned in only 0.814msec. The minimum query-times and relative errors are achieved for $R16K$, where $\texttt{CFCA}(1)$ has running time 0.103msec and relative error 0.291%, whereas $\texttt{CFCA}(6)$ has running time 0.247msec and relative error 0.059%.

Our next experiment compared different types of landmark sets. We chose various landmark sets of size $4,000$, and tried $50,000$ random queries. Figure 4.3 summarizes the performance of $\texttt{CFCA}(N)$ w.r.t. average query-times and relative errors. The average query-time for TDD was 131.873msec, implying (e.g., for $BC4K$) a speedup of more than $1,030$ with average stretch 0.536%. That is, the query-time is only slightly worse (0.128msec instead of 0.083msec) even though the measurement in this work includes also the path construction, but the corresponding average error is clearly better than the average error of 0.781% for $\texttt{FCA}(N)$ [27]. Note that TDD is executed here on the original instance (even before the vertex contraction), whereas in [27] it was executed



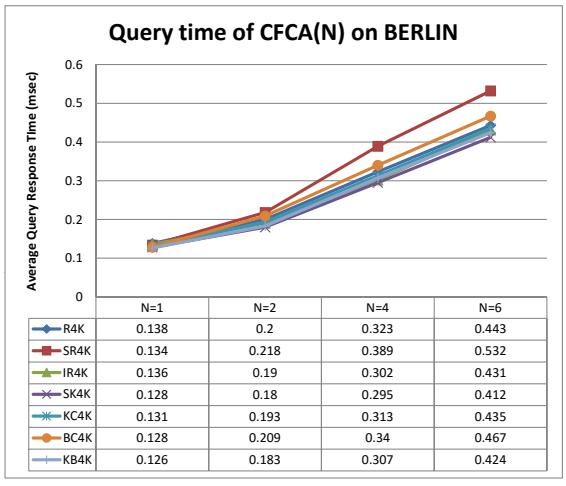
	R250	R500	R1K	R2K	R4K	R8K	R16K
CFCA(1)	0.814	0.452	0.274	0.185	0.138	0.12	0.103
CFCA(2)	1.526	0.799	0.457	0.287	0.2	0.154	0.136
CFCA(4)	4	1.512	0.83	0.497	0.323	0.234	0.19
CFCA(6)	4.346	2.186	1.191	0.7	0.443	0.316	0.247



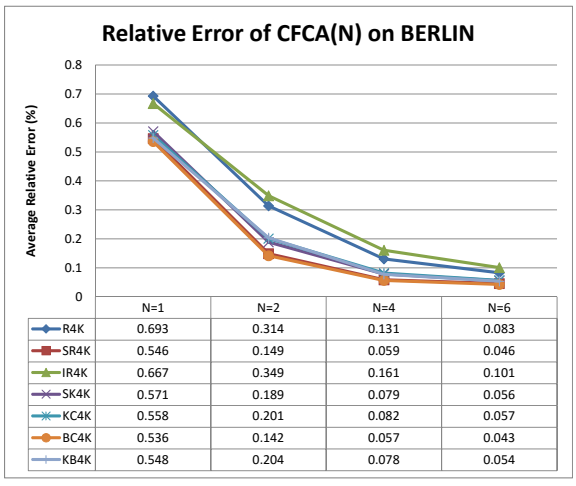
	R250	R500	R1K	R2K	R4K	R8K	R16K
CFCA(1)	2.341	1.883	1.479	0.994	0.693	0.457	0.291
CFCA(2)	0.807	0.745	0.623	0.408	0.314	0.215	0.151
CFCA(4)	0.248	0.241	0.236	0.151	0.131	0.1	0.08
CFCA(6)	0.142	0.132	0.127	0.089	0.083	0.068	0.059

Figure 4.2: Average ***query time*** (in msec) and ***relative error*** of CFCA(N), at 1.32sec resolution, for a query set of $50,000$ ***random queries*** in Berlin.

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. Concerning running times, the best curve is provided by $SK4K$, but only $SR4K$ seems to be really worse than the others. As for relative errors, $SR4K$ and $BC4K$ are clear winners. Further experiments are reported in Section 4.4.4.



	N=1	N=2	N=4	N=6
R4K	0.138	0.2	0.323	0.443
SR4K	0.134	0.218	0.389	0.532
IR4K	0.136	0.19	0.302	0.431
SK4K	0.128	0.18	0.295	0.412
KC4K	0.131	0.193	0.313	0.435
BC4K	0.128	0.209	0.34	0.467
KB4K	0.126	0.183	0.307	0.424



	N=1	N=2	N=4	N=6
R4K	0.693	0.314	0.131	0.083
SR4K	0.546	0.149	0.059	0.046
IR4K	0.667	0.349	0.161	0.101
SK4K	0.571	0.189	0.079	0.056
KC4K	0.558	0.201	0.082	0.057
BC4K	0.536	0.142	0.057	0.043
KB4K	0.548	0.204	0.078	0.054

Figure 4.3: Performance of CFCA(N), w.r.t. the average ***query times*** (in msec) and ***relative errors***, at 1.32sec resolution, for a query set of $50,000$ ***random queries*** in Berlin.

4.4.2 Evaluation of CFLAT @ Germany

Landmark Selection Policies.

CFLAT was tested with the landmark sets $R3K$, $SR3K$ with excluded neighborhood size $1,200$, $SK3K$ with excluded neighborhood size 350, and $BC3K$ with excluded neighborhood size $1,000$.

