ENGINEERING ALGORITHMS FOR ROUTE PLANNING IN MULTIMODAL TRANSPORTATION NETWORKS

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Julian Matthias Dibbelt

aus München

Tag der mündlichen Prüfung:	3. Februar 2016
Erste Gutachterin:	Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Prof. Dr. Christos Zaroliagis

For Sherrie, Marlene, and Frederick.

ACKNOWLEDGMENTS

Foremost, I wish to thank my advisor, Dorothea Wagner, for the great opportunity to join her group, the many advice given throughout the years, and the very productive, friendly and cooperative environment established at her chair. I am deeply grateful to Christos Zaroliagis for co-reviewing this thesis and for his great leadership during our joint projects. I thank PTV for the opportunity to work with them for the first year after my graduate studies, and the European Commission for partially supporting my research at KIT through grants 288094 (project eCOMPASS) and 609026 (project MOVESMART). I thank the members of the respective consortia for the successful projects and the work put into securing the proposals in the first place. I am incredibly grateful to Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck for inviting me to a very productive summer internship at Microsoft Research.

I especially want to thank my co-authors Simeon D. Andreev, Moritz Baum, Valentin Buchhold, Daniel Delling, Andreas Gemsa, Martin Nöllenburg, Thomas Pajor, Lorenz Hübschle-Schneider, Ben Strasser, Dorothea Wagner, Renato F. Werneck, and Tobias Zündorf for all the fruitful collaborations that set the foundation of this thesis. I also thank Takuya Akiba, Dennis Luxen, G. Veit Batz, Spyros Kontogiannis, Felix König, Florian Krietsch, Ignaz Rutter, Dennis Schieferdecker, Frank Schulz, Christian Sommer, Sabine Storandt, and Tim Zeitz for very valuable and insightful discussions we have had. I thank Moritz Baum and Fabian Fuchs for proof reading and general advice regarding the write-up of this thesis.

I am very grateful to all my co-workers of the different research groups at the Institute of Theoretical Informatics; this was a very cooperative and fun place to do research at. I am deeply grateful to Moritz for being a supreme office mate that I could always talk to and depend on. No less am I thankful for the administrative and IT support by Lilian Beckert, Lori Blonn, Bernd Giesinger, Hugo Hernandez, Ralf Kölmel, Simone Meinhart, and Elke Sauer.

Last but not least, I thank my loving wife, my family, and my friends for the strong support in my endeavors. I would not stand here without you.

ABSTRACT

Finding the fastest route through traffic, a well-rated restaurant en-route while traveling, the latest train to catch in order to arrive on-time, or a decently priced apartment, with a well-rated school in safe walking distance from home, in biking distance from work, and good public transit access nearby—all these problems have at their core the computation of shortest path queries on different transportation networks. The wide-scale and frequent adoption of web-based map services for route planning and driving directions, GPS navigation, and other location-based applications on large spatial network databases that we see today, has been enabled by and in turn motivated numerous research into accelerating such shortest path queries.

In particular, the development of practical algorithms for route planning in transportation networks has been a showpiece application of successful *algorithm engineering* during the past decade, where experimental evaluation on real-world networks drives the design, analysis, and implementation of algorithms in a continuous feedback loop. This has produced many *speedup techniques*, varying in preprocessing time, space, and query performance, but also simplicity and ease of implementation. Most approaches follow a similar paradigm: During *preprocessing*, auxiliary data is computed that is subsequently used to accelerate queries by pruning the search space that needs to be considered. Speedup techniques are so mature today that the fastest can answer a single query within only a few memory accesses, while retaining provable correctness of the results.

Due to these successes, focus has shifted to increasingly involved and more realistic scenarios, taking into account, e. g., traffic, user preferences, public transit schedules, and the options offered by the many modalities of modern transportation networks. The challenges here are twofold: 1) Finding the right problem formulation and modeling for the considered scenarios, e. g., what metric or set of metrics to optimize for; 2) Identifying and exploiting the structural properties of the considered problems that enable fast queries. Both themes are explored in this thesis with regards to road networks, public transit, and multimodal journey planning.

ROAD NETWORKS. Most web-based systems or stand-alone navigation devices do not offer much room for user preferences beyond specifying start and end address. But other than traveling quickly, drivers might also like, e. g., to avoid complicated turns, narrow residential streets, highways or toll bridges. Not only might different users have different cost functions to optimize—even the same user might

prefer a fast route in the morning and a safe one at night. Besides user interface and experience considerations, the main reason that fine-grained preferences are often not supported, is the prohibitively high cost of preprocessing of the implemented speedup techniques.

To enable custom user preferences, it has therefore recently been proposed to split the expensive preprocessing: Into a first phase exploiting solely the topology of the graph, and a second, lightweight phase adapting the preprocessed auxiliary data to a specific cost function. On this idea, we adapt Contraction Hierarchies, a well-established preprocessing technique to offer fast customization, and a new, faster query algorithm that avoids the overhead of a priority queue.

While this approach handles dynamically changing costs with ease—like most of the work that focuses on time-independent route planning, it still assumes that cost per road segment are constant per query. In practice, the current traffic situation significantly influences the travel time on large parts of the road network, and it changes over the day. One can distinguish between traffic congestion that can be predicted using historical traffic data, and congestion due to unpredictable events, e. g., accidents. We study a dynamic *and* time-dependent route planning problem, which takes both live traffic and prediction (i. e., short-term and based on historical data) into account. To this end, we propose a practical algorithm that, while robust to user preferences, is able to integrate global changes of the time-dependent metric (e. g., due to traffic updates or user restrictions) faster than previous approaches.

Other scenarios, such as computing routes for pedestrians, have often been neglected or simply dismissed as a trivial matter of applying a different cost function. Instead, we observe that pedestrian routing has its specific set of challenges not met by state-of-the-art route planners. For instance, the lack of detailed sidewalk data and the inability to traverse plazas and parks in a natural way often leads to unappealing and suboptimal routes. Therefore, we propose to augment the network by generating sidewalks based on the street geometry and enabling routing over plazas and squares. Using this and further information, our query algorithm seamlessly handles node-to-node queries and queries whose origin or destination is an arbitrary location on a plaza or inside a park. Our experiments show that we are able to compute appealing pedestrian routes at negligible overhead over standard routing algorithms.

PUBLIC TRANSIT. Public transit networks differ from road networks, as each vehicle follows a scheduled timetable and visits a particular sequence of stops. Typical public transit query problems consider either earliest time of arrival or even multiple criteria such as time and number of transfers, and can be formulated either for a given departure time or for a whole time range (called profile query).

Accordingly, while on road networks there is often a straightforward characterization as a shortest path problem, research into baseline algorithms (i. e., before preprocessing) is still ongoing for public transit. We introduce a novel algorithmic framework to compute journeys directly on the timetable (as opposed to a graph representation thereof). It organizes data as a single array of connections (the most basic building block of a timetable), which it scans once per query. Despite its simplicity, our algorithm is very versatile and solves earliest arrival as well as multicriteria profile queries.

Moreover, developing efficient preprocessing-based speedup techniques for public transit has been considered more challenging than for road networks: Current approaches either require massive preprocessing effort or provide limited speedups. Leveraging recent advances in Hub Labeling (the fastest algorithm for road networks) and domain-specific properties, we provide simple and efficient algorithms for earliest arrival, profile, and multicriteria queries that are orders of magnitude faster than the state of the art.

MULTIMODAL JOURNEY PLANNING. Finally, we study the problem of finding multimodal journeys in transportation networks (including unrestricted walking, driving, cycling, and schedule-based public transportation), which asks for a best *integrated* journey between two locations. Thereby, it is crucial to respect a user's modal preferences: Not every mode of transport might be feasible to him at any point along the journey. In general, the user has restrictions on the sequence of transport modes. For example, some users might be willing to take a taxi between two train rides if it makes the journey quicker. Others prefer to use public transit at a stretch. We provide a multimodal route-planning system that handles such constraints as a *user input* for each *query* (as opposed to already during preprocessing).

Another natural solution to the multimodal problem is to use multicriteria search, in an attempt to capture the multitude of available traveling options that the user might not yet be aware of. However, full multicriteria search tends to be slow, producing too many solutions of often surprisingly little value. Regarding the latter, we propose to score the solutions in a post-processing step using techniques from fuzzy logic, quickly identifying the *most significant* journeys. We also propose several (still multicriteria) heuristics to find similar journeys, but much faster. Our experiments show that this approach enables the computation of high-quality multimodal journeys on large metropolitan areas, and is fast enough for practical applications.

CONTENTS

1 INTRODUCTION 1
- 1.1 Problems Considered 2
- 1.2 Main Contributions 3
  - 1.2.1 Road Networks 3
  - 1.2.2 Public Transit Networks 5
  - 1.2.3 Multimodal Journey Planning 6
- 1.3 Thesis Outline 7

2 RELATED WORK 9
- 2.1 Preprocessing Techniques 9
  - 2.1.1 Theoretical Results 10
  - 2.1.2 Dynamic Scenarios and Customization 11
  - 2.1.3 Time-dependent Scenarios 11
- 2.2 Multicriteria Optimization 13
- 2.3 Sidewalks and Traversal of Open Areas 13
- 2.4 Public Transit Journey Planning 14
- 2.5 Multimodal Journey Planning 15

3 ROUTE PLANNING IN ROAD NETWORKS 17
- 3.1 Customizable Contraction Hierarchies 18
  - 3.1.1 Preliminaries 18
  - 3.1.2 Preprocessing 22
  - 3.1.3 Customization 28
  - 3.1.4 Queries 37
  - 3.1.5 Experiments 40
- 3.2 Time-Dependent Customizable Route Planning 70
  - 3.2.1 Preliminaries 70
  - 3.2.2 Our Approach 72
  - 3.2.3 Experiments 77
- 3.3 Pedestrian Route Planning 87
  - 3.3.1 Preliminaries 88
  - 3.3.2 Augmented Graph Model for Pedestrian Routing 89
  - 3.3.3 Computing Pedestrian Routes 93
  - 3.3.4 Experiments 95

4 PUBLIC TRANSIT JOURNEY PLANNING 103
- 4.1 Connection Scan Algorithm 104
  - 4.1.1 Preliminaries 104
  - 4.1.2 Earliest Arrival Queries 105
  - 4.1.3 Profile and Multicriteria Queries 106
  - 4.1.4 Experiments 108

4.2 Public Transit Labeling 114
4.2.1 Preliminaries 114
4.2.2 Basic Earliest Arrival and Profile Queries 116
4.2.3 Improvements 118
4.2.4 Practical Extensions 120
4.2.5 Experiments 122

5 MULTIMODAL JOURNEY PLANNING 131
5.1 User-Constraints on Multimodal Transfers 131
5.1.1 Preliminaries 132
5.1.2 Contraction Hierarchies for Multimodal Networks 136
5.1.3 UCCH: Contraction for User-Constrained Route Planning 137
5.1.4 Improvements 140
5.1.5 Experiments 142
5.2 Multicriteria Multimodal Journey Planning 153
5.2.1 Preliminaries 153
5.2.2 Exact Algorithms 157
5.2.3 Heuristics 161
5.2.4 Experiments 163

6 FINAL REMARKS 173
6.1 Future Work 173
6.1.1 User Preferences 173
6.1.2 Customizable Contraction Hierarchies 174
6.1.3 Traffic Patterns 174
6.1.4 Public Transit 175
6.1.5 Multimodal 175

BIBLIOGRAPHY 177

A CURRICULUM VITÆ 203

B LIST OF PUBLICATIONS 205

C DEUTSCHE ZUSAMMENFASSUNG 209

1 INTRODUCTION

Services for route planning have become a commodity, used daily by millions of users. The problem of quickly computing optimal routes in transportation networks presents several algorithmic challenges, and has been an active area of research in recent years.

Usually, the considered network is modeled as a weighted, directed graph. While Dijkstra's algorithm [Dij59] can be used to compute an optimal route between two vertices of the graph in almost linear time, this is still too slow for practical applications on large real-world transportation networks. They consist of several million vertices, and the user expect almost instant results.

For practical performance on large networks, numerous *speedup techniques* (or *distance oracles*) have proven useful, which augment the network with auxiliary data in a possibly expensive, offline *preprocessing phase*. This auxiliary data in turn accelerates route computation during subsequent *queries*. As a result, speedups of several orders of magnitude over Dijkstra's algorithm have been achieved. For an overview see the recent surveys by [BDG+15; Som14] and Chapter 2.

Thereby, it is critical to design algorithms that exploit modern processor architectures, which exhibit deep memory hierarchies (multiple levels of increasingly faster caches) and various forms of parallelism (instruction- and core-based). Since typical route planning problems are computationally not very expensive (mostly addition and comparison of costs on candidate paths), memory access becomes the bottleneck, requiring data structures and algorithms with good cache locality for best performance. For queries, it can be argued that parallelism should be spent to serve more users simultaneously. However, to derive fast preprocessing approaches exploitation of parallelism is usually mandatory.

Considering this, traditional algorithm design and analysis (which relies on rather abstract machine models) is best accompanied by careful implementation and experimental evaluation, where observed performance bottlenecks direct the redesign of the algorithmic approach. This framework has become known as *Algorithm Engineering*, see the expositions by [MS10; San09; SW11] for details. At its best, worst-case asymptotic complexity bounds are complemented by measures of practical performance on real-world benchmark instances obtained on real-world machines. It also has the great benefit that the derived algorithms are guaranteed to be practically implementable.

Sometimes, especially if the characteristics of realistic inputs are hard to capture (e. g., what properties define the graph families of

"road networks" and "public transit networks", respectively?), *meaningful* worst-case bounds are hard to come by (in the sense that easily obtained worst-case examples do not happen on typical inputs). In such cases, the experimental aspects of Algorithm Engineering still allow to derive practical solutions. For example, many of the first practical algorithms for route planning in road networks were derived purely experimentally. Only later, theoretical models for, e. g., road networks under travel time metric [ADF+13; AFGW10] were proposed, going full cycle from practical implementation to theoretical analysis and back to practical solutions [DGW11b]. As a result, the fastest techniques on road networks today can answer a single query in less than microseconds [ADGW11], albeit for a basic route planning scenario.

More realistic scenarios encompass consideration of the current traffic as well as predictable traffic patterns (e. g., rush hour), use of public transit, and integrated multimodal journey planning, which also entails appealing pedestrian routes. At the same time, algorithms should be flexible enough so that user preferences and choice can be taken into account. This thesis aims to broaden the state-of-the-art in route planning on these aspects.

1.1 PROBLEMS CONSIDERED

In this thesis, we examine solutions for the following problem scenarios in different transportation networks. Formal definitions are given in later chapters, once we have settled the necessary notation.

ROAD NETWORKS. Given a graph representation $G = (V, A)$ of the segments and intersections of the road network, we consider the

- *Point-to-point shortest path problem:* Given a source vertex s and target vertex t, compute the shortest path between s and t with respect to scalar (single-valued) arc costs $c\colon A \to \mathbb{R}^+$.
- *Time-dependent point-to-point earliest arrival problem:* Given a source vertex s and target vertex t, a departure time τ, compute the earliest arrival at t when leaving s no earlier than τ, taking into account functional arc costs $c\colon A \to (\mathbb{R}^+ \to \mathbb{R}^+)$ that map, per arc, time-of-day to current travel time (to represent rush hours).

PUBLIC TRANSIT. Given the vehicles, stops, and timetable of a public transit network, there are many natural problem variants:

- An *earliest arrival (EA)* problem seeks a journey that arrives at a target stop t as early as possible, given a source stop s and a departure time τ (e. g., "now").
- A *multicriteria (MC)* query also considers the number of transfers when traveling from s to t. It computes the Pareto set of

non-dominated journeys under the criteria earliest arrival and number of transfer, respecting a departure time τ.

- A *profile* query reports all quickest journeys between two stops within a time range. It computes the Pareto set of non-dominated journeys under the criteria latest departure time and earliest arrival time.
- A *multicriteria profile* query reports all quickest journeys between two stops within a time range. It computes the Pareto set of non-dominated journeys under the criteria latest departure time, earliest arrival time, and number of transfers.

MULTIMODAL NETWORKS. Finally, we consider integrated multimodal transportation networks of walking, biking, private car, taxi, and schedule-based public transit and flight. We assume that the multimodal network is augmented sufficiently with transfers between the modes of transportation. We examine two problem variants:

- *Label constrained shortest path problem (LCSPP)* [BJMoo]. Given a combined graph representation of the different input networks, with each arc labeled by its mode of transport: Compute a shortest path P, where the word $w(P)$ formed by concatenating the arc labels along P is an element of a language L, a query input. In particular, we consider a point-to-point earliest arrival variant (cf. above) of this problem, given an arbitrary source and target location, and a departure time τ.
- *Location-to-location multimodal multicriteria shortest path problem:* Given an arbitrary source and target location in a multimodal network, and a departure time τ, compute a concise but diverse set of representative multimodal journeys between source and target location that optimize travel time, convenience and costs, leaving the source not earlier than τ.

1.2 MAIN CONTRIBUTIONS

This thesis broadens the state-of-the-art in route planning in 1) road networks, in 2) public transit networks, and for 3) multimodal journey planning. The following discusses the chief contributions per topic.

1.2.1 *Road Networks*

Here, we consider fast metric customization, traffic, and pedestrian route planning.

CUSTOMIZABLE CONTRACTION HIERARCHIES. Exploiting small separators in road networks, we show that *Customizable Contraction*

Hierarchies (CCH) are feasible and practical. To this end, we employ metric-independent *nested dissection (ND) orders* [Geo73] for precomputation. This approach was proposed by [BCRW13], and a preliminary case study can be found in [Zei13].

Compared to the main competitor, CRP [DGPW15], we achieve a similar preprocessing–query trade-off, albeit with slightly better query performance at slightly slower customization speed. Interestingly, for metrics less well-behaved such as travel distance, we achieve query times well below the original metric-dependent Contraction Hierarchy (CH) described by [GSSV12].

As for CRP, our approach offers fast customization of any scalar input metric with strict worst-case performance guarantees on both customization and queries. We also introduce *perfect witness search*, which computes a CCH with provably minimum number of arcs, given a fixed metric-independent vertex order. Costing only a constant factor in customization time, this yields even better query performance.

TIME-DEPENDENT CUSTOMIZABLE ROUTE PLANNING. It is often proposed to use dynamic but scalar route planning (such as provided by CRP or CCH) to handle live traffic information—by dynamically updating arc weights of congested streets. However, this necessarily yields inaccurate results for medium and long-distance paths: Such methods will (wrongly) consider current traffic even at far away destinations—traffic that might well have dispersed once reaching the destination. For most-realistic results, a combination of dynamic and time-dependent (non-scalar, functional) route planning is necessary that accounts for current traffic, short-term predictions, and long-term historic knowledge about recurring traffic patterns.

In this thesis, we carefully extend a customizable route planning approach to handle time-dependent functions. As such, we are the first to evaluate partition-based route planning on a challenging non-scalar metric. To this end, we integrate profile search into the customization phase and compute time-dependent overlays. We observe that, unlike EVCRP [BDPW13] and TCH [BGSV13], a naïve implementation fails: Shortcuts on higher-level overlays become too expensive to be kept in memory (and too expensive to evaluate during queries). In order to reduce functional complexity, we propose to approximate overlay arcs after each level already (in contrast to ATCH [BGSV13], which uses a post-processing step). In fact, even slight approximation suffices to make our approach practical, in accordance to theory [FHS14]. The resulting algorithmic framework enables interactive queries with low average and maximum error in a very realistic scenario consisting of live traffic, short-term traffic predictions, and historic traffic patterns, while also supporting and being robust to user preferences such as lower maximum driving speeds or the avoidance of left-turns.

PEDESTRIAN ROUTE PLANNING. The customizable route planning approaches discussed above, such as CRP and CCH, easily allow to compute routes for pedestrians by applying a different cost function. We argue that this naïve approach may lead to unnatural and suboptimal solutions. In fact, pedestrians utilize the street network quite differently from cars, which is often not captured by traditional approaches. For example to save distance, pedestrians are free to deviate from the streets, using the walkable area of public open spaces such as plazas and parks. On the other hand, crossing large avenues can be expensive (due to traffic), and it may be faster and safer to walk a small detour in order to use a nearby bridge or underpass.

In this part, we address the unique challenges that come with computing pedestrian routes. In order to obtain as realistic routes as possible, we propose to first augment the underlying street network model, and then to apply a tailored routing algorithm on top of it. We discuss geometric approaches for automatically adding sidewalks, calculating realistic crossing penalties for major roads, and preprocessing plazas and parks in order to traverse them in a natural way. Our integrated routing algorithm seamlessly handles queries whose origin or destination is an arbitrary geographic location inside a plaza or park, or at a street. By experimental analysis, we demonstrate the practicability of our approach. In case studies, we observe that the pedestrian routes computed by our approach are much more appealing than those by state-of-the-art route planners.

1.2.2 *Public Transit Networks*

Unlike roads, railroad tracks cannot be accessed freely. Instead, journeys in public transit networks have to adhere to a fixed set of stops for boarding, served by a fixed set of vehicles that follow a scheduled timetable. Therefore, approaches for road networks do not directly carry over.