Preprocessing Requirements for Germany

The preprocessing requirements for the shortest-path summaries of CFLAT in Germany, for various sizes of R-landmark sets, are shown in Figure 4.4. The requirements for other landmark types are analogous. CFLAT creates the preprocessed data in 32.2h requiring (uncompressed) space 53.6GB. This indeed made it possible to consider landmark sets of size up to $4,000$ in this work.

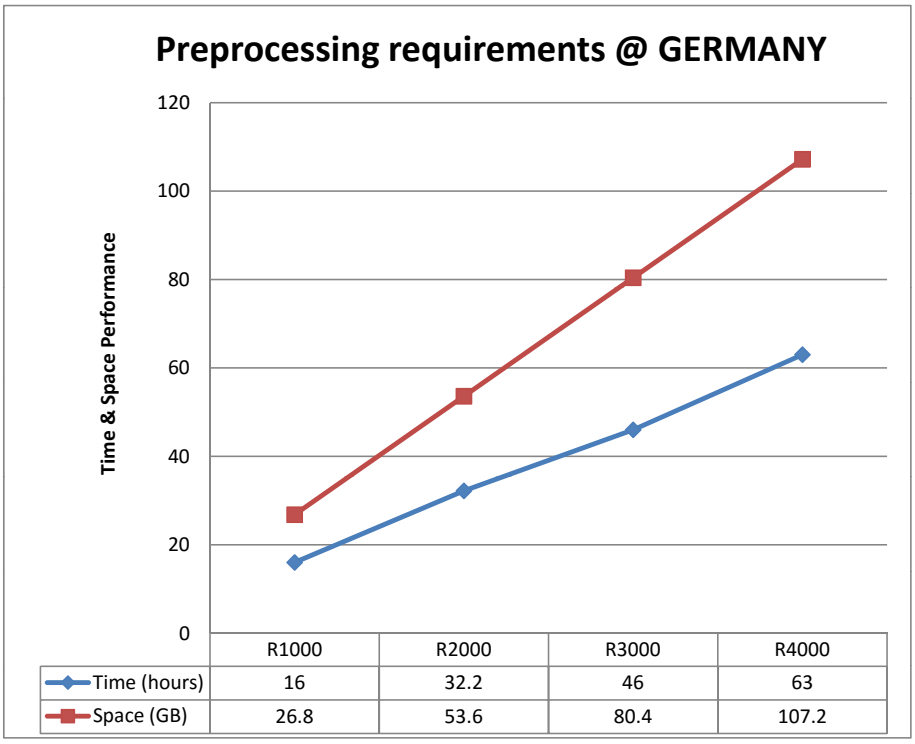


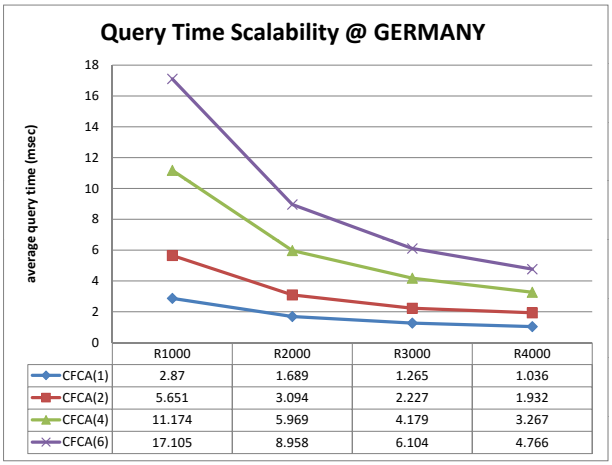
Figure 4.4: Preprocessing requirements of Germany.

Performance of CFCA(N).

We started with an experiment on various sizes of R-landmark sets, to test the scalability of CFCA(N)'s query-time and relative error, as a function of N. Figure 4.5 demon-

strates the results of this experiment. Once more, the relative errors decrease linearly and the running times decrease quadratically, as we double the number of landmarks. Remarkable relative errors of less than 0.072% are achieved for CFCA(6) even with $1,000$ landmarks which require 26.8GB, with running time less than 17.105msec. Moreover, a "quick-and-dirty" answer of average error at most 1.562% is returned in only 2.87msec. The best query-times and relative errors are achieved for $R4K$, where CFCA(1) has running time 1.036msec and relative error 0.912%, whereas CFCA(6) has running time 4.766msec and relative error 0.053%.

We proceed next with comparing various types of landmark sets, w.r.t. the performance of CFCA(N) in absolute times and relative errors (cf. Figure 4.6). For Germany we have a clear winner, $BC3K$, w.r.t. both running times and relative errors. Since the average running time of TDD is $1,421.12$msec, we get for CFCA(1) an average query-time of 0.959msec (i.e., speedup by more than $1,481$) with average relative error 0.911%, and for CFCA(6) a query-time of 5.506msec (i.e., speedup by more than 258) with relative error 0.032%. Further experiments are reported in Section 4.4.4.



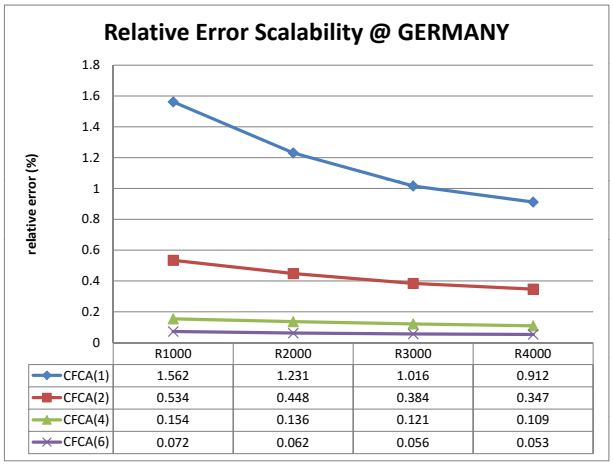
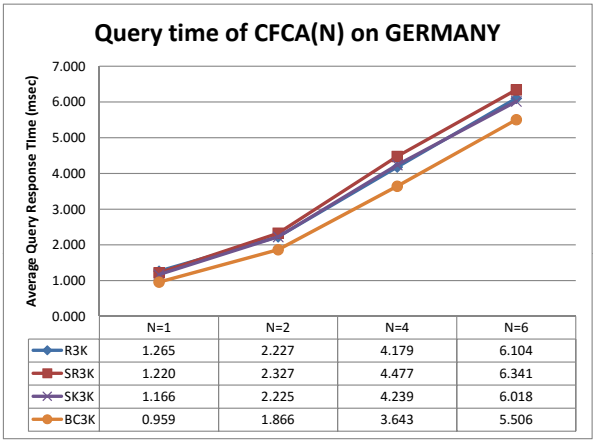
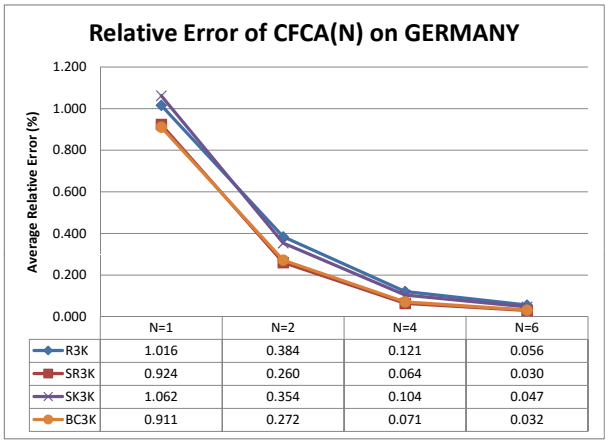


Figure 4.5: Average ***query time*** (in msec) and ***relative error*** of CFCA(N), at 1.32sec resolution, for a query set of $50,000$ ***random queries*** in Germany.



	N=1	N=2	N=4	N=6
R3K	1.265	2.227	4.179	6.104
SR3K	1.220	2.327	4.477	6.341
SK3K	1.166	2.225	4.239	6.018
BC3K	0.959	1.866	3.643	5.506



	N=1	N=2	N=4	N=6
R3K	1.016	0.384	0.121	0.056
SR3K	0.924	0.260	0.064	0.030
SK3K	1.062	0.354	0.104	0.047
BC3K	0.911	0.272	0.071	0.032

Figure 4.6: Performance of CFCA(N), w.r.t. average ***query times*** (in msec) and ***relative errors***, at 1.32sec resolution, for a query set of $50,000$ ***random queries*** in Germany.

4.4.3 Preprocessing Statistics for Berlin and Germany

Table 4.7 reports some significant preprocessing statistics. In particular, the measurements are the following: ***(i)*** the average number of vertices per landmark whose predecessor remains constant on the shortest-path tree throughout the whole time period, ***(ii)*** the remaining vertices with pwl behaviour w.r.t. their predecessor, ***(iii)*** the average number of unique departure-time sequences stored, instead keeping one sequence per destination with pwl predecessor, and ***(iv)*** the average number of intermediate points of TRAP per landmark, which we now avoid to store.

	Vertices with Unique Pred	Vertices with pwl Pred	Unique Departure Time Sequences	Intermediate Points of TRAP
R_{4K}	$272,286$	$20,070$	$5,963$	$10,663,125$
SR_{4K}	$272,287$	$20,069$	$5,831$	$10,688,275$
IR_{4K}	$272,284$	$20,072$	$5,781$	$10,672,869$
SK_{4K}	$272,282$	$20,074$	$6,011$	$10,934,712$
KC_{4K}	$272,287$	$20,069$	$5,857$	$10,758,955$
BC_{4K}	$272,293$	$20,063$	$5,858$	$10,728,776$
KB_{4K}	$272,300$	$20,056$	$5,432$	$10,643,285$

Table 4.7: Preprocessing statistics for CFLAT Oracle for Berlin.

Table 4.8 provides the preprocessing statistics related to Germany, in the same format as in the case of Berlin.