CONNECTION SCAN ALGORITHM. We present a new basic approach to journey planning in public transit networks. Like RAPTOR [DPW14], it is not graph-based. Unlike RAPTOR, it is not centered around routes but elementary *connections*, which are the most basic building block of a timetable. CSA organizes them as one single array, which it then scans once (linearly) to compute earliest arrival journeys to all stops of the network. The algorithm turns out to be intriguingly simple with excellent spatial data locality. We also extend CSA to efficiently handle multicriteria profile queries: For a full time period, it computes the Pareto set of journeys (optimizing departure and arrival time and number of transfers), again with a single scan of the connection array. Our experiments on, amongst others, the dense metropolitan network of London validate our approach. With CSA,

we compute earliest arrival queries and multicriteria profile queries faster than previous algorithms.

PUBLIC TRANSIT LABELING. Despite all the progress in recent years [BCE+10; BS14; SW14], queries on road networks are still several orders of magnitude faster than on public transit [BDG+15]. To reduce this gap, we adapt 2-hop labeling [ADGW12; CHKZ03] to public transit networks, improving query performance by orders of magnitude over previous methods, while keeping preprocessing time practical. Starting from the time-expanded graph model [PSWZ08], we extend the labeling scheme by carefully exploiting properties of public transit networks. Besides earliest arrival and profile queries, we address multicriteria and location-to-location queries. We validate our Public Transit Labeling (PTL) algorithm by careful experimental evaluation on large metropolitan and national transit networks, achieving queries within microseconds.

1.2.3 *Multimodal Journey Planning*

Here, we investigate integrated algorithms that compute journeys in a combined network of walking, biking, private car, taxi, public transit, or flight. We consider two aspects: 1) the avoidance of unwanted transfers between modes of transportation (e. g., usage of a private car between train rides), and 2) multicriteria search to provide the user with meaningful multimodal alternatives.

USER-CONSTRAINTS ON MULTIMODAL TRANSFERS. We present *User-Constrained Contraction Hierarchies* (UCCH), the first multimodal speedup technique that handles arbitrary mode-sequence constraints as user input to the query—a feature unavailable from previous techniques. Unlike its predecessor, Access-Node Routing [DPW09a], it also answers local queries correctly and requires significantly less preprocessing effort. Our experimental study shows that, unlike previous techniques, we can handle an intercontinental instance composed of cars, railways and flights, achieving query times that are fast enough for interactive scenarios.

MULTICRITERIA MULTIMODAL JOURNEY PLANNING. To capture the many options of metropolitan multimodal transportation networks, we employ multicriteria optimization. We argue that most users consider three criteria: travel time, convenience, and costs. As this produces a large Pareto set, we propose using fuzzy logic [FA04; Zad88] to identify, in a principled way, a modest-sized subset of representative journeys. This postprocessing step is very quick and can incorporate personal preferences. We can use recent algorithmic developments [DPW14; DPW15; GSSV12] to answer exact queries

optimizing time and convenience in less than two seconds within a large metropolitan area, for the simpler scenario of walking, cycling, and public transit. Unfortunately, this is not enough for interactive applications, and becomes much slower when more criteria, such as costs, are incorporated. We therefore also propose heuristics (still multicriteria) that are significantly faster, and closely match the top journeys in the Pareto set. A thorough experimental evaluation of all algorithms, in terms of both solution quality and performance, shows that our approach enables interactive applications. Moreover, since it does not rely on heavy preprocessing, it can easily be used in fully dynamic scenarios.

1.3 THESIS OUTLINE

This thesis is organized as follows:

Chapter 2 gives an overview of the related work.

Chapter 3 describes the first key contribution of this thesis. It considers route planning on road networks. In particular, Section 3.1 discusses customizable route planning, Section 3.2 extends customization to live and historic traffic data, and Section 3.3 expands on specifics of pedestrian route planning.

Chapter 4 contains the second main contribution of this thesis on the topic of journey planning in public transit networks. In particular, Section 4.1 introduces a new baseline algorithm, where as Section 4.2 discusses a preprocessing approach that enables much faster queries than before.

Chapter 5 describes the third key contribution of this thesis. It considers journey planning in multimodal networks. In particular, Section 5.1 examines fast computation of label-constrained shortest paths (with constraints as a query input). Section 5.2 discusses algorithms to obtain significant travel alternatives in metropolitan multimodal networks.

Chapter 6 concludes this thesis with an outlook on future work.

References

Contents of this thesis have appeared previously in the following publications. In particular, Section 3.1 is based on joint work with Ben Strasser and Dorothea Wagner [DSW14; DSW16]. Section 3.2 is based on joint work with Moritz Baum, Thomas Pajor, and Dorothea Wagner [BDPW15]. Section 3.3 is based on joint work with Simeon D. Andreev, Martin Nöllenburg, Thomas Pajor, and Dorothea Wagner [ADN+15]. Section 4.1 is based on joint work with Thomas Pajor, Ben Strasser, and Dorothea Wagner [DPSW13]. Section 4.2 is based on joint work with Daniel Delling, Thomas Pajor, and Renato F. Wer-

neck [DDPW15]. Section 5.1 is based on joint work with Thomas Pajor and Dorothea Wagner [DPW12b; DPW15]. Section 5.2 is based on joint work with Daniel Delling, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck [DDP+12; DDP+13]. Some figures are also taken from the respective conference talks of these publications.

2 RELATED WORK

Given a graph representation of the transport network, the classical solution to shortest path problems is Dijkstra's algorithm [Dij59]. In case of point-to-point queries, slightly better performance is achieved by employing bidirectional search from both source and target at the same time [Dan62; Dre69; LR89; Nic66; Poh71]. One may employ A* [HNR68; Poh71], using easily available bounds (e. g., Euclidean distance) to guide the search to the target. In road networks, however, such simple bounds do not pay off [GH05].

2.1 PREPROCESSING TECHNIQUES

Many *speedup techniques* have been proposed for further acceleration. Most of these divide the work into two phases: In a *preprocessing phase* the graph is augmented with auxiliary data that is then exploited during the *query phase* for faster shortest path or distance retrieval. A detailed overview of techniques is given in [BDG+15; Som14].

Examples of such techniques are *Geometric Containers* [WWZ05], *Arc-Flags* [DGNW13; HKMS09; Lau04], *Landmark*-based A* (ALT) [EP13; GH05], *Reach* [GKW07; GKW09; Gut04; MCB14], *Contraction Hierarchies* [GSSD08; GSSV12; WXD+12], *Transit Node Routing* [ALS13; BFSS07], *Hub-based Labeling* [ADF+12; ADGW11; ADGW12; AIY13; CHKZ03; DGPW14; DGSW14; GPPR04], and several *separator*-based techniques [DGPW15; DHM+09; HSW08; JP02; SWW00; SWZ02]. Various combinations of these techniques have also been studied [BD09; BDS+10; EPV15; HSWW06].

A common concept among these techniques are distance-preserving *shortcuts:* additional arcs added to the graph that allow query algorithms to bypass large regions of the graph efficiently, while preserving correctness of the results. For instance, Contraction Hierarchies (CH), at its core, consists of a systematic way of iteratively contracting vertices along a given vertex order, each time adding shortcuts between yet uncontracted neighbors.

Nearly all techniques are *metric-dependent* in the sense that the preprocessing exploits shortest path structure; If the graph metric (i. e., the path optimization objective) changes, preprocessing has to be updated or repeated. A notable exception is *Customizable Route Planning (CRP)* [DGPW11; DGPW15], discussed further below.

2.1.1 *Theoretical Results*

Although many of the above preprocessing approaches work well in practice—as has been established by extensive experimental evaluation on real-world benchmark instances—optimal preprocessing is often provably NP-hard in general: optimal landmark selection for ALT, partitioning for Arc-Flags, separator selection, variants of the optimal shortcut selection problem, or computation of an optimal (in terms of search space size) CH vertex ordering have all been shown to be hard [BBRW13; BCK+10; BDD+12; Mil12].

Hence, several studies [ADF+13; BCRW13; EG08] have discussed properties of road networks to explain the observed efficiency in practice. However, there is still no precise characterization of the family of road graphs.

Starting on the observation that few vertices cover most long shortest paths [BFSS07] (at least for travel time metric), the concept of *Highway Dimension (HD))* [ADF+11; ADF+13; AFGW10] has been developed as a theoretical model of road networks. It allows to show polynomial preprocessing and small query time bounds for several speedup techniques. However, to the best of our knowledge, road networks have only been conjectured to have low HD, this has never (experimentally) been demonstrated. Furthermore, a synthetic network generation study [BKMW10] has found that the HD model yields graphs of too high density and vertex degree. Accordingly, generated graphs do not look like road networks. See [Mei11] for more details.

Instead, it is experimentally well-established that road networks have small, balanced separators [DGRW11; EG08; HS15; SS12b; SS15]. In [BCRW13], CH has been shown to be related to the minimum fill-in problem (as in Gaussian elimination). In that terminology, the graph constructed by adding shortcuts to the CH is a *chordal supergraph*. The *treewidth* of a graph G corresponds to the minimum over the maximum clique size (minus one) over all chordal supergraphs of G. Nested dissection [Geo73; LRT79] orders, used as CH vertex orderings, yield sub-linear (in the number of vertices) bounds on the CH search space [BCRW13]. A preliminary case study that bridges these theoretical results back to practice can be found in [Zei13].

Similar ideas have also appeared in [PWK12], where graphs of low treewidth are considered, leveraging this property to compute CH-like structures (without explicitly using the term CH). Related techniques by [CZ00; Wei10] work directly on the tree decomposition. Interestingly, low highway dimension graphs do not exclude fixed-size minors and therefore do not necessarily have low treewidth [FFKP15].

2.1.2 *Dynamic Scenarios and Customization*

An important aspect of realistic route planning, is handling unforeseen dynamic changes, such as congestion after an accident. While this can trivially be solved by repeating the whole preprocessing phase, less costly approaches [DDFV12; DW07; EP13; GSSV12; SS07] enable partial updates of preprocessed data.

Building on previous separator-based, multi-level overlays [HSW08; JP02; SWW00; SWZ02], Customizable Route Planning (CRP) goes even further [DGPW11; DGPW15], introducing a three-phase speedup technique: Most preprocessing effort is offloaded to an initial, *metric-independent* phase that solely exploits the topological structure of the network. In particular, based on a nested multilevel partition of the graph, unweighted shortcuts are computed between the boundary vertices of each cell (a series of nested overlay graphs). In a second, much less expensive phase, these overlays are *customized* to a specific metric, computing distance-preserving weights for the predetermined overlay shortcuts. Traversing these shortcuts then enables the query to skip large parts of the graph.

Unless new streets or bridges are built, CRP preprocessing does not have to be repeated. Not only allows this approach fast dynamic changes to the metric, it also enables robust integration of user preferences and extended costs models such as turn-costs: Small graph separators in road networks guarantee efficient customization and query times for any scalar metric, since the induced overlay graphs are small [DGPW15]. Moreover, the graph partition also guides parallelization on multiple CPU cores or even GPUs [DKW14], yielding customization times that are faster than a single Dijkstra query.

2.1.3 *Time-dependent Scenarios*

Another important aspect of route planning in realistic scenarios is the consideration of traffic patterns such as rush hours, which greatly influence travel time [DBS10]. For *time-dependent* route planning, routing costs are no longer constant, scalar values per road segment. Rather, costs depend on the time of day at which a road is traversed, i. e., they vary as a function of time [CH66; Dre69; DW09b]. These functions are typically precomputed from historic knowledge of traffic patterns [PBB+08]. However, the complexity of the functional description is known to strongly increase for long-distance routes [FHS14], which poses a difficulty when adapting preprocessing techniques based on (long-distance) *shortcuts* [DW09b].

Dijkstra's algorithm [Dij59] is easily extended to time-dependent shortest paths [CH66; Dre69], as long as the time-dependent cost functions have certain, reasonable properties [OR90]. Two query variants are typically considered: (1) Given a source and target vertex and a

fixed departure time at the source, compute the *earliest arrival time (EA)* at the target; (2) compute earliest arrival times for all departure times of a day (*profile search*).

Time-dependent cost functions are typically expressed as piecewise-linear functions, mapping departure to travel time. The complexity of such a function is expressed by its number of breakpoints. Let n be the number of vertices in the graph: In [FHS14] it is shown that the optimal path between two vertices can change $n^{\Theta(\log n)}$ times, when varying departure time. In other words, for a given vertex pair, any representation of the functional dependency between departure time and travel time needs super-polynomial size. While these are worst-case bounds for arbitrary graph families, a very strong increase in functional complexity has also been observed by experimental evaluation of profile searches on real-world road networks [DW09b].

Several known speedup techniques have been adapted to handle time-dependency. Some works use (scalar) lower bounds on the travel time functions to guide the graph search [DN12; DW07; NDLS12], circumventing any problems with functional complexity increase of shortcuts. TD-CALT [DN12] yields reasonable EA query times (at least for approximate solutions) and allows for fast dynamic traffic updates, but does not enable profile search on large networks. TD-SHARC [Del11] offers profile search on a country-scale network. Time-dependent Contraction Hierarchies (TCH) [BGSV13] enable both fast EA and profile searches even on continental networks. Along the lines of normal CH [GSSV12], the technique greedily computes a vertex order, building a sequence of overlays by iteratively contracting "unimportant" vertices while inserting shortcuts between the remaining neighbors; However, the weights of shortcuts are obtained using profile searches. After preprocessing, piecewise-linear function approximation [II87] can be used on the arc functions to save space, dropping optimality of subsequent query results. Quite cleverly, a multi-phase extension of the query algorithms (ATCH) restores exact results, despite approximated shortcuts [BGSV13]. Unfortunately, TCH and ATCH have not been shown to be robust against user preferences. In [KWZ15; KZ14; KZ15], time-dependent shortest path oracles are considered that provide approximate time-dependent distances in sublinear query time, after subquadratic preprocessing effort. The approach was also evaluated in experimental settings [KMP+15; KMP+16], but preprocessing effort and space consumption are comparably high, despite the good theoretic bounds.

Unlike the scenarios on which CRP was analyzed (cf. above), for time-dependent metrics, small separators do not guarantee fast customization and query times: Even having a low number of overlay arcs can be too expensive if the corresponding functional arc weights increase rapidly in complexity. To the best of our knowledge, non-scalar metrics for explicitly separator-based overlay approaches have

so far only been investigated in the context of energy-optimal routes for electric vehicles (EVCRP) [BDPW13], where energy consumption depends on the battery state-of-charge, but functional complexity for long shortcuts grows very slowly.

2.2 MULTICRITERIA OPTIMIZATION

Many of the methods discussed so far optimize a single criterion (typically travel time). But some also extend to multiple criteria [Ehr05] by utilizing multi-dimensional vertex labels that represent sets of *Pareto-optimal* paths [Han79; Mar84]. Interestingly, even without preprocessing, Pareto optimization is theoretically hard [GJ79]. Depending on the criterion space and transportation network, however, this problem may actually be "feasible in practice" [MW01].

For general networks, the recent NAMOA* algorithm [MP10] is an extension of A* search [HNR68] to the multicriteria case, where vertex potentials help reducing the number of label scans. This approach was also applied to road networks [MM12] and later parallelized [EKS14; SM13]. Pareto-SHARC [DW09a] applies preprocessing to the Pareto optimization problem, however, it drops exactness in order to achieve practical performance.

For the case that the metric is a linear combination of two or more criteria, practical algorithms are available as well [DSW15; FNS14; FS13; GKS10]. However, even for the three criteria of travel time, travel distance, and fuel consumption (which are even quite correlated), diminishing returns in terms of query speed over preprocessing effort have been reported [FS13], due to a strong increase in multi-arc shortcuts. Contraction Hierarchies with edge restrictions [GRST12; RT10] use very similar techniques.

2.3 SIDEWALKS AND TRAVERSAL OF OPEN AREAS

For an overview and assessment of different sidewalk generation approaches, see [KK13]. Many works consider extraction of street networks from satellite images [MZ07; PJPZ10]. While this approach is promising for roads, extracting sidewalks is problematic due to poor image resolution and occlusion (e. g., by trees). Moreover, satellite imagery is not as easily available as street data. In contrast, a street network analysis technique proposed by [PV03] generates sidewalk information directly from street layouts, but it does neither handle multiple lanes very well, nor streets that are close to each other. An alternative technique [BPS11] leverages building layouts to generate sidewalks, however, not all streets that have sidewalks are also adjacent to a building, resulting in incomplete output.

Traversing open areas is a classical problem in robotics and computational geometry, and numerous works exist on the subject [SS88].

Cell decomposition [Lat91] yields paths that are offset from the obstacles and area boundary, and [MA04] combines several techniques, including Voronoi diagrams, to obtain robust collision-free robot motion paths. Visibility graphs [AW88] are specifically important to us, since they represent geometric shortest paths.

2.4 PUBLIC TRANSIT JOURNEY PLANNING

For purposes of journey planning, public transit networks are based on scheduled vehicles and fixed stops for entering and leaving a vehicle. (The exact course of railroad tracks, switches, signaling, or allocation of tracks to vehicles, etc., are interesting optimization problems, but of no concern to us.)

However, even more so than for road networks, deciding on an optimization objective for journeys is not straightforward. For example, while some people want to arrive as early as possible, others are willing to spend a little more time to avoid extra transfers. Hence, there are several natural problem variants [MSWZ07]: The simplest, called *earliest arrival*, takes a departure time as input, and determines a journey that arrives at the destination as early as possible. If further criteria, such as the number of transfers, are important, one may consider *multicriteria* optimization [DMS08; MS07; MW01]. Finally, a *profile query* [DKP12; DPW12a] computes a set of optimal journeys that depart during a period of time (such as a day).

Similar to road networks, these problems can be approached by variants of Dijkstra's algorithm [Dij59] applied to a graph that models the public transit network, with various techniques to handle time-dependency [CDD+14; DKP12; MMPZ13; MSWZ07; PSWZ08; Wit15]. In particular, the *time-expanded* (TE) graph encodes time in the vertices, creating a vertex for every *event* (e. g., a train departure or arrival at a stop at a specific time). In contrast, the *time-dependent* (TD) graph has stops as vertices, with scheduled departures represented as time-dependent arc cost functions. Other approaches, however, no longer require a graph or a priority queue. RAPTOR [DPW12a; DPW14] solves the multicriteria problem (arrival time and number of transfers) directly on the timetable by dynamic programming.

While preprocessing-based techniques for computing point-to-point shortest paths have been very successful on road networks, their adaption [BDS+10; BGM10; Del11; DGWZ08; DPW09b; SWW00; SWZ02] to public transit networks is "harder than expected" [Bas09; BDGM09; BDW11], i. e., they do not yield the same level of acceleration. For example, initial experiments for Timetable Contraction Hierarchies [Gei10] look promising, but a newer study shows them not to perform well on dense networks of both metropolitan and country scale, such as Switzerland or London [Wir15]. Prominent candidates for fast public transit journey planning are Transfer Patterns [BCE+10; BS14] and

ACSA [SW14], but they still do not yield the same query performance as exhibited on road networks.

For aperiodic timetables, the TE graph model yields a *directed acyclic graph* (DAG), and several public transit query problems translate to reachability problems. Different methodologies exist to enable fast reachability computation [CHWF13; JW13; MS14; SABW13; YAIY13; YCZ10; ZLWX14]. In particular, the *2-hop labeling* [CHKZ03] scheme associates with each vertex two labels (forward and backward); reachability (or shortest-path distance) can be determined by intersecting the source's forward label and the target's backward label. Note that on road networks, 2-hop labeling yields the fastest known distance queries, taking less than a microsecond even on continental networks [ADGW12].

2.5 MULTIMODAL JOURNEY PLANNING

Public transit networks necessarily also have a multimodal component, since journeys inherently require some amount of walking. To handle this, existing solutions (see above) predefine transfer arcs between nearby transit stops, then run a search algorithm on the public transit network to find the "best" journey. Thereby, query performance noticeably depends on the number of such transfer arcs, hence, adding the full clique of all stops (in the timetable) is not a viable option. Instead, transfer arcs are typically shorter than five to ten minutes.

Therefore, this approach misses interesting traveling options, e. g., for pedestrians willing to walk more. In practice, users would want an integrated solution, considering all available modes of transportation, including unrestricted walking, biking, and taxis. We refer to this as the *(fully) multimodal journey planning* problem.

Extending public transportation solutions to a fully multimodal scenario may seem trivial at first: One could just incorporate routing techniques for road networks to solve the new subproblems. However, fastest multimodal paths can have arbitrarily many and sometimes infeasible transfers between modes of transportation.

An elegant approach to restricting modal transfers is the label constrained shortest paths problem (LCSPP) [MW95]: Arcs are labeled according to the mode of transportation, and the sequence of arc labels for any feasible path must induce an element of a formal language. Ideally passed as query input, this typically enforces a hierarchy of modes [BBM06; YL12]: For example, "no use of private car between trains rides", "no train between flights", or "no walking between bike rentals".[1] A version of Dijkstra's algorithm can be used, if the language is regular [BJM00; MW95]. An experimental study of this approach, including basic goal-directed techniques, is conducted in [BBH+09].

1 The latter happing in a scenario near a large pedestrian zone, where dropping off the bike and walking to the next bike rental station might be slightly faster, overall.

In [Paj09] it is concluded that augmenting preprocessing techniques for LCSPP is a challenging task.

A first efficient multimodal speedup technique, called Access-Node Routing (ANR), has been proposed in [DPW09a]. It skips the road network during queries by precomputing distances from every road vertex to all its relevant access points of the public transportation network. It has the fastest query times of all previous multimodal techniques which are in the order of milliseconds. However, the preprocessing phase predetermines the modal constraints that can be used for queries. Also, it cannot compute short-range queries and requires a separate algorithm to handle them correctly.

Another approach, called SDALT [KLPC11], adapts ALT by precomputing different vertex potentials depending on the mode of transport. It allows fast preprocessing, but both preprocessing space and query times are high. Also, it cannot handle arbitrary modal restrictions as query input. By combining SDALT with a label-correcting algorithm, the query time can be improved by up to 50 % [KLC12].

Finally, in [RT10] a technique based on contraction is presented that handles arbitrary Kleene languages as user input. The authors use them to exclude certain road categories. They report speedups of three orders of magnitude on a continental road network. However, Kleene languages are rather restrictive: In a multimodal context, they would only allow excluding modes of transportation *globally*. In particular, they cannot define feasible *sequences* of transportation modes.

3

ROUTE PLANNING IN ROAD NETWORKS

There has been a plethora of research in algorithm engineering on methods for faster route planning in road networks, as shown in Chapter 2. Ultimately, this led to approaches [ADGW12; ALS13] that yield point-to-point query times in the order of microseconds and below, which is more than fast enough for most applications. However, these results are only achieved for the most basic optimization objective.

In contrast, in this chapter, we focus on slower but more broadly applicable approaches that aim to reduce preprocessing effort while taking the dynamics of road traffic, more precise routing models, and changing user preferences into account. At the same time, our goal is to achieve query times still fast enough to enable responsive, interactive applications for directions and navigation.

The general framework, which we follow and extend in this chapter, is the 3-phase approach of [DGPW11; DGPW15] proposed for Customizable Route Planning. Like previous works [DHM+09; HSW08; JP02; SWW00], it exploits that road networks have small, balanced separators [DGRW11; EG08; HS15; SS12b; SS15]. However, unlike previous works, it proposes to utilize this for very fast metric-dependent preprocessing (called *customization*) by offloading most of the work to a first, metric-independent preprocessing phase that only exploits the topology of the network.

CHAPTER OUTLINE. Section 3.1 introduces Customizable Contraction Hierarchies (CCH), showing how customization can be applied to the well-established Contraction Hierarchies (CH) speedup technique [GSSD08; GSSV12]. Since CH has found widespread use in practice, our results immediately enable straightforward consideration of user preferences and live traffic for many real-world applications. In turn, Section 3.2 takes a more closely look at the different aspects of road traffic that can be used as an input to route planning: the current (live) traffic situation, short-term traffic predictions, as well as long-term traffic patterns. We provide a practical approach that fulfills the requirements of both time-dependent and dynamic route planning applications, and additionally allows user-specific metric customization. Finally, in Section 3.3 we shift our attention away from driving directions for (essentially) cars, instead providing methods towards more realistic pedestrian route planning.

3.1 CUSTOMIZABLE CONTRACTION HIERARCHIES

Contraction Hierarchies (CH) [GSSV12] are a very versatile speedup technique for road networks. Originally designed to quickly answer point-to-point queries, they have been extended to many more scenarios, such as one-to-all, one-to-many, point of interests [DGNW13; DGW11a; FWL12; Gei11; GS10], ride sharing [GLS+10], and time-dependent route planning [BGSV13; BS12; GS10]. Contraction Hierarchies have also proven useful as a "technological pathway" towards the engineering [ADGW11; ADGW12] of hub labeling (HL) [CHKZ03; GPPR04] on road networks, although there are now efficient algorithms to directly derive a labeling from a vertex ordering [AIY13; DGPW14]. To accelerate CH preprocessing, different parallelization approaches have been evaluated [BGSV13; KLSV10; LS12].

With acceleration in mind, we extend Contraction Hierarchies to support a three-phase workflow (such as that of CRP [DGPW15]): The expensive preprocessing is split into 1) a phase exploiting solely the unweighted topology of the graph, and 2) a very lightweight phase that adapts auxiliary data obtained by the first phase to a given routing cost function. We provide an in-depth experimental analysis on large road networks that shows that Customizable Contraction Hierarchies (CCH) are a very practicable solution in scenarios where arc weights often change.

Customizable Contraction Hierarchies are based on nested dissection (ND) orders [Geo73] instead of the metric-dependent order computation of [GSSV12]. This approach was first proposed by [BCRW13], and a preliminary case study can be found in [Zei13]. Similar ideas have also appeared in [PWK12]. Related techniques [CZ00; Wei10] work directly on the tree decomposition.

The rest of this section is organized as follows. Section 3.1.1 sets necessary notation. Section 3.1.2 describes the metric-independent preprocessing phase, consisting of order computation, hierarchy construction, and triangle enumeration. In Section 3.1.3, we discuss how to customize the preprocessing to any scalar input metric. Section 3.1.4 describes different query algorithms. We then present in Section 3.1.5 an extensive experimental study that thoroughly evaluates the proposed algorithms.

3.1.1 *Preliminaries*

Throughout Section 3.1, we use the following notation and concepts:

We denote by $G = (V, E)$ an *undirected* n-vertex *graph* where V is the set of *vertices* and E the set of *edges*. Furthermore, $G = (V, A)$ denotes a *directed graph*, where A is the set of *arcs*. A graph is *simple* if it has no loops or multi-edges. Graphs are simple unless noted otherwise, e. g., in parts of Section 3.1.2.2. Furthermore, we assume

that input graphs are strongly connected. We denote by $N(v)$ the neighborhood of vertex $v \in G$, i. e., the set of vertices adjacent to v; for directed graphs the neighborhood ignores arc direction. A *vertex separator* is a vertex subset $S \subseteq V$ whose removal separates G into two disconnected subgraphs induced by the vertex sets A and B. The sets S, A and B are disjoint and their union forms V. Note that the subgraphs induced by A and B are not necessarily connected and may be empty. A separator S is *balanced* if $\max\{|A|, |B|\} \leqslant 2n/3$.

A *vertex order* $\pi : \{1 \dots n\} \to V$ is a bijection. Its inverse π^{-1} assigns each vertex a *rank*. Every undirected graph can be transformed into a *upward directed graph* with respect to a vertex order π, i. e., every edge $\{\pi(i), \pi(j)\}$ with $i < j$ is replaced by an arc $(\pi(i), \pi(j))$. Note that all upward directed graphs are acyclic. We denote by $N_u(v)$ the *upward neighborhood* of v, i. e., the neighbors of v with a higher rank than v, and by $N_d(v)$ the *downward neighborhood* of v, i. e., the vertices with a lower rank than v. We denote by $d_u(v) = |N_u(v)|$ the *upward degree* and by $d_d(v) = |N_d(v)|$ the *downward degree* of a vertex.

Undirected edge weights are denoted using $w : E \to \mathbb{R}_{>0}$. With respect to a vertex order π we define an *upward weight* $w_u : E \to \mathbb{R}_{>0}$ and a *downward weight* $w_d : E \to \mathbb{R}_{>0}$. For directed graphs, one-way streets are modeled by setting w_u or w_d to ∞.

A path P is a sequence of adjacent vertices and incident edges. Its *hop-length* is the number of edges in P. Its *weight-length* with respect to w is the sum over all edges' weights. Unless noted otherwise, length always refers to weight-length. A shortest st-path is a path of minimum length between vertices s and t. The minimum length in G between two vertices is denoted by $\mathrm{dist}_G(s, t)$. We set $\mathrm{dist}_G(s, t) = \infty$ if no path exists. An *up-down path* P with respect to π is a path that can be split into an upward path P_u and a downward path P_d. The vertices in the upward path P_u must occur by increasing rank π^{-1} and the vertices in the downward path P_d must occur by decreasing rank π^{-1}. The upward and downward paths meet at the vertex with the maximum rank on the path. We call this vertex the *meeting vertex*.

The vertices of every directed acyclic graph (DAG) can be partitioned into *levels* $\ell : V \to \mathbb{N}$ such that for every arc (x, y) it holds that $\ell(x) < \ell(y)$. We only consider levels such that each vertex has the lowest possible level. Note that such levels can be computed in linear time given a directed acyclic graph.

The unweighted *vertex contraction* of v in G consists of removing v and all incident edges and inserting edges between all neighbors $N(v)$ if not already present. The inserted edges are called *shortcuts* and the other edges are *original edges*. Given an order π the *core graph* $G_{\pi,i}$ is obtained by contracting all vertices $\pi(1) \dots \pi(i-1)$ in order of their rank. We call the original graph G augmented by the set of shortcuts a *contraction hierarchy* $G^*_\pi = \bigcup_i G_{\pi,i}$. Furthermore, we denote by $G^{\triangle}_\pi$ the corresponding upward directed graph.

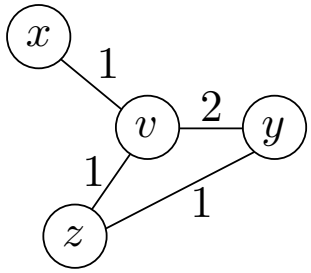


Figure 3.1: Contraction of v. If the pair x, y is considered first, a shortcut $\{x, y\}$ with weight 3 is inserted. If the pair x, z is considered first, an edge $\{x, z\}$ with weight 2 is inserted. This shortcut is part of a witness $x \to z \to y$ for the pair x, y. The shortcut $\{x, y\}$ is thus *not* added if the pair x, z is considered first.

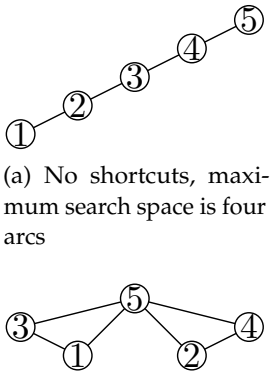
Given a fixed weight w, one can exploit that in many applications it is sufficient to only preserve all shortest path distances [GSSV12]. *Weighted vertex contraction* of a vertex v in the graph G is defined as the operation of removing v and inserting (a minimum number) of shortcuts among the neighbors of v to obtain a graph G' such that $\text{dist}_G(x, y) = \text{dist}_{G'}(x, y)$ for all vertices $x \neq v$ and $y \neq v$. To compute G', one iterates over all pairs of neighbors x, y of v increasing by $\text{dist}_G(x, y)$. For each pair one checks whether a xy-path of length $\text{dist}_G(x, y)$ exists in $G \backslash \{v\}$, i. e., one checks whether removing v destroys the xy-shortest path. This check is called *witness search* [GSSV12] and the xy-path is called *witness*, if it exists. If a witness is found, the considered vertex pair is skipped and no shortcut added. Otherwise, if an edge $\{x, y\}$ already exists, its weight is decreased to $\text{dist}_G(x, y)$, or a new shortcut edge with that weight is added to G. This new shortcut edge is considered in witness searches for subsequent neighbor pairs as part of G. If shortest paths are not unique, it is important to iterate over the pairs increasing by $\text{dist}_G(x, y)$, because otherwise more edges than strictly necessary can be inserted: Shorter shortcuts can make longer shortcuts superfluous. However, if we insert the shorter shortcut after the longer ones, the witness search will not consider them. See Figure 3.1 for an example. (This effect was independently observed by [RT10] in a different setting.) Note that the witness searches are expensive and therefore the witness search is usually aborted after a certain number of steps [GSSV12]. If no witness was found, we assume that none exists and add a shortcut. This does not affect the correctness of the technique but might result in slightly more shortcuts than necessary. To distinguish, *perfect witness search* is without such a one-sided error.

For an order π and a weight w the *weighted core graph* $G_{w,\pi,i}$ is obtained by contracting all vertices $\pi(1) \dots \pi(i-1)$. The original graph G augmented by the set of weighted shortcuts is called a *weighted contraction hierarchy* $G^*_{w,\pi}$. The corresponding upward directed graph is denoted by $G^{\wedge}_{w,\pi}$.

The search space $SS(v)$ of a vertex v is the subgraph of $G^{\wedge}_{\pi}$ (respectively $G^{\wedge}_{w,\pi}$) reachable from v. For every vertex pair s and t, it has been shown that a shortest up-down path must exist. This up-down path can be found by running a bidirectional search from s restricted to $SS(s)$ and from t restricted to $SS(t)$ [GSSV12].

TREE-DECOMPOSITIONS, SPARSE MATRICES AND MINIMUM FILL-IN. Customizable speedup techniques for shortest path queries are a very recent development but the idea to order vertices along nested dissection orders is significantly older. To the best of our knowledge the idea first appeared in 1973 in [Geo73] and was refined in [LRT79]. They use nested dissection orders to reorder the columns and rows of *sparse matrices* to ensure that Gaussian elimination preserves as many zeros as possible. From the matrix they derive a graph and show that vertex contraction in this graph corresponds to Gaussian variable elimination. Inserting an extra edge in the graph destroys a zero in the matrix. The additional edges are called the *fill-in*. The *minimum fill-in problem* asks for a vertex order that results in a minimum number of extra arcs. In CH terminology these extra edges are called shortcuts. The supergraph constructed by adding the additional edges is a *chordal graph*. The *treewidth* of a graph G can be defined using chordal supergraphs: For every supergraph consider the number of vertices in the maximum clique minus one. The treewidth of a graph G is the minimum of this number over all chordal supergraphs of G. This establishes a relation between sparse matrices and treewidth and in consequence with CHs. We refer to [Bod07] and [Bod93] for an introduction to the broad field of treewidth and tree decompositions.

Minimizing the number of extra edges, i. e., minimizing the fill-in, is NP-hard [Yan81] but fixed parameter tractable in the number of extra edges [KST99]. Note, however, that from the CH point of view, optimizing the number of extra edges, i. e., the number of shortcuts, is not the sole optimization criterion. Consider for example a path graph as depicted in Figure 3.2: Optimizing the CH search space and the number of shortcuts are competing criteria. A tree relevant in the theory of treewidth is the *elimination tree*. [BCRW13] have shown that the maximum search space size in terms of vertices corresponds to the height of this elimination tree. Unfortunately, minimizing the elimination tree height is also NP-hard [Pot88]. For planar graphs, it has been shown that the number of additional edges is in $O(n \log n)$ [GT86]. However, this does not imply a $O(\log n)$ search space bound in terms of vertices as search spaces can share vertices.



(a) No shortcuts, maximum search space is four arcs

(b) Two shortcuts, maximum search space is two arcs

Figure 3.2: Contraction Hierarchies for a path graph.

A graph is *chordal* if for every cycle of at least four vertices there exists a pair of vertices that are non-adjacent in the cycle but are connected by an edge. An alternative characterization is that a vertex order π exists such that for every i the neighbors of $\pi(i)$

in $G_{\pi,i}$, i. e., the core graph before the contraction of $\pi(i)$, form a clique [FG65]. Such an order is called a *perfect elimination order*. Another way to formulate this characterization in CH terminology is as follows: A graph is chordal if and only if a contraction order exists such that the CH construction without witness search does not insert any shortcuts. A chordal supergraph can be obtained by adding the CH shortcuts.

The elimination tree $T_{G,\pi}$ is a tree directed towards its root $\pi(n)$. The parent of vertex $\pi(i)$ is its upward neighbor $v \in N_u(\pi(i))$ of minimal rank $\pi^{-1}(v)$. Note that this definition already yields a straightforward algorithm for constructing the elimination tree. As shown in [BCRW13], the set of vertices on the path from v to $\pi(n)$ is the set of vertices in $SS(v)$. Computing a contraction hierarchy search of graph G without witness consists of computing a chordal supergraph G^*_π with perfect elimination order π. Then, the height of the elimination tree corresponds to the maximum number of vertices in the search space. Note that the elimination tree is only well-defined for undirected unweighted graphs.

3.1.2 *Preprocessing*

3.1.2.1 *Metric-Independent Order*

The metric-dependent orders presented in the previous section lead to very good results on road graphs with travel time metric. However, the results for the distance metric are not as good and the orders are completely impracticable to compute Contraction Hierarchies without witness search as our experiments in Section 3.1.5 show. To support metric-independence, we therefore use *nested dissection* orders as suggested in [BCRW13] or ND-orders for short. An order π for G is computed recursively by determining a balanced separator S of minimum cardinality that splits G into two parts induced by the vertex sets A and B. The vertices of S are assigned to $\pi(n-|S|+1)\dots\pi(n)$ in an arbitrary order. Orders π_A and π_B are computed recursively and assigned to $\pi(1)\dots\pi(|A|)$ and $\pi(|A|+1)\dots\pi(|A|+|B|)$, respectively. The base case of the recursion is reached when the subgraphs are empty. Computing ND-orders requires good graph bisectors, which in theory is NP-hard. However, recent years have seen heuristics that solve the problem very well even for continental road graphs [DGRW11; DGRW12; SS13]. This justifies assuming in our particular context that an efficient bisection oracle exists. We experimentally examine the performance of nested dissection orders computed by NDMetis [KK99] and KaHIP [SS13] in Section 3.1.5. After having obtained the nested dissection order we reorder the in-memory vertex IDs of the input graph accordingly, i. e., the contraction order of the reordered graph is the identity function. This improves cache locality and we have seen a resulting acceleration of a factor 2 to 3 in query times.

3.1.2.2 *Constructing the Contraction Hierarchy*

In this section, we describe how to efficiently compute the hierarchy $G_\pi^\wedge$ for a given graph G and order π. Weighted contraction hierarchies are commonly constructed using a dynamic adjacency array representation of the core graph. Our experiments show that this approach also works for the unweighted case, however, requiring more computational and memory resources because of the higher growth in shortcuts. It has been proposed by [Zei13] to use hash-tables on top of the dynamic graph structure to improve speed but at the cost of significantly increased memory consumption. In this section, we show that the contraction hierarchy construction can be done significantly faster on unweighted and undirected graphs. Note that in our toolchain, graph weights and arc directions are accounted for during the customization phase.

Denote by n the number of vertices in G (and $G_\pi^\wedge$), by m the number of edges in G, by $\hat{m}$ the number of arcs in $G_\pi^\wedge$, and by $\alpha(n)$ the inverse $A(n,n)$ Ackermann function. For simplicity we assume that G is connected. Our approach enumerates all arcs of $G_\pi^\wedge$ in $O(\hat{m}\,\alpha(n))$ running time and has a memory consumption in $O(m)$. To store the arcs of $G_\pi^\wedge$, additional space in $O(\hat{m})$ is needed. The approach is heavily based upon the method of the quotient graph [GL78]. To the best of our knowledge it has not yet been applied in the context of route planning and there exists no complexity analysis for the specific variant employed by us. Therefore we discuss both the approach and present a running time analysis in the remainder of the section.

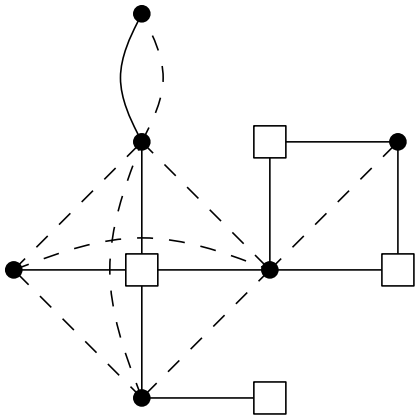


Figure 3.3: Dots represent vertices in G' and H. Squares are additional super vertices in H. Solid edges are in H and dashed ones in G'. Notice how the neighbors of each super vertex in H form a clique in G'. Furthermore, there are no two adjacent super vertices in H, i. e., they form an independent set.

Recall that to compute the contraction hierarchy $G_\pi^\wedge$ from a given input graph G and order π, one iteratively contracts each vertex, adding shortcuts between its neighbors. Let $G' = G_{\pi,i}$ be the core graph in iteration i. We do not store G' explicitly but employ a special data structure called *contraction graph* for efficient contraction and neighborhood enumeration. The contraction graph H contains both yet uncontracted core vertices as well as an independent set of virtually contracted *super vertices*, see Figure 3.3 for an illustration. These super vertices enable us to avoid the overhead of dynamically adding shortcuts to G'. For each vertex in H we store a marker bit indicating whether it is a super vertex. Note that G' can be obtained by contracting all super vertices in H.

CONTRACTING VERTICES. A vertex x in G' is contracted by turning it into a super vertex. However, creating new super vertices can violate the independent set property. We restore it by merging neighboring

super vertices: Denote by y a super vertex that is a neighbor of x. We rewire all edges incident to y to be incident to x and remove y from H. To support efficiently merging vertices in H, we store a linked list of neighbors for each vertex. When merging two vertices we link these lists together. Unfortunately, combining these lists is not enough as the former neighbors z of y still have y in their list of neighbors. We therefore further maintain a union-find data structure: Initially all vertices are within their own set. When merging x and y, the sets of x and y are united. We chose x as representative as y was deleted.[1] When z enumerates its neighbors, it finds a reference to y. It can then use the union-find data structure to determine that the representative of y's set is x. The reference in z's list is thus read as pointing to x.

It is possible that merging vertices can create multi-edges and loops. For example, consider that the neighborhood list of y contains x. After merging, the united list of x will therefore contain a reference to x. Similarly, it will contain a reference to y, which after looking up the representative is actually x. Two loops are thus created at x per merge. Furthermore, consider a vertex z that is a neighbor of both y and x. In this case the neighborhood list of x will contain two references to z. These multi-edges and loops need to be removed. We do this lazily and remove them in the neighborhood enumeration instead of removing them in the merge operation.