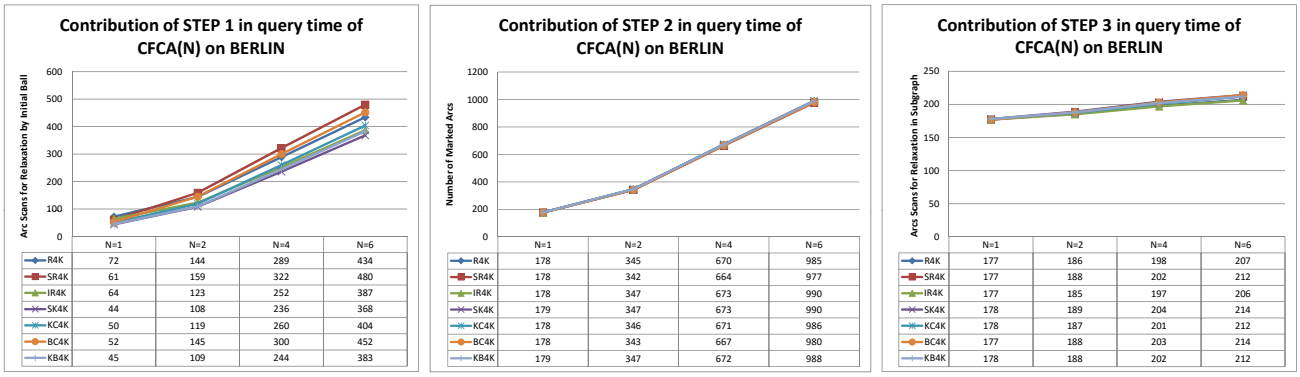
	Vertices with Unique Pred	Vertices with pwl Pred	Unique Departure Time Sequences	Intermediate Points of TRAP
$R3K$	$3,201,577$	$229,636$	$38,102$	$112,137,488$
$SR3K$	$3,201,642$	$229,571$	$37,212$	$112,081,032$
$SK3K$	$3,201,503$	$229,710$	$38,068$	$113,536,811$
$BC3K$	$3,201,637$	$229,576$	$37,207$	$112,067,442$

Table 4.8: Preprocessing statistics for CFLAT in Germany.

4.4.4 Detailed auditing of CFCA(N)'s computational effort

We also study how the total amount of computational effort is split among the major steps of the query algorithm. In particular, we measured the total number of touched

arcs (for relaxation or marking) per step.



	N=1	N=2	N=4	N=6
R4K	72	144	289	434
SR4K	61	159	322	480
IR4K	64	123	252	387
SK4K	44	108	236	368
KC4K	50	119	260	404
BC4K	52	145	300	452
KB4K	45	109	244	383

	N=1	N=2	N=4	N=6
R4K	178	345	670	985
SR4K	178	342	664	977
IR4K	178	347	673	990
SK4K	179	347	673	990
KC4K	178	346	671	986
BC4K	178	343	667	980
KB4K	179	347	672	988

	N=1	N=2	N=4	N=6
R4K	177	186	198	207
SR4K	177	188	202	212
IR4K	177	185	197	206
SK4K	178	189	204	214
KC4K	178	187	201	212
BC4K	177	188	203	214
KB4K	178	188	202	212

Figure 4.7: Per step performance of CFCA(N), at 1.32sec resolution, for a query set of $50,000$ ***random queries*** in Berlin.

Figure 4.7 gives these measurements of CFCA(N) in Berlin, i.e. the number of arcs checked for relaxation by the initial TDD-ball from (o, t_o) in step 1, the number of marked arcs connecting predecessors to intermediate vertices in step 2, and the number of arcs checked for relaxation during the extension of the TDD-ball within the marked subgraph, in order to provide the resulting od-path. It is clear from Figure 4.7 that only step 1 depends on the type of landmarks that we consider. Observe also that step 3 is essentially independent of the value of N, whereas the other two steps depend linearly on it. For sake of comparison, it is mentioned that the number of arcs checked for relaxation by TDD for the set of $50,000$ random queries in Berlin is $235,880$. This for example implies a speed-up, w.r.t. the machine-independent measure of "touched" vertices, of more than 588 for CFCA(1) and $SK4K$. Recall that the measurement does not only concern the estimation of an upper-bound on the earliest arrival-time at (or equivalently, the shortest travel-time towards) the destination, but also the explicit construction of the corresponding od-path that guarantees this bound. Observe also that in absolute running times the speed-up is almost double, because the computationally most demanding step 2 only concerns accesses to the preprocessed data and there is no need for handling priority queues. Moreover, step 3 only concerns a very limited subgraph, containing only a few hundreds of arcs in overall.

Figure 4.8 demonstrates this measurement for Germany. Again we observe the remarkable stability (and independence of the landmark set) for steps 2 and 3, as well as the linear dependence of steps 1 and 2, and the independence of step 3 on the value of N. Since the average number of touched arcs for TDD was $2,351,697$ vertices, the overall speedup of CFCA(1) for $BC3K$ is more than $1,666$ w.r.t. the machine

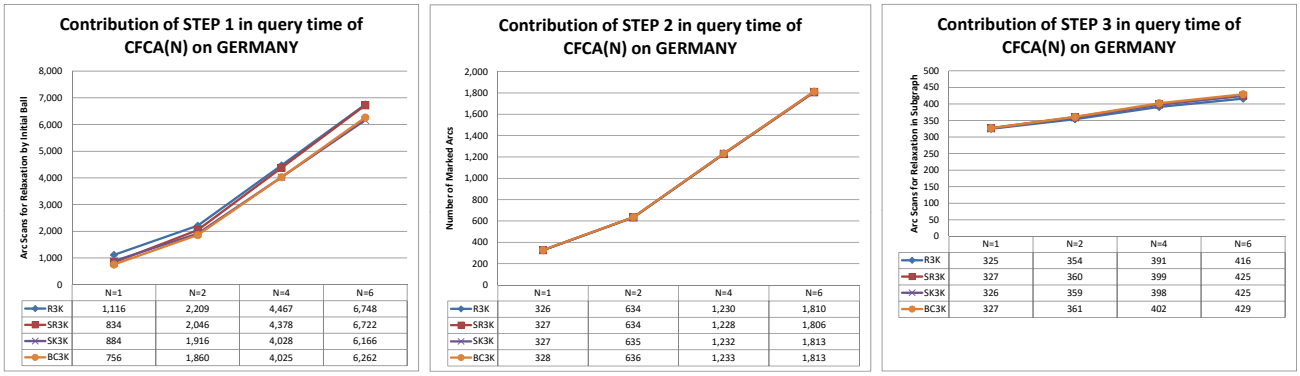


Figure 4.8: Per step performance of CFCA(N), at 1.32sec resolution, for a query set of $50,000$ ***random queries*** in Germany.

independent measure of number of "touched" arcs.