ENUMERATING NEIGHBORS. Suppose that we want to enumerate the neighbors of a vertex x in H. Note that x's neighborhood in H differs from its neighborhood in G'. The neighborhood of x in H can contain super vertices, as super vertices are only contracted in G'. We maintain a boolean marker that indicates which neighbors have already been enumerated. Initially no marker is set. We iterate over x's neighborhood list. For each reference we lookup the representative v. If v was already marked or is x, we remove the reference from the list. If v was not marked and is not x, we mark it and report it as a neighbor. After the enumeration we reset all markers by enumerating the neighbors again.

However, during the execution of our algorithm, we are not interested in the neighborhood of x in H, but we want the neighborhood of x in G', i. e., the algorithm should not list super vertices. Our algorithm conceptually first enumerates the neighborhood of x and then contracts x. We actually do this in reversed order. We first contract x. After the contraction x is a super vertex. Because of the independent set property, we know that x has no super vertex neighbors in H.

1 Or alternatively, we can let the union-find data structure choose the new representative. We then denote by x the new representative and by y the other vertex. In this variant, it is possible that the new x is the old y, which can be confusing. For this reason, we describe the simpler variant, where x is always chosen as representative and thus x always refers to the same vertex.

We can thus enumerate x's neighbors in H and exploit that in this particular situation the neighborhoods of x in G' and H coincide.

PERFORMANCE ANALYSIS. As there are no memory allocations, it is clear that the working space memory consumption is in $O(m)$. Proving a running time in $O(\hat{m}\alpha(n))$ is less clear. Denote by $d(x)$ the degree of x just before x is contracted. $d(x)$ coincides with the upward degree of x in $G_\pi^\wedge$ and thus $\sum d(x) = \hat{m}$. We first prove that we can account for the neighborhood cleanup operations outside of the actual algorithm. This allows us to assume that they are free within the main analysis. We then show that contracting a vertex x and enumerating its neighbors is in $O(d(x)\alpha(n))$. Processing all vertices thus has a running time in $O(\hat{m}\alpha(n))$.

The neighborhood list of x can contain duplicated references and thus its length can be larger than the number of neighbors of x. Further, for each entry in the list, we need to perform a union find lookup. The costs of a neighborhood enumeration can thus be larger than $O(d(x)\alpha(n))$. Fortunately, the first neighborhood enumeration compactifies the neighborhood list and thus every subsequent enumeration runs in $O(d(x)\alpha(n))$. Removing a reference has a cost in $O(\alpha(n))$. Our algorithm never adds references. Initially there are $\Theta(m)$ references. The total costs for removing references over the whole algorithm are thus in $O(m\alpha(n))$. As our graph is assumed to be connected, we have that $m \in O(m')$ and therefore $O(m\alpha(n)) \subseteq O(\hat{m}\alpha(n))$. We can therefore assume that removing references is free within the algorithm. As removing a reference is free, we can assume that even the first enumeration of the neighbors of x is within $O(d(x)\alpha(n))$. Merging two vertices consists of redirecting a constant number of references within a linked list. The merge operation is thus in $O(1)$.

Our algorithm starts by enumerating all neighbors of x to determine all neighboring super vertices in $O(d(x)\alpha(n))$ time. There are at most $d(x)$ neighboring super vertices and therefore the costs of merging all super vertices into x is in $O(d(x))$. We subsequently enumerate all neighbors a second time to output the arcs of $G_\pi^\wedge$. The costs of this second enumeration is also within $O(d(x)\alpha(n))$. The overall algorithm thus runs in $O(\hat{m}\alpha(n))$ time, since $\sum d(x) = \hat{m}$. This completes the proof for our analysis.

ADJACENCY ARRAY. While the described algorithm is efficient in theory, linked lists cause too many cache misses in practice. We therefore implemented a hybrid of a linked list and an adjacency array, which has the same worst case performance, but is more cache-friendly in practice. An element in the linked list does not only hold a single reference, but a small set of references organized as small arrays called blocks. The neighbors of every original vertex form a single block. The initial linked neighborhood list are therefore composed of a single

block. We merge two vertices by linking their blocks together. If all references are deleted from a block, we remove it from the list.

3.1.2.3 *Enumerating Triangles*

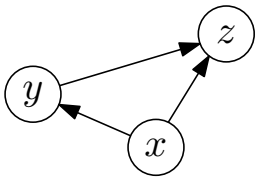


Figure 3.4: A triangle in $G_\pi^\triangle$. The triple $\{x, y, z\}$ is a lower triangle of the arc (y, z), an intermediate triangle of the arc (x, z), and an upper triangle of the arc (x, y).

A triangle $\{x, y, z\}$ is a set of three adjacent vertices. A triangle can be an upper, intermediate or lower triangle with respect to an arc (x, y), as illustrated in Figure 3.4. A triangle $\{x, y, z\}$ is a *lower triangle* of (y, z) if x has the lowest rank among the three vertices. Similarly, $\{x, y, z\}$ is a *upper triangle* of (x, y) if z has the highest rank and $\{x, y, z\}$ is a *intermediate triangle* of (x, z) if y's rank is between the ranks of x and z. The triangles of an edge (a, b) can be characterized using the upward N_u and downward N_d neighborhoods of a and b. There is a lower triangle $\{a, b, c\}$ of an arc (a, b) if and only if $c \in N_d(a) \cap N_d(b)$. Similarly, there is an intermediate triangle $\{a, b, c\}$ of an arc (a, b) with $\pi^{-1}(a) < \pi^{-1}(b)$ if and only if $c \in N_u(a) \cap N_d(b)$ and an upper triangle $\{a, b, c\}$ of an arc (a, b) if and only if $c \in N_u(a) \cap N_u(b)$. The triangles of an arc can thus be enumerated by intersecting the neighborhoods of the arc's endpoints.

Efficiently enumerating all lower triangles of an arc is an important base operation of the customization (Section 3.1.3) and path unpacking algorithms (Section 3.1.4). It can be implemented using adjacency arrays or accelerated using extra preprocessing. Note that in addition to the vertices of a triangle we are interested in the IDs of the participating arcs as we need these to access the metric of an arc.

BASIC TRIANGLE ENUMERATION. Triangles can be efficiently enumerated by exploiting their characterization using neighborhood intersections. We construct an upward and a downward adjacency array for $G_\pi^\triangle$, where incident arcs are ordered by their head respectively tail vertex ID. The lower triangles of an arc (x, y) can be enumerated by simultaneously scanning the downward neighborhoods of x and y to determine their intersection. Intermediate and upper triangles are enumerated analogously using the upward adjacency arrays. For later access to the metric of an arc, we also store each arc's ID in the adjacency arrays. This approach requires space proportional to the number of arcs in $G_\pi^\triangle$.

TRIANGLE PREPROCESSING. Instead of merging the neighborhoods on demand to find all lower triangles, we propose to create a *triangle adjacency array* structure that maps the arc ID of (x, y) to the set of pairs of arc IDs of (z, x) and (z, y) for every lower triangle $\{x, y, z\}$ of (x, y). This requires space proportional to the number of triangles t in $G_\pi^\triangle$, but allows for a very fast access. Analogous structures enable efficient enumeration of all upper or intermediate triangles.

HYBRID APPROACH. For less well-behaved graphs the number of triangles t can significantly outgrow the number of arcs in $G_\pi^\wedge$. In the worst case G is the complete graph and the number of triangles t is in $\Theta(n^3)$ whereas the number of arcs is only in $\Theta(n^2)$. It can thus be prohibitive to store a list of all triangles. We therefore propose a hybrid approach, where only some triangles are precomputed.

The basic triangle enumeration algorithm computes the intersection of two lower neighborhoods and thus encounters two cases: Either a neighbor is common (yielding a triangle that has to be processed) or it is not. With precomputed lower triangles, this second case can be eliminated, resulting in faster enumeration times.

Now, for arcs where both endpoints have a high level, many (of their numerous lower triangles) are contained in the top level cliques of the CH. As a consequence, for them the ratio of common neighbors to non-common neighbors is very high. For lower level arcs, on the other hand, this ratio is often lower. This gives precomputed triangles for these lower levels the greater benefit over basic triangle enumeration. Hence, we propose to only precompute triangles for those arcs (u, v) where the level of u is below a certain threshold. The threshold is a tuning parameter that trades space for time.

COMPARISON WITH CRP. Triangle preprocessing has similarities with micro and macrocode in CRP [DW13]. In the following, we compare the space consumption of these two approaches against our lower triangles preprocessing scheme. However, note that at this stage we do not yet consider travel direction on arcs. Hence, let t be the number of undirected triangles and m be the number of arcs in $G_\pi^\wedge$; further let t' be the number of directed triangles and m' be the number of arcs used in [DW13]. If every street is a one-way street, then $m' = m$ and $t' = t$; otherwise, without one-way streets, $m' = 2m$ and $t' = 2t$.

Microcode stores an array of triples of pointers to the arc weights of the three arcs in a directed triangle, i. e., it stores the equivalent of $3t'$ arc IDs. Computing the exact space consumption of macrocode is more difficult. However, it is easy to obtain a lower bound: Macrocode must store for every triangle at least the pointer to the arc weight of the upper arc. This yields a space consumption equivalent to *at least* t' arc IDs. In comparison, our approach stores for each triangle the arc IDs of the two lower arcs. Additionally, the index array of the triangle adjacency array, which maps each arc to the set of its lower triangles, maintains $m + 1$ entries. Each entry has a size equivalent to an arc ID. Our total memory consumption is thus $2t + m + 1$ arc IDs.

Hence, our approach always requires less space than microcode. It has similar space consumption as macrocode if one-way streets are rare, otherwise it needs at most twice as much data. However, the main advantage of our approach over macrocode is that it allows for

random access, which is crucial in the algorithms presented in the following sections.

3.1.3 *Customization*

Up to now we only considered the metric-independent first preprocessing phase. In this section, we describe the second, metric-dependent preprocessing phase, known as customization. That is, we show how to efficiently extend the weights of the input graph to a corresponding metric with weights for all arcs in $G_\pi^\wedge$. We consider three different distances between the vertices: We refer to $\mathrm{dist}_I(s,t)$ as the shortest st-path distance in the input graph G. With $\mathrm{dist}_{UD}(s,t)$ we denote the shortest st-path distance in $G_\pi^\wedge$ when only considering up-down paths. Finally, let $\mathrm{dist}_A(s,t)$ be the shortest st-path distance in G_π^*, i. e., when allowing arbitrary not-necessarily up-down paths in $G_\pi^\wedge$.

For correctness of the CH query algorithms (cf. Section 3.1.4) it is necessary that between any pair of vertices s and t a shortest up-down st-path in $G_\pi^\wedge$ exists with the same distance as the shortest st-path in the input graph G. In other words, $\mathrm{dist}_I(s,t) = \mathrm{dist}_A(s,t) = \mathrm{dist}_{UD}(s,t)$ must hold for all vertices s and t. We say that a metric that fulfills $\mathrm{dist}_I(s,t) = \mathrm{dist}_A(s,t)$ *respects* the input weights. If additionally $\mathrm{dist}_A(s,t) = \mathrm{dist}_{UD}(s,t)$ holds, we call the metric *customized*. Note that customized metrics are not necessarily unique. However, there is a special customized metric, called *perfect* metric m_P, where for every arc (x,y) in $G_\pi^\wedge$ the weight of this arc $m_P(x,y)$ is equal to the shortest path distance $\mathrm{dist}_I(x,y)$. We optionally use the perfect metric to perform perfect witness search.

Constructing a respecting metric is trivial: Assign to all arcs of $G_\pi^\wedge$ that already exist in G their input weight and to all other arcs $+\infty$. Computing a customized metric is less trivial. We therefore describe in Section 3.1.3.1 the basic customization algorithm that computes a customized metric m_C given a respecting one. Afterwards, we describe the perfect customization algorithm that computes the perfect metric m_P given a customized one (i. e., m_C). Finally, we show how to employ the perfect metric to perform a perfect witness search.

3.1.3.1 *Basic Customization*

A central notion of the basic customization algorithm is the *lower triangle inequality*, which is defined as follows. A metric m_C fulfills it if for all lower triangles $\{x,y,z\}$ of each arc (x,y) of $G_\pi^\wedge$, it holds that $m_C(x,y) \leqslant m_C(x,z) + m_C(z,y)$. We show that every respecting metric that also fulfills this inequality is customized. Our algorithm exploits this by transforming the given respecting metric in a coordinated way that maintains the respecting property and ensures that the lower triangle inequality holds. The resulting metric is thus customized. We first describe the algorithm and prove that the resulting metric

is respecting and fulfills the inequality. We then prove that this is sufficient for the resulting metric to be customized.

Our algorithm iterates over all arcs $(x, y) \in G_\pi^\wedge$ ordered *increasingly* by the rank of x in a bottom-up fashion. For each arc (x, y), it enumerates all lower triangles $\{x, y, z\}$ and checks whether the path $x \to z \to y$ is shorter than the path $x \to y$. If this is the case, then it decreases $m_C(x, y)$ so that both paths are equally long. Formally, it performs for every arc (x, y) the operation $m_C(x, y) \leftarrow \min\{m_C(x, y), m_C(x, z) + m_C(z, y)\}$. Note that this operation never assigns values that do not correspond to a path length and therefore m_C remains respecting. By induction over the vertex levels, we can show that after the algorithm is finished, the lower triangle inequality holds for every arc, i. e., for every arc (x, y) and lower triangle $\{x, y, z\}$ the inequality $m_C(x, y) \leqslant m_C(x, z) + m_C(z, y)$ holds. The key observation is that by construction the rank of z must be strictly smaller than the ranks of x and y. The final weights of $m_C(x, z)$ and $m_C(z, y)$ have therefore already been computed when considering (x, y). In other words, when the algorithm considers the arc (x, y), the weights $m_C(x, z)$ and $m_C(z, y)$ are guaranteed to be final.

Theorem 1. *Every respecting metric that additionally fulfills the lower triangle inequality is customized.*

Proof. We need to show that between any pair of vertices s and t a shortest up-down st-path exists. As we assumed for simplicity that G is connected, there always exists a shortest not-necessarily up-down path from s to t. Either this is an up-down path, or a subpath $x \to z \to y$ with $\pi^{-1}(x) > \pi^{-1}(z)$ and $\pi^{-1}(y) > \pi^{-1}(z)$ must exist. As z is contracted before x and y, an edge $\{x, y\}$ must exist. Because of the lower triangle inequality, we further know that $m(x, y) \leqslant m(x, z) + m(z, y)$ and thus replacing $x \to z \to y$ by $x \to y$ does not make the path longer. Either the path is now an up-down path or we can apply the argument iteratively. As the path has only a finite number of vertices, this is guaranteed to eventually yield the up-down path required by the theorem and thus this completes the proof. □

3.1.3.2 *Perfect Customization*

Given a customized metric m_C, we want to compute the perfect metric m_P. We first copy all values of m_C into m_P. Our algorithm then iterates over all arcs (x, y) *decreasing* by the rank of x in a top-down fashion. For every arc it enumerates all intermediate and upper triangles $\{x, y, z\}$ and checks whether the path over z is shorter and adjusts the value of $m_P(x, y)$ accordingly, i. e., it performs $m_P(x, y) \leftarrow \min\{m_P(x, y), m_P(x, z) + m_P(z, y)\}$. After all arcs have been processed m_P is the perfect metric, as is shown in the following theorem.

Theorem 2. *After the perfect customization, $m_P(x, y)$ corresponds to the shortest xy-path distance for every arc (x, y), i. e., m_P is the perfect metric.*

Proof. We have to show that after the algorithm has finished processing a vertex x, all of its outgoing arcs in $G_\pi^\wedge$ are weighted by the shortest path distance. We prove this by induction over the level of the processed vertices. The top-most vertex is the only vertex in the top level. It does not have any upward arcs and thus the algorithm does not have anything to do. This forms the base case of the induction. In the inductive step, we assume that all vertices with a strictly higher level have already been processed. As consequence, we know that the upward neighbors of x form a clique weighted by shortest path distances. Denote these neighbors by y_i. The situation is depicted in Figure 3.5. The weights of the y_i encode a complete shortest path distance table between the upward neighbors of x.

Pick some arbitrary arc (x, y_j). We show the correctness of our algorithm by proving that either $m_C(x, y_j)$ is already the shortest path distance or a neighbor $y_k \in N_u(x)$ must exist such that $x \to y_k \to y_j$ is a shortest up-down path. For the rest of this paragraph assume the existence of y_k, we prove its existence in the next paragraph. If $m_C(x, y_j)$ is already the shortest xy_j-path distance, then enumerating triangles will not change $m_C(x, y_j)$ and is thus correct. If $m_C(x, y_j)$ is not the shortest xy_j-path distance, then enumerating all intermediate and upper triangles of (x, y_j) is guaranteed to find the $x \to y_k \to y_j$ path and thus the algorithm is correct. The upper triangles correspond to paths with $\ell(y_k) > \ell(y_j)$ while the intermediate triangles to paths with $\ell(y_k) < \ell(y_j)$.

It remains to show that the $x \to y_k \to y_j$ shortest up-down path actually exists. As the metric is customized at every moment during the perfect customization, we know that a shortest up-down xy_j-path K exists. As K is an up-down path, we can conclude that the second vertex of K must be an upward neighbor of x. We denote this neighbor by y_k . K thus has the following structure: $x \to y_k \to \ldots \to y_j$. As y_k has a higher rank than x, $m_P(y_k, y_j)$ is guaranteed to be the shortest $y_k y_j$-path distance, and therefore we can replace the $y_k \to \ldots \to y_j$ subpath of K by $y_k \to y_j$, and we have proven that the required $x \to y_k \to y_j$ shortest up-down path exists. This completes the proof. □

3.1.3.3 *Perfect Witness Search*

Using the perfect customization algorithm, we can efficiently compute the weighted CH with a minimum number of arcs with respect to the same contraction order. We present two variants of our algorithm. The first variant consists of removing each arc (x, y) whose weight $m_C(x, y)$ after basic customization does not correspond to the shortest xy-path distance $m_P(x, y)$. While simple and correct, this variant does not remove as many arcs as possible, if a pair of vertices a and b exists in the input graph such that there are multiple shortest ab-paths. The

second variant[2] also removes these additional arcs. An arc (x, y) is removed if and only if an upper or intermediate triangle $\{x, y, z\}$ exists such that the shortest path from x over z to y is no longer than the shortest xy-path. However, before we can prove the correctness of the second variant, we need to introduce some technical machinery, which will also be needed in the correctness proof of the stalling query algorithm. We define the "height" of a not-necessarily up-down path in G^*_π. We show that with respect to every customized metric, for every path that is not up-down, an up-down path must exist that is strictly higher and is no longer.

VARIANT FOR GRAPHS WITH UNIQUE SHORTEST PATHS. The first algorithm variant consists of removing all arcs (x, y) from the CH for which $m_P(x, y) \neq m_C(x, y)$. It is optimal if shortest path are unique in the input graph, i. e., between every pair of vertices a and b there is only one shortest ab-path. This simple algorithm is correct as the following theorem shows.

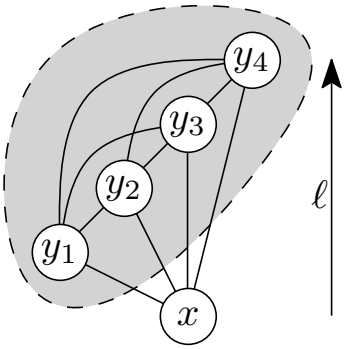


Figure 3.5: The vertices $y_1 \dots y_4$ denote the upper neighborhood $N_u(x)$ of x. They form a clique (grey area) because x was contracted first. As $\ell(x) < \ell(y_j)$ for every j, we know by the induction hypothesis that the arcs in this clique are weighted by shortest path distances. We therefore have an all-pair shortest path distance table among all y_j. We have to show that using this information we can compute shortest path distances for all arcs outgoing of x.

Theorem 3. *If the input graph has unique shortest paths between all pairs of vertices, then we can remove an arc (x, y) from the CH if and only if $m_P(x, y) \neq m_C(x, y)$.*

Proof. We need to show that after removing all arcs, there still exists a shortest up-down path between every pair of vertices s and t. We know that before removing any arc a shortest up-down st-path K exists. We show that no arc of K is removed and thus K also exists after removing all arcs. Every subpath of K must be a shortest path as K is a shortest path. Every arc of K is a subpath. However, we only remove arcs such that $m_P(x, y) \neq m_C(x, y)$, i. e., which are not shortest paths.