Observe finally that for Germany the speedups within the two measures (absolute running times, and "touched" arcs) are analogous. This is due to the fact that, since we have a quite small landmark set size this time, step 1 actually dominates the computational effort in this case.

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 with the most prominent landmark set $BC4K$.

In particular, taking the average error value as our baseline, we constructed buckets corresponding to multiples of this baseline, and then measured the percentage of queries resulting in a relative error belonging to each bucket. We chose $BC4K$ because this is the landmark set that results in the best trade-off of query time and relative error. Figure 4.9 demonstrates the results of this analysis for CFCA(1), CFCA(2), CFCA(4) and CFCA(6). We observe the following: CFCA(1) has an average relative error for is 0.536%, whereas 81.6% of total responses are below this average value. CFCA(2) has an average relative error 0.142%, and 90.7% of total responses are below this average value, and 94.2% of queries have relative error up to 0.568%. CFCA(4) has an average relative error 0.057%, and 97.3% of queries have relative error at most 0.456%. CFCA(6) has an average relative error 0.043%, and 97.6% of queries have relative error at most 0.344%. It is finally mentioned that for all values of N we faced the same worst-case error of 71.558%, which appeared for only one short-range query.

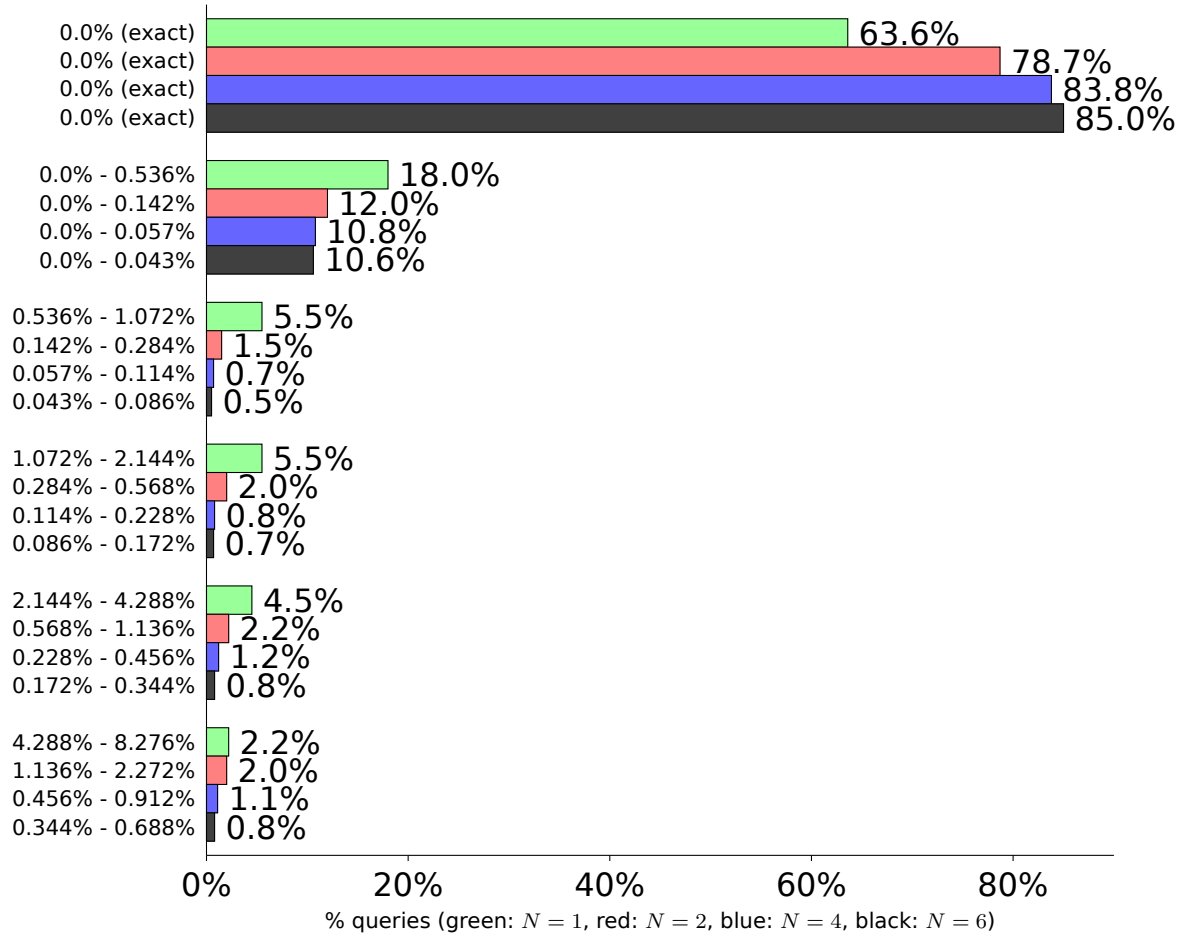


Figure 4.9: Tails of error percentages of $\texttt{CFCA}(N)$ for $50,000$ randomly chosen queries in the instance of Berlin, with the $BC4K$ landmark set.

Exploring Outliers in Relative Errors.

We conducted the same statistical analysis on the relative error of $\texttt{CFCA}(N)$ in Germany, as we did for Berlin. Figure 4.10 demonstrates the results of this analysis for $\texttt{CFCA}(1)$, $\texttt{CFCA}(2)$, $\texttt{CFCA}(4)$ and $\texttt{CFCA}(6)$. The outcome is slightly different compared to the Berlin instance. In particular, the vast majority of the queries result in relative errors below the average value, as we observed for Berlin, while some of them (but not most of them) provide the exact shortest-path. In more detail, for $BC3K$ in Germany we observe the following: The average relative error for $\texttt{CFCA}(1)$ is 0.911%. The algorithm discovers the exact shortest path in 23.4% of queries. 74.7% of total responses are below the average value. The worst-case error was 79.315%. The average relative error for $\texttt{CFCA}(2)$ is 0.272%. The algorithm discovers the exact shortest path in 35.3% of queries. 84.2% of total responses are below the average value. The worst-case error was 43.179%. The average relative error for $\texttt{CFCA}(4)$ is 0.071%. The algorithm

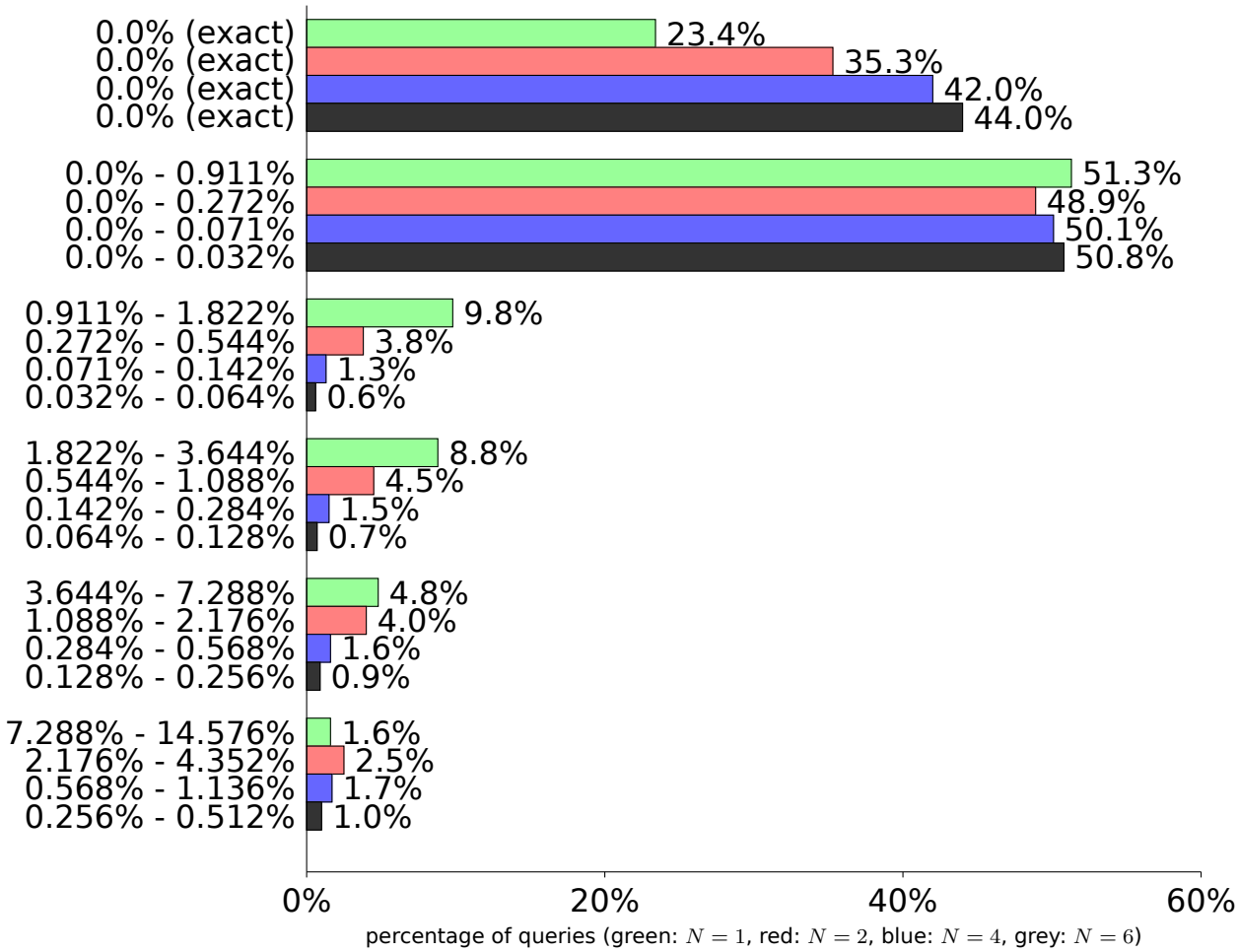


Figure 4.10: Tails of error percentages of CFCA(N) for $50,000$ randomly chosen queries in the instance of Germany, with the BC3K landmark set.

discovers the exact shortest path in 42.0% of queries. 92.1% of total responses are below the average value. The worst-case error was 26.840%. The average relative error for CFCA(6) is 0.032%. The algorithm discovers the exact shortest path in 44.0% of queries. 94.8% of total responses are below the average value. The worst-case error was 12.860%.