To show that no further arcs can be removed we need to show that if $m_P(x, y) = m_C(x, y)$, then the path $x \to y$ is the only shortest up-down path. Denote the $x \to y$ path by Q. Suppose that another shortest up-down path R existed. R must be different than Q, i. e., a vertex z must exist that lies on R but not on Q. As z must be reachable from x, we know that z is higher than x. Unpacking the path Q in the input graph yields a path where x and y are the highest ranked vertices and thus this unpacked path cannot contain z. Unpacking R yields a path that contains z and is therefore different. Both paths are shortest paths from x to y in the input graph. This contradicts the assumption that shortest paths are unique. We have thus proven that,

2 Note that the second algorithm variant exploits that we defined weights as being non-zero. If zero weights are allowed, it may remove too many arcs. A workaround consists of replacing all zero weights with a very small but non-zero weight.

if the input graph has unique shortest paths, we can remove an arc (x, y) if and only if $m_P(x, y) \neq m_C(x, y)$. □

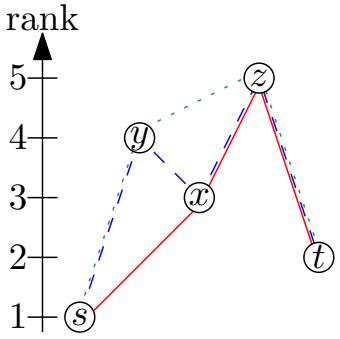


Figure 3.6: The rank sequence of the solid red path is $[3, 2, 1]$. The 3 is the minimum of the ranks of the endpoints of the $\{x, z\}$ edge. Similarly, the 2 is induced by the $\{z, t\}$ edge and the 1 by the $\{s, x\}$ edge. The rank sequence of the blue dashed path is $[3, 3, 2, 1]$ and the rank sequence of the green dotted path is $[4, 2, 1]$. The solid red path is the lowest followed by the blue dashed path and the green dotted path is the highest.

VARIANT FOR GENERAL GRAPHS. If shortest paths are not unique in the original graph, using the first variant of our algorithm still yields correct results. However, it is possible that some arcs are not removed that could be removed. Our second algorithm variant does not have this weakness. It removes all arcs (x, y) for which an intermediate or upper triangle $\{x, y, z\}$ exists such that $m_P(x, y) = m_P(x, z) + m_P(z, y)$. These arcs can efficiently be identified while running the perfect customization algorithm. An arc (x, y) is marked for removal if an upper or intermediate triangle $\{x, y, z\}$ with $m_C(x, y) \geqslant m_C(x, z) + m_C(z, y)$ is encountered. However, before we can prove the correctness of the second variant, we need to introduce some additional technical machinery.

We want to order paths by "height". To achieve this, we first define for each path K in G^*_π its *rank sequence*. We order paths by comparing the rank sequences lexicographically. Denote by v_i the vertices in K. For each edge $\{v_i, v_{i+1}\}$ in K the rank sequence contains $\min\{\pi^{-1}(v_i), \pi^{-1}(v_{i+1})\}$. The numbers in the rank sequences are sorted in non-increasing order. Two paths have the same height if one rank sequence is a prefix of the other. Otherwise we compare the rank sequences lexicographically. This ordering is illustrated in Figure 3.6. We prove the following technical lemma:

Lemma 1. *Let m_C be some customized metric. For every st-path K that is no up-down path, an up-down st-path Q exists such that Q is strictly higher than K and Q is no longer than K with respect to m_C.*

Proof. Denote by v_i the vertices on the path K. As K is no up-down path, there must exist a vertex v_i on K that has lower ranks than its neighbors v_{i-1} and v_{i+1}. v_{i-1} and v_{i+1} are different vertices because they are part of a shortest path and zero weights are not allowed. Further, as v_i is contracted before its neighbors, there must be a edge between v_{i-1} and v_{i+1}. As the metric is customized, $m_C(v_{i-1}, v_{i+1}) \leqslant m_C(v_{i-1}, v_i) + m_C(v_i, v_{i+1})$ must hold. We can bypass v_i by replacing the subpath (v_{i-1}, v_i, v_{i+1}) with the single arc (v_{i-1}, v_{i+1}) without making the path longer. Denote this new path by R. Clearly, R is higher than K as we replaced $\pi^{-1}(v_i)$ in the rank sequence by $\min\{\pi^{-1}(v_{i-1}), \pi^{-1}(v_{i+1})\}$, which must be larger. Either R is an up-down path or we apply the argument iteratively. In each iteration, the path loses a vertex. Hence, we can guarantee that eventually we obtain an up-down path that is strictly higher than K and not longer. This is the desired up-down path Q. □

Note that this lemma does not exploit any property that is inherent to CHs with a metric-independent contraction ordering and is thus applicable to every CH.

Given this technical lemma, we can prove the correctness of the second variant of our algorithm.

Theorem 4. *We can remove an arc (x, y) if and only if an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$.*

Proof. We need to show that for every pair of vertices s and t a shortest up-down st-path exists, that uses no removed arc. We show that a highest shortest up-down st-path has this property. As the metric is customized, we know that a shortest up-down st-path K exists before removing any arcs. If K does not contain an arc (x, y) for which an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$, then there is nothing to show. Otherwise, we modify K by inserting z between x and y. This does not modify the length of K, but we can no longer guarantee that K is an up-down path. If $\{x, y, z\}$ was an intermediate triangle, then K is still an up-down path. However, it is strictly higher, as we added $\pi^{-1}(z)$ into the rank sequence, which is guaranteed to be larger than $\pi^{-1}(x)$. If $\{x, y, z\}$ was an upper triangle, then K is no longer an up-down path. Fortunately, using Lemma 1 we can transform K into an up-down path, that is no longer and strictly higher. In both cases, the new K is an up-down path or we apply the argument iteratively. As K gets strictly higher in each iteration and the number of up-down paths is finite, we know that we will eventually obtain a shortest up-down st-path where no arc can be removed.

Further, we need to show that if no such triangle exists, then an arc cannot be removed, i. e., we need to show that the only shortest up-down path from x to y is the path consisting only of the (x, y) arc. Assume that no such triangle and a further up-down path Q existed. Q must contain a vertex beside x and y and all vertices in Q must have the rank of x or higher. Consider the vertex z that comes directly after x in Q. As x is contracted before z and y, an arc between z and y must exist. Therefore, a triangle $\{x, y, z\}$ must exist that is an intermediate triangle, if z has a lower rank than y and is an upper triangle, if z has a higher rank than y. However, we assumed that no such triangle can exist. We have thus proven that we can remove an arc (x, y) if and only if an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$. □

3.1.3.4 *Parallelization*

The basic customization can be parallelized by processing the arcs (x, y) that depart within a level in parallel. Between levels, we need to synchronize all threads using a barrier. As all threads only write to the arc they are currently assigned to and only read from arcs processed

in a strictly lower level, we can thus guarantee that no read/write conflict occurs. Hence, no locks or atomic operations are needed.

On most modern processors, perfect customization can be parallelized analogously to basic customization: We iterate over all arcs departing within a level in parallel and synchronize all threads between levels. For every arc (x, y), we enumerate all upper and intermediate triangles and update $m_P(x, y)$ accordingly.

Correctness of this algorithm is not obvious because the exact order in which threads are executed influences intermediate results. Consider two threads A and B. Suppose that thread A processes an arc (x, y_A) at the same time as thread B processes another arc (x, y_B). Furthermore, suppose that thread A updates $m_P(x, y_A)$ at the same moment as thread B enumerates an intermediate or upper (wrt. (x, y_B)) triangle $\{x, y_B, y_A\}$. In this situation it is unclear what value for (x, y_A) thread B will read. However, we will show in the following that our algorithm is correct as long it is guaranteed that thread B will either read the old value or the new value. Then, the end result within each level is always the same, independent of execution order. Overall correctness follows.

In the proof of Theorem 2 we have shown that for every vertex x and arc (x, y_i) either the arc (x, y_i) already has the shortest path distance or an upper or intermediate triangle $\{x, y_i, y_j\}$ exists such that $x \to y_j \to y_i$ is a shortest path. No matter the order in which the threads process the arcs, they do not modify shortest path weights. This implies that the shortest path $x \to y_j \to y_i$ is thus retained, regardless of the execution order. This shortest path is not modified and is guaranteed to exist before any arcs outgoing from the current level are processed. Every thread is thus guaranteed to see it. However, other weights can be modified. Fortunately, this is not a problem as long as we can guarantee that no thread sees a value that is below the corresponding shortest path distance. Therefore, if we can guarantee that thread B either sees the old value or the new value, as is the case on x86 processors, then the algorithm is correct.

Otherwise, if thread B can see some mangled combination of the old value's bits and new value's bits, there are ways to mitigate the problem. To still apply parallelization, however, we would need to use locks or to make sure that all outgoing arcs of x are processed by the same thread.

3.1.3.5 *Directed Graphs*

Up to now we have focused on customizing undirected graphs. If the input graph G is *directed*, our toolchain works as follows: Based on the *undirected unweighted* graph induced by G we compute a vertex ordering π (Section 3.1.2.1), build the upward directed Contraction Hierarchy $G_\pi^\triangle$ (Section 3.1.2.2), and optionally perform triangle preprocessing (Section 3.1.2.3). For customization, however, we consider

two weights per arc in $G_\pi^\wedge$, one for each direction of travel. One-way streets are modeled by setting the weight corresponding to the forbidden traversal direction to ∞. With respect to π we define an upward metric m_u and a downward metric m_d on $G_\pi^\wedge$. For each arc $(x, y) \in G$ in the directed input graph with input weight $w(x, y)$, we set $m_u(x, y) = w(x, y)$ if $\pi^{-1}(x) < \pi^{-1}(y)$, i. e., if x is ordered before y; otherwise, we set $m_d(x, y) = w(x, y)$. All other values of m_u and m_d are set to ∞. In other words, each arc $(x, y) \in G_\pi^\wedge$ of the Contraction Hierarchy has upward weight $m_u(x, y) = w(x, y)$ if $(x, y) \in G$, downward weight $m_d(x, y) = w(y, x)$ if $(y, x) \in G$, and ∞ otherwise.

The basic customization considers both metrics m_u and m_d simultaneously. For every lower triangle $\{x, y, z\}$ of (x, y) it sets

$$m_u(x, y) \leftarrow \min\{m_u(x, y), m_d(x, z) + m_u(z, y)\},$$
$$m_d(x, y) \leftarrow \min\{m_d(x, y), m_u(x, z) + m_d(z, y)\}.$$

The perfect customization can be adapted analogously. For every intermediate triangle $\{x, y, z\}$ of (x, y) the perfect customization sets

$$m_u(x, y) \leftarrow \min\{m_u(x, y), m_u(x, z) + m_u(z, y)\},$$
$$m_d(x, y) \leftarrow \min\{m_d(x, y), m_d(x, z) + m_d(z, y)\}.$$

Similarly, for every upper triangle $\{x, y, z\}$ of (x, y) the perfect customization sets

$$m_u(x, y) \leftarrow \min\{m_u(x, y), m_u(x, z) + m_d(z, y)\},$$
$$m_d(x, y) \leftarrow \min\{m_d(x, y), m_d(x, z) + m_u(z, y)\}.$$

The perfect witness search might need to remove an arc only in one direction. It therefore produces, just as in the original CHs, two search graphs: an upward search graph and a downward search graph. The forward search in the query phase is limited to the upward search graph and the backward search to the downward search graph, just as in the original CHs. The arc (x, y) is removed from the upward search graph if and only if an intermediate triangle $\{x, y, z\}$ with $m_u(x, y) = m_u(x, z) + m_u(z, y)$ exists or an upper triangle $\{x, y, z\}$ with $m_u(x, y) = m_u(x, z) + m_d(z, y)$ exists. Analogously, the arc (x, y) is removed from the downward search graph if and only if an intermediate triangle $\{x, y, z\}$ with $m_d(x, y) = m_d(x, z) + m_d(z, y)$ exists or an upper triangle $\{x, y, z\}$ with $m_d(x, y) = m_d(x, z) + m_u(z, y)$ exists.

3.1.3.6 *Single Instruction Multiple Data*

The weights attached to each arc in the CH can be replaced by an interleaved set of k weights by storing for every arc a vector of k elements. Vectors allow us to customize all k metrics in one go, amortizing triangle enumeration time. Additionally, they allow us to use single instruction multiple data (SIMD) operations. As we use essentially

two metrics to enable directed graphs, we can store both of them in a 2-dimensional vector. This allows us to handle both directions in a single processor instruction. Similarly, if we have k directed input weights we can store them in a $2k$-dimensional vector. (Depending on the width of SIMD registers, we might require more than one SIMD instruction per vector; nonetheless, we would still benefit from amortized triangle enumeration, which is only done once per arc.)

The processor needs to support component-wise minimum and saturated addition, i. e., $a + b = \text{int}_{\max}$ must hold in the case of an overflow. In the case of directed graphs it additionally needs to support efficiently swapping neighboring vector components. A current SSE-enabled processor supports all the necessary operations for 16-bit integer components. For 32-bit integer saturated addition is missing. There are two possibilities to work around this limitation: The first is to emulate saturated-add using a combination of regular addition, comparison and blend/if-then-else instruction. The second consists of using 31-bit weights and use $2^{31} - 1$ as value for ∞ instead of $2^{32} - 1$. The algorithm only computes the saturated addition of two weights followed by taking the minimum of the result and some other weight, i. e., if computing $\min(a + b, c)$ for all weights a, b and c is unproblematic, then the algorithms works correctly. We know that a and b are at most $2^{31} - 1$ and thus their sum is at most $2^{32} - 2$ which fits into a 32-bit integer. In the next step we know that c is at most $2^{31} - 1$ and thus the resulting minimum is also at most $2^{31} - 1$.

3.1.3.7 *Partial Updates*

Until now we have only considered computing metrics from scratch. However, in many scenarios this is overkill, as we know that only a few edge weights of the input graph were changed. It is unnecessary to redo all computations in this case. The ideas employed by our algorithm are somewhat similar to those presented in [GSSV12], but our situation differs as we know that we do not have to insert or remove arcs. Denote by $U = \left\{((x_i, y_i), w_i^{\text{new}})\right\}$ the set of arcs whose weights should be updated, where (x_i, y_i) is the arc ID and w_i^{new} the new weight. Note that modifying the weight of one arc can trigger further changes. However, these new changes have to be at higher levels. We therefore organize U as a priority queue ordered by the level of x_i. We iteratively remove arcs from the queue and apply the change. If new changes are triggered we insert these into the queue. The algorithm terminates once the queue is empty.

Denote by (x, y) the arc that was removed from the queue and by w^{new} its new weight and by w^{old} its old weight. We first have to check whether w^{new} can be bypassed using a lower triangle. For this reason, we iterate over all lower triangles $\{x, y, z\}$ of (x, y) and perform $w^{\text{new}} \leftarrow \min\{w^{\text{new}}, m(z, x) + m(z, y)\}$. Furthermore, if $\{x, y\}$ is an edge in the input graph G, we might have overwritten its weight

with a shortcut weight, which after the update might not be shorter anymore. Hence, we additionally test that w^{new} is not larger than the input weight. If after both checks $w^{\text{new}} = m(x, y)$ holds, then no change is necessary and no further changes are triggered. If w^{old} and w^{new} differ we iterate over all upper triangles $\{x, y, z\}$ of (x, y) and test whether $m(x, z) + w^{\text{old}} = m(y, z)$ holds; if so, the weight of the arc (y, z) must be set to $m(x, z) + w^{\text{new}}$. We add this change to the queue. Analogously we iterate over all intermediate triangles $\{x, y, z\}$ of (x, y) and queue up a change to (z, y) if $m(x, z) + w^{\text{old}} = m(z, y)$ holds.

How many subsequent changes a single change triggers heavily depends on the metric and can significantly vary. Slightly changing the weight of a dirt road has near to no impact whereas changing a heavily used highway segment will trigger many changes. In the game setting such largely varying running times are undesirable as they lead to lag-peaks. We propose to maintain a queue into which all changes are inserted. Every round a fixed amount of time is spent processing elements from this queue. If time runs out before the queue is emptied the remaining arcs are processed in the next round. This way costs are amortized resulting in a constant workload per turn. The downside is that as long the queue is not empty some distance queries will use outdated data. How much time is spent each turn updating the metric determines how long an update needs to be propagated along the whole graph.

3.1.4 *Queries*

In this section, we describe how to answer distance queries, i. e., given a customized metric, we compute the distance in G between two vertices s and t by constructing a shortest up-down st-path in $G_\pi^\triangle$. We further describe how to unpack this path into a shortest path in G.

3.1.4.1 *Basic Query Algorithm*

The basic query runs two instances of Dijkstra's algorithm on $G_\pi^\triangle$ from s and from t. If G is undirected, then both searches use the same metric. Otherwise if G is directed the search from s uses the upward metric m_u and the search from t the downward metric m_d. In either case in contrast to [GSSV12] they operate on the same upward search graph $G_\pi^\triangle$. Once the radius of one of the two searches is larger than the shortest path found so far, we stop the search because we know that no shorter path can exist. We alternate between processing vertices in the forward search and processing vertices in the backward search.

3.1.4.2 *Stalling*

We implemented a basic version of an optimization presented in [GSSV12; SS12a] called stall-on-demand. The optimization exploits

that the shortest strictly upward sv-path in $G_\pi^\wedge$ can be longer than the shortest sv-path in G_π^*, which can go up and down arbitrarily. The search from s only finds upward paths and if we observe that an up-down path exists that is not longer, then we can prune the upward search. Denote by x the vertex removed from the queue. We iterate over all outgoing arcs (x, y) and test whether $d(x) \geqslant m(x, y) + d(y)$ holds. If this is the case for any arc, we prune x.

If $d(x) > m(x, y) + d(y)$ holds, then pruning is correct because all subpaths of shortest up-down paths must be shortest paths and the upward path ending at x is not shortest path as a shorter up-down path through y exists. We can also prune when $d(x) \geqslant m(x, y) + d(y)$, but a different argument is needed. To the best of our knowledge, correctness has so far not been proven for the $d(x) = m(x, y) + d(y)$ case. Notice that we do not exploit any special properties of metric independent orders and thus our proof works for every CH.

Theorem 5. *The upward search can be pruned if* $d(x) \geqslant m(x, y) + d(y)$.

Proof. We show that for every pair of vertices s and t an unprunable, shortest, up-down st-path exists. Our proof relies on Lemma 1 which orders paths by height and states that st-path that are no up-down paths can be transformed into up-down paths that are no longer and strictly higher. We know that some shortest st-path K exists. If K is not pruned, then there is nothing to show. If K is pruned, then there exists a vertex x on K at which the search is pruned. Without loss of generality we assume that x lies on the upward part of K. Further, there must exist a vertex y and a path Q from s to x going through y such that Q is no longer than the sx-prefix of K. Consider the path R obtained by concatenating Q with the xt-suffix of K. R is by construction not longer than K. If x is the highest vertex on K then R is an up-down path and R is strictly higher. Otherwise, R is no up-down path, but using Lemma 1 R can be transformed into an up-down path that is strictly higher and no longer. In both cases, R is no longer and strictly higher. Either, R is unprunable or we apply the argument iteratively. As there are only finitely many up-down paths and each iteration increases the height of R, we eventually end up at an unprunable, shortest, up-down st-path. □

3.1.4.3 *Elimination Tree Query Algorithm*

We precompute for every vertex its parent's vertex ID in the elimination tree. This allows us to efficiently enumerate all vertices in $SS(s)$ and $SS(t)$ at query time, increasingly by rank.

We store two tentative distance arrays $d_f(v)$ and $d_b(v)$. Initially these are all set to ∞. In a first step we compute the lowest common ancestor (LCA) x of s and t in the elimination tree. We do this by simultaneously enumerating all ancestors of s and t by increasing rank until a common ancestor is found. In a second step we iterate

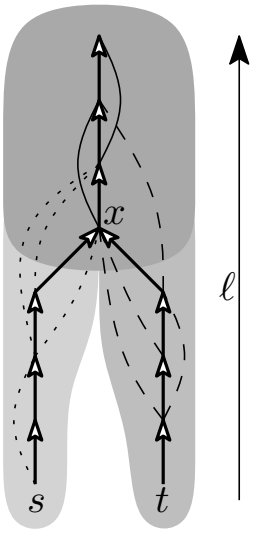


Figure 3.7: The union of the darkgray and lightgray areas is the search space of s. Analogously the union of the darkgray and middlegray areas is the search space of t. The darkgray area is the intersection of both search spaces. The dotted arcs start in the search space of s, but not in the search space of t. Analogously the dashed arcs start in the search space of t, but not in the search space of s. The solid arcs start in the intersection of the two search spaces. The vertex x is the LCA of s and t.

over all vertices y on the tree-path from s to x and relax all forward arcs of such y. In a third step we do the same for all vertices y from t to x in the backward search. In a final fourth step we iterate over all vertices y from x to the root r and relax all forward and backward arcs. Further, in the fourth step we also determine the vertex z that minimizes $d_f(z) + d_b(z)$. A shortest up-down path must exist that goes through z. Knowing z is necessary to determine the shortest path distance and to compute the sequence of arcs that compose the shortest path. In a fifth cleanup step we iterate over all vertices from s and t to the root r to reset all d_f and d_b to ∞. This fifth step avoids having to spend $O(n)$ running time to initialize all tentative distances to ∞ for each query. Consider the situation depicted in Figure 3.7. In the first step the algorithm determines x. In the second step it relaxes all dotted arcs and the tree arcs departing in the lightgray area. In the third step it relaxes all dashed arcs and the tree arcs departing in the middlegray area. In the fourth step the solid arcs and the remaining tree arcs follow.

The elimination tree query can be combined with the perfect witness search. Before pruning any arc, we compute the elimination tree. We then prune the arcs. It is now possible that a vertex has an ancestor in the tree that is not in its pruned search space. However, we can still guarantee that every vertex in the pruned search space is an ancestor and this is enough to prove the query correctness. To avoid relaxing the outgoing arcs of an ancestor outside of the search space, we prune vertices whose tentative distance $d_f(x)$ respectively $d_b(x)$ is ∞.

Contrary to the approaches based upon Dijkstra's algorithm the elimination tree query approach does not need a priority queue. This leads to significantly less work per processed vertex. Unfortunately the query must always process all vertices in the search space. Luckily, our experiments show that for random queries with s and t sampled uniformly at random the query time ends up being lower for the

elimination tree query. If s and t are close in the original graph, the Dijkstra-based approaches are faster.

3.1.4.4 *Path Unpacking*

All shortest path queries presented only compute shortest up-down paths. This is enough to determine the distance of a shortest path in the original graph. However, if the sequence of edges that form a shortest path should be computed, then the up-down path must be unpacked. The original CH of [GSSV12] unpacks an up-down path by storing for every arc (x, y) the vertex z of the lower triangle $\{x, y, z\}$ that caused the weight at $m(x, y)$. This information depends on the metric and we want to avoid storing additional metric-dependent information. We therefore resort to a different strategy: Denote by $p_1 \dots p_k$ the up-down path found by the query. As long as a lower triangle $\{p_i, p_{i+1}, x\}$ of an arc (p_i, p_{i+1}) exists with $m(p_i, p_{i+1}) = m(x, p_i) + m(x, p_{i+1})$, our algorithm inserts the vertex x between p_i and p_{i+1} into the path.

3.1.5 *Experiments*

In this section, we present an extensive experimental evaluation of the algorithms introduced and described before.

COMPILER AND MACHINE. We implemented our algorithms in C++, using g++ 4.7.1 with -O3 for compilation. The customization and query experiments were run on a dual 8-core Intel Xeon E5-2670 processor, which is based on the Sandy Bridge architecture, clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. The order computation experiments reported in Table 3.2 were run on a single core of an Intel Core i7-2600K processor.

METRIC-DEPENDENT ORDERS. Most publications on applications and extensions of Contraction Hierarchies use greedy orders in the spirit of [GSSV12], but details of vertex order computation and witness search vary. For reproducibility, we describe our precise approach in this section, extending on the general description of metric-dependent CH preprocessing given in Section 3.1.1.

Our witness search aborts once it finds some path shorter than the shortcut—or when both forward and backward search each have settled at most p vertices. For most experiments we choose $p = 50$. The only exception is the distance metric on road graphs, where we set $p = 1500$. We found that a higher value of p increases the time per witness search but leads to sparser cores. For the distance metric we needed a high value because otherwise our cores get too dense. This effect did not occur for the other weights considered in the experiments. Our weighting heuristic is similar to the one of [ADGW12]. We denote by $L(x)$ a value that estimates the level of

Table 3.1: Benchmark instances. We report the number of vertices and directed arcs, as well as the number of edges in the induced undirected graph. For comparison, we also report the running time of Dijkstra's algorithm (with stop criterion) averaged over 10 000 st-queries, where s and t are chosen uniformly at random.

Instance	# Vertices	# Arcs	# Edges	Symmetric?	Dij. [ms]
Karlsruhe	120 412	302 605	154 869	no	6
TheFrozenSea	754 195	5 815 688	2 907 844	yes	58
Europe	18 010 173	42 188 664	22 211 721	no	1 560

a vertex x. Initially all $L(x)$ are 0. If x is contracted, then for every incident edge $\{x, y\}$ we perform $L(y) \leftarrow \max\{L(y), L(x) + 1\}$. We further store for every arc a a hop length $h(a)$. This is the number of arcs that the shortcut represents if fully unpacked. Denote by $D(x)$ the set of arcs removed if x is contracted and by $A(x)$ the set of arcs that are inserted. Note that $A(x)$ is not necessarily a full clique because of the witness search and because some edges may already exist. We greedily contract a vertex x that minimizes an *importance* $I(x)$ defined by

$$I(x) = L(x) + \frac{|A(x)|}{|D(x)|} + \frac{\sum_{a \in A(x)} h(a)}{\sum_{a \in D(x)} h(a)}.$$

We maintain a priority queue that contains all vertices weighted by I. Initially all vertices are inserted with their exact importance. As long as the queue is not empty, we remove a vertex x with minimum importance $I(x)$ and contract it. This modifies the importance of other vertices. However, our weighting function is chosen such that only the importance of adjacent vertices is influenced, if the witness search was perfect. We therefore only update the importance values of all vertices y in the queue that are adjacent to x. In practice, with a limited witness search, we sometimes choose a vertex x with a sightly suboptimal $I(x)$. However, preliminary experiments have shown that this effect can be safely ignored. Hence, for the metric-dependent CH experiments presented in this section, we do not use lazy updates or periodic queue rebuilding as proposed in [GSSV12].

INSTANCES. We evaluate three large instances of practical relevance in detail. In Section 3.1.5.8 we provide summarized experiments on further instances. The sizes of our main test instances are reported in Table 3.1: The DIMACS Europe graph[3] was provided by PTV[4] for the DIMACS challenge [DGJ09]. The vertex positions are depicted in Figure 3.8. It is the standard benchmarking instance used by many

3 Visit http://i11www.iti.kit.edu/resources/roadgraphs.php for details on how to acquire this graph.

4 http://www.ptvgroup.com

Table 3.2: Orders. Duration of order computation in seconds, without parallelization. KaHIP was parametrized for quality only, disregarding running time (as part of the metric-independent, first phase).

Instance	MetDep [s]	Metis [s]	KaHIP [s]
Karlsruhe	4.1	0.5	< 1 532
TheFrozenSea	1 280.4	4.7	< 22 828
Europe	813.5	131.3	< 249 082

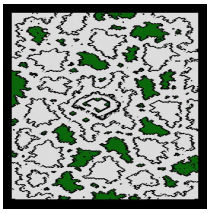
research papers on route planning in road networks. Note that besides roads it also contains a few ferries to connect Great Britain and some other islands with the continent. The Europe graph analyzed here, is its largest strongly connected component, which is a common method to remove bogus vertices. The graph is directed, and we consider two different weights. The first weight is the travel time and the second weight is the straight-line distance between two vertices on a perfect Earth sphere. Note that in the input data highways are often modeled using only a small number of vertices compared to the streets going through cities. This differs from other data sources, such as OpenStreetMap (`http://www.openstreetmap.org`) that have a high number of vertices on highways to model road bends. As demonstrated in Section 3.1.5.8, degree-2 vertices do not hamper the performance of CHs.

The Karlsruhe graph is a subgraph of the PTV graph for a larger region around Karlsruhe. We consider the largest connected component of the graph induced by all vertices with a latitude between 48.3° and 49.2°, and a longitude between 8° and 9°.

The TheFrozenSea graph is based on the largest Star Craft map presented in [Stu12]. The map is composed of square tiles having at most eight neighbors and distinguishes between walkable and non-walkable tiles. These are not distributed uniformly, but rather form differently-sized pockets of freely walkable space alternating with *choke points* of very limited walkable space. The corresponding graph contains for every walkable tile a vertex and for every pair of adjacent walkable tiles an edge. Diagonal edges are weighted by $\sqrt{2}$, while horizontal and vertical edges have weight 1. The graph is symmetric, i. e., for each forward arc there is a backward arc, and contains large grid subgraphs.

For comparability with other works we report in Table 3.1 the time needed by Dijkstra's algorithm. Our implementation uses a 4-ary heap. As usual, it is uni-directional and employs a stopping criterion for point-to-point queries.

(a) DIMACS Europe [DGJ09]



(b) TheFrozenSea [Stu12]

Figure 3.8: Main benchmarks.

3.1.5.1 *Orders*

We analyze three different vertex orders: 1) The greedy metric-dependent order in the spirit of [GSSV12]. We refer to it as "MetDep" in the tables. 2) The Metis 5.0.1 graph partitioning package contains a tool called `ndmetis` to create ND-orders. 3) KaHIP 0.61 provides only graph partitioning. We therefore implemented a very basic nested dissection computation on top of it: For every graph we iteratively compute bisections with different random seeds using the "strong" configuration of KaHIP, until for ten consecutive runs no better cut is found. We recursively bisect the graph until the parts are too small for KaHIP to handle and assign the order arbitrarily in these small parts. We set the imbalance for KaHIP to 20%. Note that our program is solely tuned for quality, completely disregarding running time. This does not imply that the *original* KaHIP package is slow. Table 3.2 reports the times needed to compute the orders. Interestingly, Metis is even faster than the metric-dependent greedy vertex ordering strategy.

RESULTING CH SIZES In Table 3.3, we report the resulting CH sizes for various approaches. Computing a CH on Europe *without witness search* with the greedy, metric-dependent order is infeasible, even using the Contraction Graph data structure. This is also true if we only want to count the number of arcs: We aborted calculations after several days. However, we can state with certainty that there are at least 1.3×10^{12} arcs in the CH, and the maximum upward vertex degree is at least 1.4×10^{6}. As the original graph has only 4.2×10^{7} arcs, it is safe to assume that, using this order, it is impossible to achieve a speedup over Dijkstra's algorithm on the input graph. However, at least on the Karlsruhe graph we can compute the CH without witness search and perform a perfect witness search. The numbers show that the heuristic witness search employed by [GSSV12] is nearly optimal. Furthermore, the numbers clearly show that using metric-dependent orders in a metric-independent setting, i. e., without witness search, results in unpractical CH sizes. However, they also show that a metric-dependent order exploiting the weight structure dominates ND-orders.

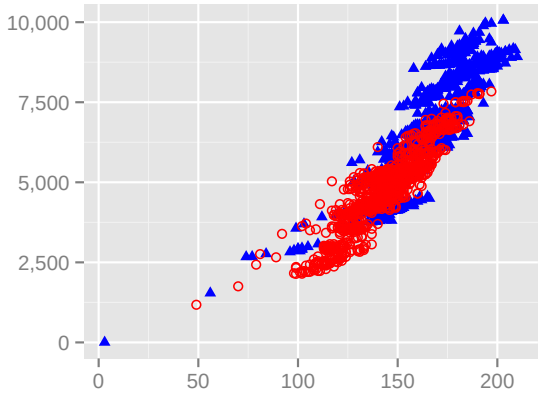
In Figure 3.9 we plot the number of arcs in the search space vs the number of vertices. The plots show that the KaHIP order significantly outperforms the Metis order on the road graphs whereas the situation is a lot less clear on the game map where the plots suggest nearly a tie. KaHIP only slightly outperforms Metis with perfect customization.

Table 3.4 examines the elimination tree. Most noticeably, it has a relatively small height compared to the number of vertices in G. Note that the height of the elimination tree corresponds[5] to the number of vertices in the (undirected) search space.

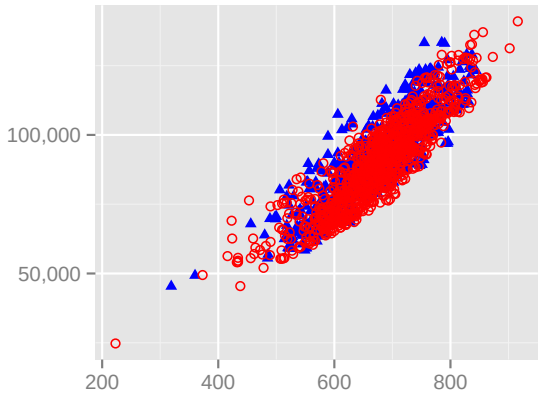
5 The numbers in Table 3.3 and Table 3.4 deviate a little because the search spaces in the former table are sampled while in the latter we compute precise values.

Table 3.3: Size of the Contraction Hierarchies for different instances and orders. We report the average number of vertices and arcs reachable in the upward search space of a vertex. This number varies depending on whether a witness search is performed or not. It also varies depending on whether we follow one-way streets in both direction or not. We also report the number of triangles. As an indication for query performance, we report the average search space size in vertices and arcs, by sampling the search space of 1000 random vertices. Metis and KaHIP orders are metric-independent. We report resulting figures after applying different variants of witness search. A heuristic witness search is one that exploits the metric in the preprocessing phase. A perfect witness search is described in Section 3.1.3.

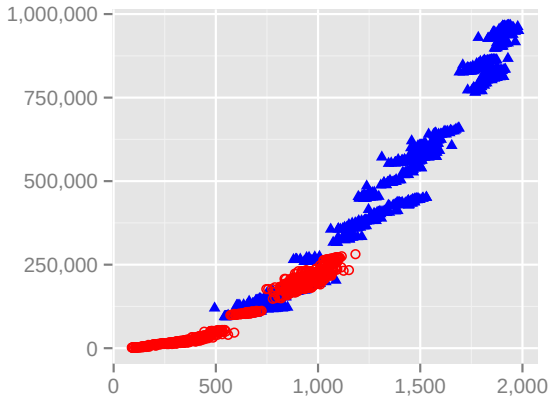
						Average upward search space size			
			# Arcs [$\cdot 10^3$]		# Triangles	unweighted		weighted	
	Order	Witness search	undir.	upward	[$\cdot 10^3$]	# Vertices	# Arcs	# Vertices	# Arcs
Karlsruhe	MetDep	none	21 926	17 661	37 439 858	5 870	15 786 622	5 246	11 281 564
		heuristic	—	244	—	—	—	108	503
		perfect	—	239	—	—	—	107	498
	Metis	none	478	463	2 590	164	6 579	163	6 411
		perfect	—	340	—	—	—	152	2 903
	KaHIP	none	528	511	2 207	143	4 723	142	4 544
		perfect	—	400	—	—	—	136	2 218
TheFrozenSea	MetDep	heuristic	—	6 400	—	—	—	1 281	13 330
	Metis	none	21 067	21 067	601 846	676	92 144	676	92 144
		perfect	—	10 296	—	—	—	644	32 106
	KaHIP	none	25 100	25 100	864 041	674	89 567	674	89 567
		perfect	—	10 162	—	—	—	645	24 782
Europe	MetDep	heuristic	—	33 912	—	—	—	709	4 808
	Metis	none	70 070	65 546	1 409 250	1 291	464 956	1 289	453 366
		perfect	—	47 783	—	—	—	1 182	127 588
	KaHIP	none	73 920	69 040	578 248	652	117 406	651	108 121
		perfect	—	55 657	—	—	—	616	44 677



(a) Karlsruhe



(b) TheFrozenSea



(c) Europe

Figure 3.9: The number of vertices (horizontal) versus the number of arcs (vertical) in the search space of 1000 random vertices. The red hollow circles represent KaHIP, and the blue filled triangles represent Metis.

Table 3.4: Elimination tree characteristics. Note that unlike in Table 3.3, these values are exact and not sampled over a random subset of vertices. We also report upper bounds on the treewidth of the input graphs, after dropping the directions of arcs.

Instance	Order	#Children		Height		Upper bound of Treewidth
		avg.	max.	avg.	max.	
Karlsruhe	Metis	1	5	163.48	211	92
	KaHIP	1	5	142.19	201	72
TheFrozenSea	Metis	1	3	675.61	858	282
	KaHIP	1	3	676.71	949	287
Europe	Metis	1	8	1283.45	2017	876
	KaHIP	1	7	654.07	1232	479

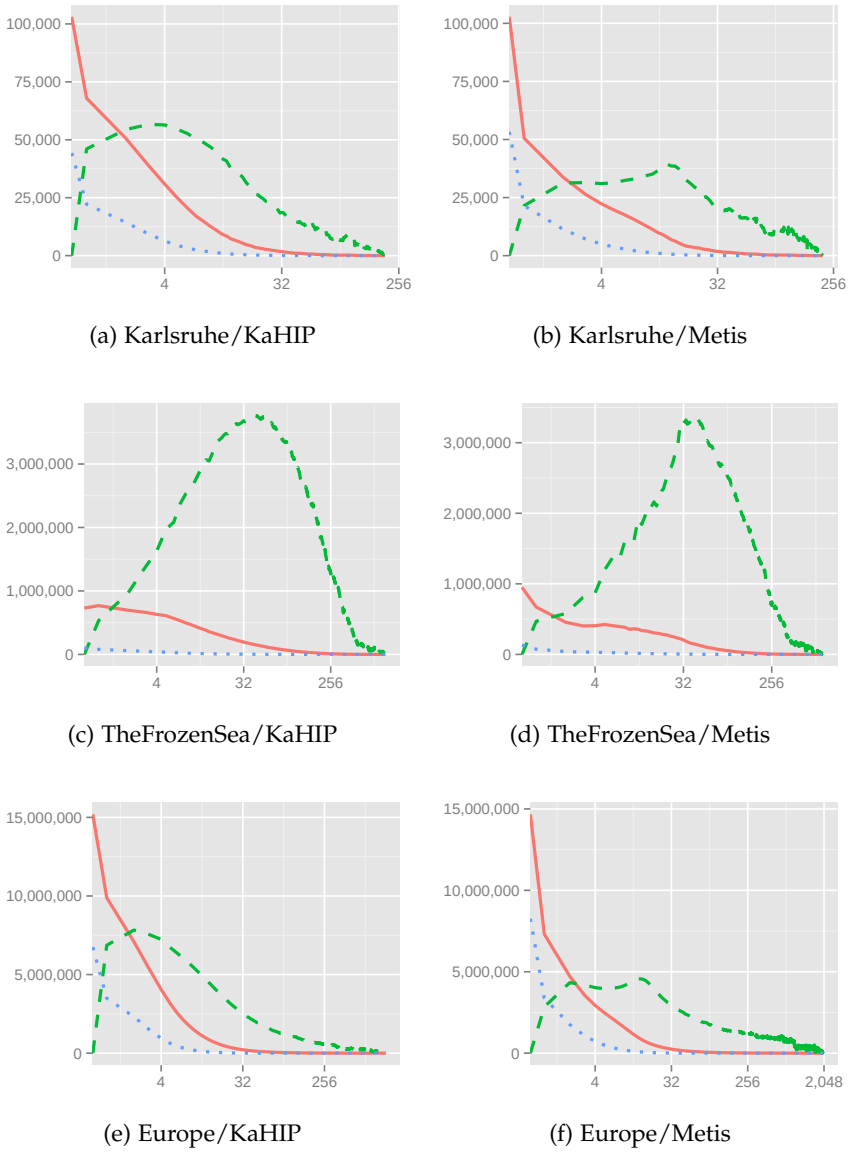
The treewidth of a graph is a measure widely used in theoretical computer science, deeply coupled with the notion of chordal supergraphs and vertex separators. See [BK10] for details. The authors show in their Theorem 6 that the maximum upward degree $d_u(v)$ over all vertices v in $G_\pi^\wedge$ is an upper bound to the treewidth of a graph G. This theorem yields a straightforward algorithm that gives us the upper bounds presented in Table 3.4.

In Table 3.5 we evaluate the witness search performances for different metrics. It turns out that the distance metric is the most difficult one of the tested metrics. That the distance metric is more difficult than the travel time metric is well known. However, it surprised us that uniform and random metrics are easier than the distance metric. We suppose that the random metric contains a few very long arcs that are nearly never used. These could just as well be removed from the graph resulting in a thinner graph with nearly the same shortest path structure. The CH of a thinner graph with a similar shortest path structure naturally has a smaller size. To explain why the uniform metric behaves more similar to the travel time metric than to the distance metric, we have to realize that, on our data source, highways do not have many degree-2 vertices in the input graph. Highways are therefore also preferred by the uniform metric. We expect an instance with more degree-2 vertices on highways to behave differently. Interestingly, the heuristic witness search is perfect for a uniform metric. We expect this effect to disappear on larger graphs.

Recall that a CH is a DAG, and in DAGs each vertex can be assigned a level. If a vertex can be placed in several levels we put it in the lowest level. Figure 3.10 illustrates the amount of vertices and arcs in each level of a CH. The many highly ranked extremely thin levels are a result of the top level separator clique: Inside a clique every vertex

Table 3.5: Detailed analysis of the size of CHs after perfect witness search. We evaluate uniform, random and distance weights on the Karlsruhe input graph. Random weights are sampled from $[0, 10000]$. The distance weight is the straight distance along a perfect Earth sphere's surface. All weights respect one-way streets of the input graph.

Instance	Metric	Order	Witness search	# Upward arcs	Avg. weighted upw. search space	
					# Vertices	# Arcs
Karlsruhe	Distance	MetDep	none	8 000 880	3 276	4 797 224
			heuristic	295 759	283	2 881
			perfect	295 684	281	2 873
		Metis	perfect	382 905	159	3 641
		KaHIP	perfect	441 998	141	2 983
	Uniform	MetDep	none	5 705 168	2 887	3 602 407
			heuristic	272 711	151	808
			perfect	272 711	151	808
		Metis	perfect	363 310	153	2 638
		KaHIP	perfect	426 145	136	2 041
	Random	MetDep	none	6 417 960	3 169	4 257 212
			heuristic	280 024	160	949
			perfect	276 742	160	948
		Metis	perfect	361 964	154	2 800
		KaHIP	perfect	424 999	138	2 093
Europe	Distance	MetDep	heuristic	39 886 688	4 661	133 151
		Metis	perfect	53 505 231	1 257	178 848
		KaHIP	perfect	60 692 639	644	62 014



(a) Karlsruhe/KaHIP

(b) Karlsruhe/Metis

(c) TheFrozenSea/KaHIP

(d) TheFrozenSea/Metis

(e) Europe/KaHIP

(f) Europe/Metis

Figure 3.10: The number of vertices (y-axis) per level (x-axis) is represented by the blue dotted line. The number of arcs departing in each level is represented by the red solid line, and the number of lower triangles in each level is represented by the green dashed line. Warning: In contrast to Figure 3.11, these figures have a logarithmic x-scale.