4.4.5 Live Traffic Reporting with CFLAT

As was done in the case of FLAT oracle, we conducted an experiment to assess the responsiveness of CFLAT to live-traffic updates. We remind that the goal is, when a disruption occurs "on the fly" (e.g., the abrupt and unforeseen congestion, or even blockage of a road segment for half an hour due to a car accident), that the oracle can take into account, for the affected route plans that have already been suggested

or will be suggested in the near future, the temporal traffic-related information, as fast as possible. Using a 15-min radius for the disruptions, we executed $1,000$ live-traffic updates for the instances of Berlin and Germany, for the landmark set $BC4K$ and $BC3K$, respectively. For Berlin, the average number of affected landmarks was 82 for Berlin, and the updating procedure of the affected landmarks' summaries requires average time 8.7sec, using 6 threads. For Germany, the average number of affected landmarks was only 6, and the updating procedure of the affected landmarks' summaries requires average time 11sec, again using 6 threads.

Chapter 5

Conclusion

We provided an extensive experimental evaluation of landmark-based oracles for time-dependent road networks.

Our experimentation of the FLAT oracle in the instance of Berlin has shown that the average query time can be as small as 85μsec, achieving a speedup (against the time-dependent variant of Dijkstra) more than 863 in absolute running times, and more than $1,222$ in Dijkstra rank measures, with worst-case observed stretch less than 0.717%.

It was proved that the advantage of HORN over FLAT is that it achieves query times sublinear, not just in the size of the network, but in the actual Dijkstra rank of the query at hand, be it long-range, mid-range, or short-range, while requiring asymptotically similar preprocessing space and time. Our experiments on the Berlin instance indeed confirm the improved stretch factors, but also better speedups due to sophisticated early-stopping criteria, compared to the experimentation on FLAT for the same subsets of "global" landmarks.

Our main goal in this work is to demonstrate the practicality of FLAT and HORN, which provide ***provable*** guarantees w.r.t. query times, stretch factors and preprocessing requirements, for large-scale real data sets. The strong aspects of our oracles are the simplicity of the query algorithms, the remarkably small (optimal in most cases) observed stretches, and the achieved speedups. On the negative side, the preprocessing space and time requirements are rather large.

The CFLAT oracle achieves much better preprocessing requirements, with similar

query-times and better approximation guarantees, compared to FLAT. To the best of our knowledge, CFCA(N) is the only query algorithm which accounts in its query-time, apart from the estimation of the minimum travel-time along an od-path, also the actual path construction. And yet, the achieved query times are quite competitive. Moreover, CFLAT is totally scalable, as a typical landmark-based oracle, and it also achieves a noticeable performance against other state-of-art speedup techniques for time-dependent road networks.

For Berlin, the only experimentally evaluated speedup technique we are aware of, TDCRP [29], requires 21min of preprocessing time on a 16-core machine, 31MB of preprocessing space, and achieves 0.28msec query time and average error of 1.47%. For an analogous amount of preprocessing ***work***, CFLAT preprocesses $R500$ in 54min on a 6-core machine, in space 1.3GB, and achieves query-times varying from 0.452msec up to 2.186msec and relative error from 1.883% down to 0.132%, depending on the value of N. If query-time is the main goal, then CFLAT can preprocess $BC4K$ in 7h consuming 10.4GB and achieving query-times varying from 0.128msec up to 0.467msec and average errors from 0.536% down to 0.043%, depending on the value of N. It is also noted that CFCA(6) achieves error at most 0.043% for 95.6%, and at most 0.688% for 98.4% of the $50,000$ queries (cf. Figure 4.9).

For Germany, we compare our oracle with the two most prominent speedup techniques we are aware of, TDCRP and inex.TCH(0.1), using their evaluation as reported in [29]. TDCRP requires total preprocessing time 4h41min on a 16-core machine, using 361MB preprocessing space, and achieves 1.17msec query-time and average error of 0.68%. For an analogous amount of preprocessing ***work***, CFLAT preprocesses $R1K$ in 16h on a 6-core machine, consumes space 26.8GB and achieves query-times varying from 2.87msec up to 17.105msec, and relative error from 1.562% down to 0.072%, depending on the value of N. If query-time is the main goal, then CFLAT preprocesses $BC3K$ in 46h consuming 80.4GB and achieving query-times varying from 0.959msec up to 5.506msec and average errors from 0.911% down to 0.032%, depending on the value of N. It is also noted that CFCA(6) achieves error at most 0.032% for 94.8%, and at most 0.512% for 98% of the $50,000$ queries (cf. Figure 4.10). inex.TCH(0.1), on the other hand, preprocesses the instance in 6h18min, requiring space 1.34GB, and achieves average query-time 0.7msec with average error 0.02% and worst-case error 0.1%.

CFCA(N) gave a speedup of more than $1,030$ w.r.t. query time against the time-

dependent variant of Dijkstra with average stretch 0.536%. That is, the query-time is only slightly worse (0.128msec instead of 0.083msec) even though the measurement in this work includes also the path construction, but the corresponding average error is clearly better than the average error of 0.781% for FCA(N) in FLAT oracle.

Bibliography

[1] C. Sommer, "Shortest-path queries in static networks," ***ACM Comp. Surveys***, vol. 46, 2014.

[2] H. Bast, D. Delling, A. V. Goldberg, M. Mller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck, "Route planning in transportation networks," Tech. Rep. MSR-TR-2014-4, Microsoft Research, 2014.

[3] G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter, "Minimum time-dependent travel times with contraction hierarchies," ***ACM Journal of Experimental Algorithmics***, vol. 18, pp. 1–43, 2013.

[4] D. Delling, "Time-dependent sharc-routing," ***Algorithmica***, vol. 60, pp. 60–94, 2011.