Table 3.6: Construction of the Contraction Hierarchy. We report the time in seconds required to compute the arcs in $G_\pi^\triangle$ given a KaHIP ND-order π. No witness search is performed. No weights are assigned.

Instance	Dyn. Adj. Array	Contraction Graph
Karlsruhe	0.6	<0.1
TheFrozenSea	490.6	3.8
Europe	305.8	15.5

must be on its own level. A few big separators therefore significantly increase the level count.

3.1.5.2 *CH Construction*

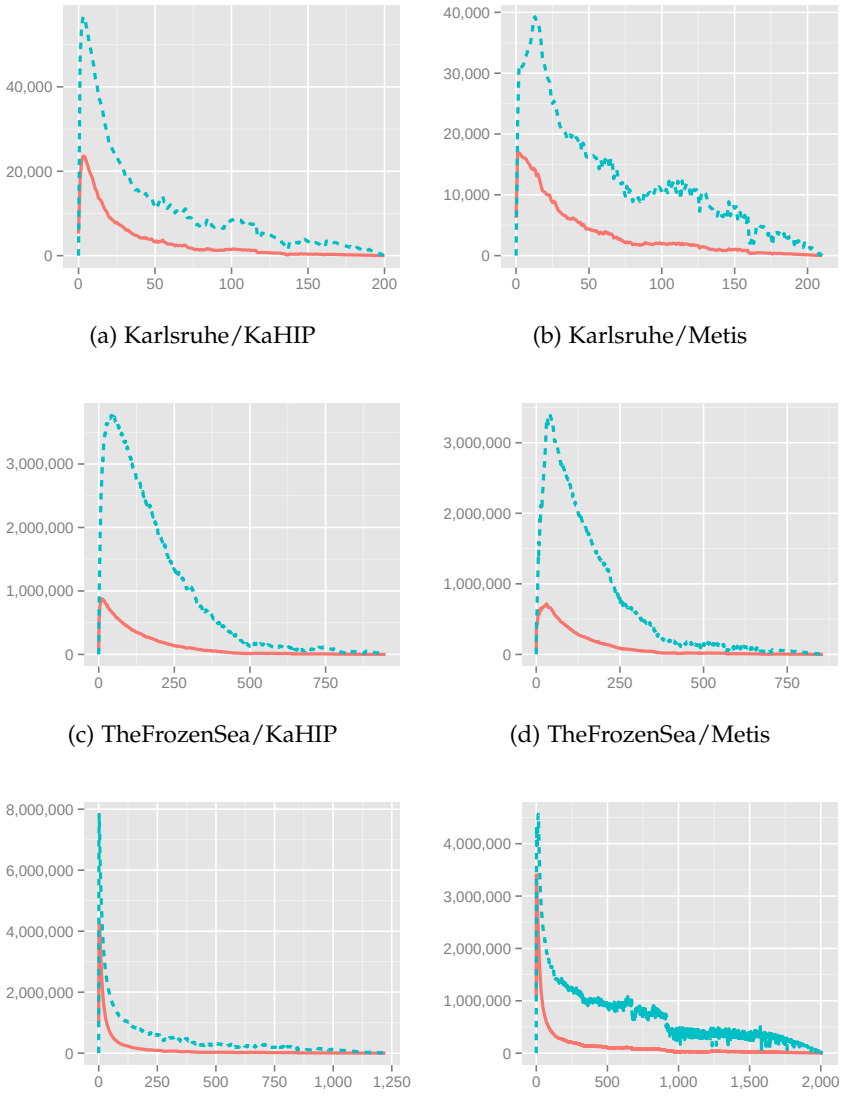
Table 3.6 reports the time required to compute the unweighted CH $G_\pi^\triangle$ from a given KaHIP nested dissection order π. Results show that our specialized Contraction Graph data structure, described in Section 3.1.2.2, dramatically improves performance over the commonly used dynamic adjacency structure, e. g., as in [GSSV12]. However, to be fair, our approach cannot immediately be extended to directed or weighted graphs, without applying our customization scheme. We do not report numbers for the hash-based approach of [Zei13] as it is fully dominated.

3.1.5.3 *Triangle Enumeration*

We first evaluate the running time of the adjacency-array-based triangle enumeration algorithm. Figure 3.11 clearly shows that most time is spent enumerating the triangles of the lower levels. This justifies our suggestion to only precompute the triangles for the lower levels as these are the levels where the optimization is most effective. However, precomputing more levels does not hurt if enough memory is available. We propose to determine the threshold level up to which triangles are precomputed based on the size of the available unoccupied memory. On modern server machines such as our benchmarking machine there is enough memory to precompute all levels. The memory consumption is summarized in Table 3.7. However, note that precomputing all triangles is prohibitive in the game scenario as less available memory should be expected.

3.1.5.4 *Customization*

In Table 3.8 we report the times needed to compute a customized metric using the basic customization algorithm. A first observation is that on the road graphs the KaHIP order leads to a faster customization whereas on the game map Metis dominates. Using all optimizations presented we customize Europe in below one second.



(a) Karlsruhe/KaHIP (b) Karlsruhe/Metis

(c) TheFrozenSea/KaHIP (d) TheFrozenSea/Metis

(e) Europe/KaHIP (f) Europe/Metis

Figure 3.11: The number of lower triangles (y-axis) per level (x-axis) is represented by the blue dashed line, and the time needed to enumerate all of them per level is represented by the red solid line. The time unit is 100 ns. If the time curve thus rises to 1,000,000 on the plot, the algorithm needs 0.1 seconds. Warning: In contrast to Figure 3.10, these figures do not have a logarithmic x-scale.

Table 3.7: Precomputed triangles. As show in Section 3.1.2.3, the memory needed is proportional to $2t + m + 1$, where t is the triangle count and m the number of arcs in the CH. We use 4 byte integers. We report t and m for precomputing all levels ("full") and all levels below a reasonable threshold level ("partial"). We further indicate the percentage of total unaccelerated enumeration time spent below the given threshold level. We chose the threshold level such that this factor is about 33 %.

		Karlsruhe		TheFrozenSea		Europe	
		Metis	KaHIP	Metis	KaHIP	Metis	KaHIP
full	# Triangles [10^3]	2 590	2 207	601 846	864 041	1 409 250	578 247
	# CH arcs [10^3]	478	528	21 067	25 100	70 070	73 920
	Memory [MB]	22	19	4 672	6 688	11 019	4 694
partial	Threshold level	16	11	51	54	42	17
	# Triangles [10^3]	507	512	126 750	172 240	147 620	92 144
	# CH arcs [10^3]	367	393	13 954	15 996	58 259	59 282
	Memory [MB]	5	5	1 020	1 375	1 348	929
	Enum. time [%]	33	32	33	33	32	33

Table 3.8: Basic customization performance. The input graphs are assumed to be directed, i.e., separate upward and downward metrics are used. We show the impact of enabling SSE, precomputing triangles (Pre. trian.), multi-threading (# Thr.), and customizing several metric pairs at once.

				Karlsruhe		TheFrozenSea		Europe	
SSE	Pre. trian.	# Thr.	# Metrics Pairs	Metis time [s]	KaHIP time [s]	Metis time [s]	KaHIP time [s]	Metis time [s]	KaHIP time [s]
no	none	1	1	0.0567	0.0468	7.88	10.08	21.90	10.88
yes	none	1	1	0.0513	0.0427	7.33	9.34	19.91	9.55
yes	all	1	1	0.0094	0.0091	3.74	3.75	7.32	3.22
yes	all	16	1	0.0034	0.0035	0.45	0.61	1.03	0.74
yes	all	16	2	0.0035	0.0033	0.66	0.76	1.34	1.05
yes	all	16	4	0.0040	0.0048	1.19	1.50	2.80	1.66

Table 3.9: Detailed basic customization performance on TheFrozenSea. We show the impact of exploiting undirectedness, customizing several metrics at once, reducing the bitwidth of the metric, enabling SSE, multi-threading (# Thr.), and precomputing triangles (Pre. trian.). Note that the order in which improvements are investigated is different from Table 3.8. Also note that results are based on the Metis order as Table 3.8 shows that KaHIP is outperformed.

Undir.	# Metrics	Metric bits	SSE	# Threads	Precomputed triangles	Customization time [s]	Amortized time [s]
no	2 (up & down)	32	no	1	none	7.88	7.88
yes	1	32	no	1	none	6.65	6.65
yes	4	32	no	1	none	9.36	2.34
yes	4	32	yes	1	none	8.51	2.13
yes	8	16	yes	1	none	8.52	1.06
yes	8	16	yes	2	none	5.00	0.63
yes	8	16	yes	2	all	2.16	0.27
yes	8	16	yes	16	all	0.63	0.08

When amortized[6], we even achieve 415 ms which is only slightly above the non-amortized 347 ms reported in [DW13] for CRP. Note that their experiments were run on a different machine with a faster clock but 2×6 instead of 2×8 cores, while using a turn-aware data structure, making an exact comparison difficult.

Previous works have tried to accelerate the preprocessing phase of the original two-phase CH to the point that it can be used in a similar scenario as our technique. A fast preprocessing phase can be viewed as form of customization phase. In [GSSV12] a sequential preprocessing time of 451 s was reported. This compares best to our 9.5 s sequential customization time. Note that the machine on which the 451 s were measured is slower than our machine. However, the gap in performance is large enough to conclude that we achieve a significant speedup. Furthermore, [ADGW12] report a CH preprocessing time of 2 min when parallelized on 12 cores. This compares best against our 415 ms parallelized amortized customization time. While the machine used in [ADGW12] is slightly older and slower than our machine and the number of cores differs (12 vs. 16), again, the performance gap is large enough to safely conclude that a significant speedup is present. Besides these differences, both CH preprocessing experiments were only performed for travel time weights. To the best of our knowledge, nobody has been able to match performance achieved for travel time weights for less well-behaved weights, such as travel distance. For example, CH preprocessing times reported in [GSSV12] show at least

6 We refer to a server scenario of multiple active users that require simultaneous customization, e. g., due to traffic updates.

Table 3.10: Partial update performance. We report time required in milliseconds and number of arcs changed for partial metric updates. We report median, average and maximum over 10,000 runs. In each run we change the upward and the downward weight of a single random arc in G to random values in $[0, 10^5]$. The metric is reset to initial state between runs. Timings are sequential without SSE. No triangles were precomputed.

		Arcs removed from queue			Partial update [ms]		
		med.	avg.	max.	med.	avg.	max.
Karlsruhe	Metis	2	3.5	857	0.001	0.003	0.9
	KaHIP	3	3.7	466	0.001	0.002	1.0
TheFrozenSea	Metis	6	311.7	14 494	0.008	1.412	100.2
	KaHIP	6	343.1	19 417	0.008	1.490	164.6
Europe	Metis	2	10.2	14 188	0.005	0.052	134.6
	KaHIP	3	9.8	8 202	0.008	0.045	81.0

a factor of 2 difference in performance on distance over travel time metric on any of the considered benchmarks. This contrasts with CCH, for which basic customization and elimination-tree query performance are provably independent of the metric considered.

Unfortunately, the optimizations illustrated in Table 3.8 are pretty far from what is possible with the hardware normally available in a game scenario. Regular PCs do not have 16 cores and one cannot clutter up the whole RAM with several GB of precomputed triangles. We therefore ran additional experiments with different parameters and report the results in Table 3.9. The experiments show that it is possible to fully customize TheFrozenSea in an amortized[7] time of 1.06s without precomputing triangles or using multiple cores. However a whole second is still too slow to be usable, as graphics, network and game logic also require resources.

We therefore evaluated the time needed by partial updates as described in Section 3.1.3.7. We report our results in Table 3.10, also for the road networks. The median, average and maximum running times significantly differ. There are a few arcs that trigger a lot of subsequent changes, whereas for most arcs a weight change has nearly no effect. The explanation is that highway arcs and choke point arcs are part of many shortest paths, and thus updating such an arc triggers significantly more changes. On the Europe road network, the maximum observed time for a partial CCH update (81.0 ms) is similiar to CRP (73.77 ms), but the average time is much lower for CCH (0.045 ms) than for CRP (17.94 ms), cf. [DGPW15].

7 We refer to a multiplayer scenario, where, e. g., fog of war requires player-specific simultaneous customization.

Table 3.11: Perfect Customization. We report the time required to turn an initial metric into a perfect metric. Runtime is given in seconds, without use of SSE.

		Karlsruhe		TheFrozenSea		Europe	
# Thr.	Pre. trian.	Metis	KaHIP	Metis	KaHIP	Metis	KaHIP
1	none	0.15	0.13	30.54	33.76	67.01	32.96
16	none	0.03	0.02	3.26	4.37	14.41	5.47
1	all	0.05	0.05	8.95	12.51	23.93	10.75
16	all	0.01	0.01	1.93	2.29	3.50	2.35

Finally, we report the running times of the perfect customization algorithm in Table 3.11. The required running time is about three times the running time needed by the basic customization. Recall that the basic customization enumerates all lower triangles, i. e., visits every triangle once, whereas the perfect customization also enumerates all intermediate and upper triangles, i. e., visits every triangle three times.

3.1.5.5 *Query Performance*

We experimentally evaluated the running times of the query algorithms. For this we ran 10^6 shortest path distance queries with the source and target vertices picked uniformly at random. The presented times are averaged running times on a single core without SSE.

In Table 3.12– 3.14 we compare query performance for our different algorithmic variants. The "MetDep+w" variant uses a metric-dependent order and a non-perfect witness search as in [GSSV12]. The "Metis-w" and "KaHIP-w" variants use a metric-independent order computed by Metis or KaHIP. Only a basic customization was performed, i. e., no witness search was performed. The "Metis+w" and "KaHIP+w" variants use the same metric-independent order but a perfect customization followed by a perfect witness search was performed. We evaluate three query variants. The "Basic" variant uses a bidirectional variant of Dijkstra's algorithm with stopping criterion. The "Stalling" variant additionally uses the stall-on-demand optimization as described in Section 3.1.4.2. Finally, we also evaluate the elimination tree query and refer to it as "Tree". This query requires the existence of an elimination tree of low depth and is therefore not available for metric-dependent orders. We ran our experiments on all three of our main benchmark instances. Experiments on additional instances are available in Section 3.1.5.8. For both road graphs, we evaluate the travel-time and distance variants. We report the average running time needed to perform a distance query, i. e., we do not unpack the paths. We further report the average number of "visited" vertices in the forward search. For the "basic" and "stalling" queries,

Table 3.12: Query performance in *microseconds* as well as the search space visited, averaged over 10^6 queries with source and target vertices picked uniformly at random. We use "visited" to differentiate from the maximum reachable search space given in Table 3.3. If applicable, we additionally report the number of vertices stalled, as well as the number of arcs touched during the stalling test. Note that stalled vertices are not counted as visited. All reported vertex and arc counts only refer to the forward search. We evaluate several algorithmic variants. Each variant is composed of an input graph, a contraction order, and whether a witness search is used. "+w" means that a witness search is used, whereas "-w" means that no witness search is used. "MetDep+w" corresponds to the original CHs. The metrics used for "-w" are directed and maximum. Results for Karlsruhe. Table 3.13 and 3.14 contain the results for FrozenSea and Europe.

				Visited up. search space		Stalling		Time
Instance	Metric	Variant	Algorithm	# Vertices	# Arcs	# Vertices	# Arcs	[µs]
Karlsruhe	Travel-Time	MetDep+w	Basic	81	370	—	—	17
			Stalling	43	182	167	227	16
		Metis-w	Basic	138	5 594	—	—	62
			Stalling	104	4 027	32	4 278	67
			Tree	164	6 579	—	—	33
		KaHIP-w	Basic	120	4 024	—	—	48
			Stalling	93	3 051	26	3 244	55
			Tree	143	4 723	—	—	25
		Metis+w	Basic	127	2 432	—	—	32
			Stalling	104	2 043	19	2 146	41
			Tree	164	2 882	—	—	17
		KaHIP+w	Basic	114	1 919	—	—	27
			Stalling	93	1 611	18	1 691	35
			Tree	143	2 198	—	—	14
	Distance	MetDep+w	Basic	208	1978	—	—	57
			Stalling	70	559	46	759	35
		Metis-w	Basic	142	5 725	—	—	65
			Stalling	115	4 594	26	4 804	75
			Tree	164	6 579	—	—	33
		KaHIP-w	Basic	123	4 117	—	—	50
			Stalling	106	3 480	17	3 564	59
			Tree	143	4 723	—	—	26
		Metis+w	Basic	138	3 221	—	—	39
			Stalling	115	2 757	21	2 867	50
			Tree	164	3 604	—	—	21
		KaHIP+w	Basic	122	2 626	—	—	32
			Stalling	106	2 302	14	2 350	43
			Tree	143	2 956	—	—	17

Table 3.13: Query performance. Continuation of Table 3.12.

				Visited up. search space		Stalling		Time
Instance	Metric	Variant	Algorithm	# Vertices	# Arcs	# Vertices	# Arcs	[µs]
TheFrozenSea	Map-Distance	MetDep+w	Basic	1 199	12 692	—	—	539
			Stalling	319	3 460	197	4 345	286
		Metis-w	Basic	610	81 909	—	—	608
			Stalling	578	78 655	24	79 166	837
			Tree	676	92 144	—	—	317
		KaHIP-w	Basic	603	82 824	—	—	644
			Stalling	560	74 244	50	74 895	774
			Tree	674	89 567	—	—	316
		Metis+w	Basic	567	28 746	—	—	243
			Stalling	474	25 041	86	25 445	333
			Tree	676	31 883	—	—	120
		KaHIP+w	Basic	578	22 803	—	—	203
			Stalling	475	19 978	81	20 138	276
			Tree	674	24 670	—	—	106

these are the vertices removed from the queue. For the "tree" query, we regard every ancestor as "visited". The numbers for the backward search are analogous and therefore not reported. We also report the average number of arcs relaxed in forward search of each query variant. Finally, we also report the average number of vertices stalled and the average number of arcs that need to be tested in the stalling test. Note that a stalled vertex is not counted as "visited".

An important detail necessary to reproduce these results consists of reordering the vertex IDs according to the contraction order. Preliminary experiments showed that this reordering results in better cache behavior and a speed-up of about 2 to 3 because much query time is spent on the topmost clique and this order ensures that these vertices appear adjacent in memory.

As observed in [GSSD08; GSSV12], we confirm that the stall-on-demand heuristic improves running times by a factor of 2–5 compared to the basic algorithm for "greedy+w". Interestingly, this is not the case with any variant using a metric-independent order. This can be explained by the density of the search spaces. While, the number of vertices in the search spaces are comparable between metric-independent orders and metric-dependent order, the number of arcs are not comparable and thus metric-independent search spaces are denser. As consequence, we need to test significantly more arcs in the stalling-test, which makes the test more expensive and therefore the additional time spent in the test does not make up for the time economized in

Table 3.14: Query performance. Continuation of Table 3.12.

				Visited up. search space		Stalling		Time
Instance	Metric	Variant	Algorithm	# Vertices	# Arcs	# Vertices	# Arcs	[µs]
Europe	Travel-Time	MetDep+w	Basic	546	3 623	—	—	283
			Stalling	113	668	75	911	107
		Metis-w	Basic	1 126	405 367	—	—	2 838
			Stalling	719	241 820	398	268 499	2 602
			Tree	1 291	464 956	—	—	1 496
		KaHIP-w	Basic	581	107 297	—	—	810
			Stalling	418	75 694	152	77 871	857
			Tree	652	117 406	—	—	413
		Metis+w	Basic	1 026	110 590	—	—	731
			Stalling	716	83 047	271	89 444	951
			Tree	1 291	126 403	—	—	398
		KaHIP+w	Basic	549	41 410	—	—	305
			Stalling	418	33 078	117	34 614	425
			Tree	652	45 587	—	—	161
	Distance	MetDep+w	Basic	3 653	104 548	—	—	2 662
			Stalling	286	7 124	426	11 500	540
		Metis-w	Basic	1 128	410 985	—	—	3 087
			Stalling	831	291 545	293	308 632	3 128
			Tree	1 291	464 956	—	—	1 520
		KaHIP-w	Basic	584	108 039	—	—	867
			Stalling	468	85 422	113	87 315	1 000
			Tree	652	117 406	—	—	426
		Metis+w	Basic	1 085	157 400	—	—	1 075
			Stalling	823	124 472	247	127 523	1 400
			Tree	1 291	177 513	—	—	557
		KaHIP+w	Basic	575	56 386	—	—	425
			Stalling	467	46 657	101	47 920	578
			Tree	652	61 714	—	—	214

the actual search. We thus conclude that stall-on-demand is not useful, when using metric-independent orders.