[5] D. Delling, L. Liberti, G. Nannicini, and D. Schultes, "Bidirectional A* search on time-dependent road networks," ***Networks***, vol. 59, pp. 240–251, 2012.

[6] B. C. Dean, "Shortest paths in fifo time-dependent networks: Theory and algorithms," tech. rep., Massachusetts Institute of Technology, 2004.

[7] F. Dehne, O. T. Masoud, and J. R. Sack, "Shortest paths in time-dependent fifo networks," ***Algorithmica***, vol. 62, pp. 416–435, 2012.

[8] L. Foschini, J. Hershberger, and S. Suri, "On the complexity of time-dependent shortest paths," ***Algorithmica***, vol. 68, pp. 1075–1097, 2014.

[9] A. Orda and R. Rom, "Shortest-path and minimum delay algorithms in networks with time-dependent edge-length," ***Journal of the ACM***, vol. 37, pp. 607–625, 1990.

[10] K. Cooke and E. Halsey, "The shortest route through a network with time-dependent intermodal transit times," ***Journal of Mathematical Analysis and Applications***, vol. 14, pp. 493–498, 1966.

[11] K. Ozbay, H. D. Sherali, and S. Subramanian, "The time-dependent shortest pair of disjoint paths problem: Complexity, models, and algorithms," ***Networks***, vol. 31, pp. 259–272, 1998.

[12] S. E. Dreyfus, "An appraisal of some shortest-path algorithms," ***Operations Research***, vol. 17, pp. 395–412, 1969.

[13] B. C. Dean, "Continuous-time dynamic shortest path algorithms," ***Master's thesis***, ***Massachusetts Institute of Technology***, 1999.

[14] B. C. Dean, "Algorithms for minimum-cost paths in time-dependent networks with waiting policies," ***Networks***, vol. 44, pp. 41–46, 2004.

[15] S. Kontogiannis, D. Wagner, and C. Zaroliagis, "Hierarchical time-dependent oracles," ***ISAAC***, 2016.

[16] S. Kontogiannis and C. Zaroliagis, "Distance oracles for time-dependent networks," ***Algorithmica***, vol. 74, no. 4, pp. 1404–1434, 2016.

[17] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies," in ***Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)***, pp. 97–105, 2009.

[18] R. Agarwal and P. Godfrey, "Distance oracles for stretch less than 2," in ***Proceedings of the 24th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'13)***, pp. 526–538, 2013.

[19] E. Porat and L. Roditty, "Preprocess, set, query!," in ***Proceedings of the 19th European Symposium on Algorithms (ESA 2011)***, vol. LNCS 6942, pp. 603–614, 2011.

[20] M. Thorup and U. Zwick, "Approximate distance oracles.," ***Journal of the ACM***, vol. 52, pp. 1–24, 2005.

[21] M. Hilger, E. Kohler, R. H. Mohring, and H. Schilling, "Fast point-to-point shortest path computations with arc-flags," ***The Shortest Path Problem: Ninth DIMACS Implementation Challenge***, vol. 74 of DIMACS Book, pp. 41–72, 2009.

[22] S. Kontogiannis and C. Zaroliagis, "Approximating time-dependent shortest paths in road networks," Tech. Rep. eCOMPASS-TR-17, September 2013.

[23] S. Kontogiannis, D. Wagner, and C. Zaroliagis, "Distance oracles for time-dependent networks," ***ICALP***, no. 1, pp. 713–724, 2014.

[24] Y. Bartal, L. Gottlieb, T. Kopelowitz, M. Lewenstein, and L. Roditty, "Fast, precise and dynamic distance queries," ***ACM-SIAM SODA***, pp. 840–853, 2011.

[25] D. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," ***Springer***, pp. 124–137, 2007.

[26] 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 2015)***, ***SIAM***, pp. 147–158, 2015.

[27] 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 2016)***, ***SIAM***, pp. 1–14, 2016.

[28] G. Mali, P. Michail, A. Paraskevopoulos, and C. Zaroliagis, "A new dynamic graph structure for large-scale transportation networks," ***Algorithms and Complexity – CIAC 2013***, vol. LNCS 7878, pp. 312–323, 2013.

[29] M. Baum, J. Dibbelt, T. Pajor, and D. Wagner, "Dynamic time-dependent route planning in road networks with user preferences," ***15th International Symposium, on Experimental Algorithms (SEA2016)***, vol. LNCS 9685, pp. 33–49, 2016.

Author's Publications

- Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, Christos D. Zaroliagis: Engineering Oracles for Time-Dependent Road Networks. ALENEX 2016: 1-14
- Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, Christos D. Zaroliagis: Analysis and Experimental Evaluation of Time-Dependent Distance Oracles. ALENEX 2015: 147-158

Short Biography

- **Name and Contact Information**

  Name: Georgia Papastavrou

  Address: 13A Neotitos, Heraklion, Athens, 14122

  Date of Birth: 21^{st} June 1986

  Nationality: Greek

  Email: gpapasta21@gmail.com

- **Education**

  Degree on Computer Science, University of Ioannina, 2013.

  Undergraduate student, Department of Primary Education, University of Ioannina

- **Foreign Languages**

  Cambridge Proficiency in English

- **Employment History**

  IT Teacher, Ellinoagliki Primary School, Amarousion, Athens, 2017

  Scientific Associate in Computer Technology Institute and Press, Diophantus, 2013-2016

- **Professional Skills**

  Programming, Software Engineering, Teaching

- **Research Interests**

  Graph Theory, Design and Analysis of Algorithms, Route Planning