Very interesting is the comparison between the elimination tree query and the basic query. The elimination tree query always explores the whole search space. In contrast to the basic query, it does not have a stopping criterion. However, the elimination tree query does not require a priority queue. It performs thus less work per vertex and arc than the basic query. Our experiments show, that the basic query always explores large parts of the search space regardless of the stopping criterion. The elimination tree query therefore does not visit significantly more vertices. A consequence of this effect is that the time spent in the priority queue outweighs the additional time necessary to explore the remainder of the search space. The elimination tree query is therefore always the fastest among the three query types when using metric-independent orders. Combining a perfect witness search with the elimination tree query results in the fastest queries for metric-independent orders. However, the perfect witness search results in three times higher customization times. Whether it is superior therefore depends on the specific application and the specific trade-off between customization and query running time needed.

The orders computed by KaHIP are nearly always significantly better than those produced by Metis. However, significantly more running time must be invested in the preprocessing phase to obtain these better order. It therefore depends on the situation which order is better. If the running time of the preprocessing phase is relevant, then Metis seems to strike a very good balance between all criteria. However, if the graph topology is fixed, as we expect it to be, then the flexibility gained by using Metis is not worth the price. Interestingly, on the game map KaHIP and Metis seem to be very close in terms of search space size. The difference is only apparent when using the perfect customization. For a setup with basic customization, the two orders are nearly indistinguishable.

On travel-time, the metric-dependent orders outperform the metric-independent orders. However, it is very interesting how close the query times actually are. On the Europe graph, the basic query visits about the same number of vertices, regardless of whether a metric-dependent or the KaHIP order is used. The real difference lies in the number of arcs that need to be relaxed. This number is significantly higher with metric-independent orders. However, the effect this has on the actual running times is comparatively slim. Using KaHIP without perfect witness search results in an elimination tree query that is only about 4 times slower than using the stalling query combined with metric-dependent orders. If a perfect witness search is used then, the gap is below a factor of 2. Further, the metric-dependent orders only win because of the stall-on-demand optimization. The KaHIP

order combined with perfect customization *outperforms* the basic query combined with metric-dependent orders.

It is well-known that metric-dependent CHs work significantly better with the travel-time metric than with other less well behaved metrics such as the geographic distance. For such metrics, the KaHIP order outperforms the metric-dependent orders. For example the basic query with perfect customization visits less vertices and less arcs. This is very surprising, especially considering, that the metric-dependent orders that we computed are better than those reported in [GSSV12], i. e., the gap with respect to the original implementation is even larger. However, combining the stalling query with metric-dependent orders yields the smallest number of visited vertices and relaxed arcs. Unfortunately, combining the stalling query with metric-independent orders does not yield the same benefit and even makes the query running times worse. Fortunately, the metric-independent orders can be combined with the elimination tree query. As result, the fastest variant is the combination of KaHIP order, perfect witness search, and elimination tree query, which is over a factor of two faster than stalling with the metric-dependent order. Interestingly, the latter is even beaten when no perfect witness search is performed, but with a significantly lower margin.

A huge advantage of metric-independent orders compared to metric-dependent orders is that the resulting CH performs equally well regardless of the weights of the input graph. The combination of metric-independent order, elimination tree query and basic customization results in a setup where the order in which the vertices are visited and the order in which the arcs are relaxed during the query execution does not even depend on the weights of the input graph. It is thus impossible to construct a metric, where this setup performs badly. This contrasts with the CH of [GSSV12], whose performances varies significantly depending on the input metric.

In Table 3.15, we give a more in-depth experimental analysis of the elimination tree query algorithm without perfect witness search. We break the running times down into the time needed to compute the least common ancestor (LCA), the time needed to reset the tentative distances and the time needed to relax all arcs. We further report the total distance query time, which is in essence the sum of the former three. We additionally report the time needed to unpack the full path. Our experiments show that the arc-relaxation phase clearly dominates the running times. It is therefore not useful to further optimize the LCA computation or to accelerate tentative distance resetting using, e. g., timestamps. We only report path unpacking performance without precomputed lower triangles. Using them would result in a further speedup with a similar speed-memory trade-off as already discussed for customization.

Table 3.15: Detailed elimination tree performance without perfect witness search. We report running time *in microseconds* for the elimination-tree-based query algorithms. We report the time needed to compute the LCA, the time needed to reset the tentative distances, the time needed to relax the arcs, the total time of a distance query, and the time needed for full path unpacking as well as the average number of vertices on such a path which is metric-dependent.

			Distance query				Path	
			LCA [µs]	Reset [µs]	Arc relax [µs]	Total [µs]	Unpack [µs]	Length [vert.]
Karlsruhe	Travel-Time	Metis	0.6	0.8	31.3	33.0	20.5	189.6
		KaHIP	0.6	1.4	23.1	25.2	18.6	
	Distance	Metis	0.6	0.8	31.5	33.2	27.4	249.4
		KaHIP	0.6	1.4	23.5	25.7	24.7	
TheFrozenSea	Map-Distance	Metis	2.7	3.1	310.1	316.5	220.0	596.3
		KaHIP	3.0	3.2	308.7	315.5	270.8	
Europe	Travel-Time	Metis	4.6	19.0	1471.2	1496.3	323.9	1390.6
		KaHIP	3.4	9.9	399.4	413.3	252.7	
	Distance	Metis	4.7	19.0	1494.5	1519.9	608.8	3111.0
		KaHIP	3.6	10.0	411.6	425.8	524.1	

3.1.5.6 *Comparison with Related Work*

We conclude our experimental analysis on the DIMACS Europe road network with a final comparison of related techniques, as shown in Table 3.16. For Contraction Hierarchies (CH), we report results based on implementations by [DGPW15; GSSV12] and ourselves, covering different trade-offs in terms of preprocessing versus query speed. More precisely, we observe that our own CH implementation (used for detailed analysis and comparison in Section 3.1.5.1–3.1.5.5) has slightly slower queries on travel time metric but factor of 2.1 faster queries on distance metric, at the cost of higher preprocessing time. Recall that we employ a different vertex priority function and no lazy updates. For Customizable Route Planning (CRP), we report results from [DGPW11; DGPW15].

Traditional, metric-dependent CH offers the fastest query time (91 µs, on our machine), but it incurs substantial metric-dependent preprocessing costs, even when parallelized (109 s, 12 cores). Furthermore, CH performance is very sensible regarding metrics used: For distance metric, preprocessing time increases by factor of 3.2–11.5 and query time by factor of 4.9–12.8.

Table 3.16: Comparison with related work on the DIMACS Europe instance with travel time and distance metric. We compare our approaches, CCH, CCH with amortized customization (CCH+a), and CCH with perfect witness search (CCH+w), with different CRP and CH implementations from the literature. We report performance of the metric-*dependent* fraction of overall preprocessing, i. e., vertex ordering and contraction time for CH, customization time for CRP and CCH. We further report average query search space, including stalled vertices for CH (which might not be included in the CH figures taken from [DGPW15]). We finally report running time in microseconds. If parallelized, the number of threads used is noted in parenthesis. Since the CH performance in [GSSV12] was evaluated on a ten year old machine (AMD Opteron 270), we obtained the source code and re-ran experiments on our hardware (Intel Xeon E5-2670) for better comparability. Also note that the latest CRP implementation by [DGPW15], evaluated on an Intel Xeon X5680, is turn-aware (•), i. e., it uses turn tables (set to zero in the reported experiments); We therefore additionally take results from [DGPW11] obtained on an Intel Core-i7 920, which uses a turn-unaware implementation but parallelizes queries.

					Metric-Dep. Prepro.	Queries	
Algorithm	Impl.	Machine	Metric	Turn-aware	Time [s] (# Threads)	Search Space [# Vertices]	Time [µs] (# Threads)
CH	[GSSV12]	Opt 270	Time	○	1 809 (1)	356	152 (1)
CH	[GSSV12]	Opt 270	Dist	○	5 723 (1)	1 582	1 940 (1)
CH	[GSSV12]	E5-2670	Time	○	1 075.88 (1)	353	91 (1)
CH	[GSSV12]	E5-2670	Dist	○	3 547.44 (1)	1 714	1 135 (1)
CH	our	E5-2670	Time	○	813.53 (1)	375	107 (1)
CH	our	E5-2670	Dist	○	9 390.32 (1)	1422	540 (1)
CH	[DGPW15]	X5680	Time	○	109 (12)	280	110 (1)
CH	[DGPW15]	X5680	Dist	○	726 (12)	858	870 (1)
CRP	[DGPW15]	X5680	Time	•	0.37 (12)	2 766	1 650 (1)
CRP	[DGPW15]	X5680	Dist	•	0.37 (12)	2 942	1 910 (1)
CRP	[DGPW11]	i7 920	Time	○	4.7 (4)	3 828	720 (2)
CRP	[DGPW11]	i7 920	Dist	○	4.7 (4)	4 033	790 (2)
CCH	our	E5-2670	Time	○	0.74 (16)	1 303	413 (1)
CCH	our	E5-2670	Dist	○	0.74 (16)	1 303	426 (1)
CCH+a	our	E5-2670	Time	○	0.42 (16)	1 303	416 (1)
CCH+a	our	E5-2670	Dist	○	0.42 (16)	1 303	421 (1)
CCH+w	our	E5-2670	Time	○	2.35 (16)	1 303	161 (1)
CCH+w	our	E5-2670	Dist	○	2.35 (16)	1 303	214 (1)

In contrast to traditional CH, Customizable Contraction Hierarchies (CCH) by design achieve a performance trade-off with much lower metric-dependent preprocessing costs, similar to CRP. Accounting for differences in hardware, CCH basic customization time is about a factor of 2–3 slower than CRP customization, but still well below a second. On the other hand, CCH query performance is factor of 2–4 faster than CRP, both in terms of search space as well as query time (even when accounting for differences due to turn-aware implementation and hardware used). Overall, CCH is more robust wrt. the metric than CRP: By design, CCH customization processes the same sequence of lower triangles for any metric, while the CCH elimination-tree query (given a fixed source and target) processes the same sequence of vertices and arcs for any metric—unless, of course, we employ perfect witness search (CCH+w), see below.

The CRP implementation of [DGPW15] uses SSE to achieve its customization time of 0.37 s. In a server scenario where customization is run for many users concurrently, e. g., to customize a stream of traffic updates for all active users at once, we propose to amortize triangle enumeration time as described in Section 3.1.3.6. By using SSE and processing metrics for four users at once, this amortized customization (CCH+a, 0.42 s) can almost close the gap to CRP customization performance. Refer to Table 3.8 for other configurations.

Most interestingly, in terms of query performance on travel distance, CCH outperforms even the best CH result. For even better CCH query performance, we may employ perfect customization and witness search (CCH+w). It increases customization time by factor of 3.2 (enumerating all lower, intermediate and upper triangles), but enables a CCH query variant that, while still visiting all vertices in the elimination tree, needs to consider far fewer arcs (cf. Table 3.14). Thereby, CCH+w further improves CCH query performance by factor of 1.9 for distance metric and factor of 2.6 for time metric. With 161 µs for travel time, CCH+w query performance is almost as good as the best CH result of 91 µs.

3.1.5.7 *Further Metric-Independent Ordering Strategies*

So far, we have only discussed metric-independent orders based on nested dissection. For completeness, we consider two other metric-independent orders in the following.

In the context of sparse matrix factorization a common approach is the *minimum degree heuristic* (MinDeg). To the best of our knowledge, the first variant of this ordering heuristic was described in [TW67], but we refer to [GL89] for more details. The basic idea is simple: Iteratively contract a vertex with a minimum degree in the core. Note the difference to sorting vertices by degree in the input graph, which does not work well on road networks [DGPW14].

Table 3.17: Time in seconds to compute minimum degree (MinDeg) and minimum shortcut (MinArc) orders.

	Karlsruhe	TheFrozenSea	Europe
MinDeg	1.7	67	250
MinArc	2.1	6 907	30 220

Table 3.18: Further ordering strategies: minimum degree (MinDeg) and minimum shortcut (MinArc).

			Upper		Search Space			
		# Triangles	treewidth	# Arcs in	#Vertices		#Arcs [$\cdot 10^3$]	
Graph	Order	[$\cdot 10^6$]	bound	CH [$\cdot 10^3$]	Avg.	Max.	Avg.	Max.
Karlsruhe	MinDeg	2.37	94	423	244.6	369	11.9	16.2
	MinArc	1.63	75	393	222.4	386	9.2	16.0
	Metis	2.59	92	478	163.5	211	6.5	10.0
	KaHIP	2.21	72	528	142.2	201	4.7	7.9
TheFrozenSea	MinDeg	1 123	500	25 698	1 462.1	2 351	351	502
	MinArc	769	303	22 554	1 192.1	2 034	200	336
	Metis	601	282	21 067	675.6	858	92	135
	KaHIP	864	287	25 100	676.7	949	90	146
Europe	MinDeg	1 800	938	64 313	2 348.0	3 719	1 052	1 494
	MinArc	767	599	56 948	1 815.4	3 256	552	889
	Metis	1 409	876	70 070	1 283.5	2 017	462	967
	KaHIP	578	479	73 920	638.6	1 224	114	284

A variant of this heuristic was proposed in [DGPW15]. The idea is also simple: Iteratively contract a vertex that adds the least number of arcs to the chordal supergraph (MinArc), using degree in the core to break ties. However, [DGPW15] already observed that MinArc orders can be improved when augmented with partitioning information from what they refer to as guidance levels. Their reported experimental results are only with respect to these hybrid orders.

We implemented both MinDeg and MinArc in the straightforward way using a priority queue of vertices ordered by the respective weighting function. Table 3.17 shows the resulting order computation times on our three main instances. Note, that at least for MinDeg more sophisticated strategies exist [GL89] that might be faster. Nonetheless, MinArc is significantly slower than MinDeg because it simulates the contraction of every vertex in the graph, including those that yield a high number of shortcuts but are only contracted in the end, once their degree has already decreased due to the graph shrinking.

More importantly, Table 3.18[8] reports performance indicators for both MinDeg and MinArc in comparison to Metis and KaHIP. Recall that the number of triangles determines customization running times, the number of arcs in the CH is proportional to the memory consumption if precomputed triangles are not used, and the number of arcs in the search space gives a good indication of query performance. MinArc nearly always dominates MinDeg with respect to every criterion except computation time. Yet, while both can be computed faster than the KaHIP-based orders, Metis is still fastest. Similarly, both MinArc and MinDeg result in lower memory consumption than KaHIP-based orders, but not than Metis on the TheFrozenSea instance. Upper treewidth bound from KaHIP-based orders are consistently better than from MinDeg or MinArc.

Note, however, that, at least for our work, customization and query performance are of most importance. In both aspects, MinDeg and MinArc are clearly dominated by Metis on the large game map and by KaHIP-based order on the large road network. Therefore, we did not further consider MinDeg and MinArc orderings in our experiments.

3.1.5.8 *Further Instances*

OPENSTREETMAP-BASED ROAD GRAPHS. OpenStreetMap (OSM) is a very popular collaborative effort to create a map of the world. From this huge data source very large road graphs can be extracted, that are very detailed depending on the exact region considered. Using the data provided by GeoFabrik[9] and the tools provided by OSRM[10], we extracted a road graph of Europe and report its size in Table 3.19. The exact graph is available in DIMACS format on our website[11]. The geographic region corresponding to the graph is depicted in Figure 3.12. Note that compared to the DIMACS Europe, our OSM Europe graph also contains Eastern Europe and Turkey. The graph's east border ends at the east border of Turkey and then goes upward cutting through Russia. On the other hand, the DIMACS Europe graph stops at the German-Poland border.

Figure 3.12: Europe instance from DIMACS and OSM.

At first glance the DIMACS Europe graph looks drastically smaller, at least in terms of vertex count. However, this is very misleading. A peculiarity of OSM is that the road graphs have a huge number of degree-2 vertices. These vertices are used to encode the curvature of a road. This information is needed to correctly represent a road graph on a map but not necessarily for routing. However, most other data sources, including the one on which the DIMACS graph is based, encode this information as arc attributes and thus have fewer degree-2

8 The KaHIP and Metis numbers slightly differ from those in Table 3.3, where they were only sampled over 1000 random search spaces.

9 http://download.geofabrik.de/

10 http://project-osrm.org/

11 http://i11www.iti.kit.edu/resources/roadgraphs.php

Table 3.19: Size of the DIMACS Europe instance compared to the OSM Europe instance.

Instance	# Vertices	# Arcs	# Deg. 1 vertices	# Deg. 2 vertices	# Deg. >2 vertices
DIMACS-Eur	18 M	42 M	4 M (22 %)	2 M (11 %)	11 M (61 %)
OSM-Eur	174 M	348 M	8 M (5 %)	143 M (82 %)	23 M (13 %)

Table 3.20: CH sizes for OSM Europe. The search space sizes were obtained by sampling 10 000 vertices uniformly at random.

	Order	Vertices	Arcs
Input Graph Size	—	174 M	348 M
Search Graph Size	Metis	174 M	400 M
	KaHIP	174 M	434 M
Avg. Search Space	Metis	1 312	495 930
	KaHIP	678	119 295

vertices. Accelerating shortest path computations on graphs with a huge number of vertices of degree 1 or 2 is significantly easier relative to the graph size. One reason is that Dijkstra's algorithm cannot exploit the abundance of low-degree vertices. Dijkstra's algorithm with stopping criterion needs on average 27 s for a st-query with s and t picked uniformly at random on the OSM-Europe graph. This contrasts with the DIMACS Europe graph, where only 1.6 s are needed. A slower baseline obviously leads to larger speedups. Table 3.19 shows that the difference between the two Europe graphs in terms of vertex count is significantly smaller, when discarding degree 1 and degree-2 vertices. In fact, relative to their geographical region's area, the two graphs seem to be approximately comparable in size.

We computed contraction orders for OSM-Europe. The sizes of the resulting CHs are reported in Table 3.20. These sizes can be compared with the "undirected" numbers of Table 3.3. We did not perform experiments with a perfect witness search. Metis ordered the vertices within 29 minutes, whereas the KaHIP-based ordering algorithm needed slightly less than 3 weeks. However, as already discussed in detail, we did not optimize the latter for speed and therefore one must *not* conclude from this experiment that KaHIP is slow. The CHs for OSM-based graphs are significantly larger. The DIMACS-Europe CH only contains 70M arcs for Metis whereas the OSM-Europe CH contains 400M arcs for Metis. However, this is due to the huge amount of low-degree vertices in the input. On the DIMACS graph the size increase compared to the number of input arcs is $70\,\text{M}/42\,\text{M} = 1.67$ whereas for the OSM-based graph the size increase is only $400\,\text{M}/348\,\text{M} = 1.15$.

Table 3.21: Customization performance on OSM Europe. We vary the number of threads and whether precomputed triangles are used. SSE is enabled, running times are non-amortized and no perfect customization was performed.

	#Thr.	Triangle space [GB]	Customization time [s]
Metis	1	—	43.1
	16	—	5.3
	1	16.0	17.3
	16	16.0	2.1
KaHIP	1	—	30.6
	16	—	3.4
	1	7.2	11.4
	16	7.2	1.7

Table 3.22: Query performance on OSM Europe, averaged over 10 000 random st-pairs chosen uniformly at random.

		Query time [ms]
Dijkstra-based	Metis	3.7
	KaHIP	1.0
Elimination-Tree	Metis	1.7
	KaHIP	0.5

This effect can be explained by considering what happens when contracting a graph consisting of a single path. In the input graph every vertex, except the endpoints, has 2 outgoing arcs, one in each direction. As long as the endpoints are contracted last, every vertex, except the endpoints, in the resulting CH search graph also have degree 2. There is thus no size increase. As the OSM-based graph has many degree-2 vertices, this effect dominates and explains the comparatively small size increase. The search space sizes are nearly identical. For example the KaHIP search space contains 117K arcs for the DIMACS Europe graph, whereas it contains 119K arcs for the OSM Europe graph. This effect is explained by the fact, that both data sources correspond to almost the same geographical region. The mountains and rivers are thus in the same locations and the number of roads through these geographic obstacles are the same in both graphs, i. e., both graphs have very similar recursive separators. The small size increase is explained by the fact that the OSM-based graph also includes Eastern Europe.

Customization performance is reported in Table 3.21. As the OSM-based graph has more arcs, the customization times are higher on

that graph. On the DIMACS Europe graph 0.61 s are needed whereas 1.7 s are needed on OSM Europe for the KaHIP order and 16 threads, which is a surprisingly small gap considering the differences in input sizes. Eliminating the degree-2 vertices from the input should further narrow this gap. As the search space sizes are very similar, it is not surprising that the query performance reported in Table 3.22 is nearly identical.

FURTHER DIMACS-INSTANCES. During the DIMACS challenge on shortest path [DGJ09] several benchmark instances were made available. Among them is the Europe instance used throughout our in-depth experiments in previous sections. Besides this instance, also a set of graphs representing the road network of the USA was published. In Table 3.23 we report experiments for these additional DIMACS road graphs. Other than the DIMACS-Europe instance, these USA instances originate from the U.S. Census Bureau. Note that the USA instances have some known data quality issues: The graphs are generally undirected (no one-way streets) and highways are sometimes not connected at state borders. The DIMACS-Europe comes from another data source and does not have these limitations. This is the reason why we focus on the Europe instance in the main part of our evaluation.

However, as the graphs are undirected we can evaluate the impact that using a single undirected metric has on customization running times compared to using two directed metrics (as used on DIMACS Europe). Experiments using a single metric are marked with "Uni" in the table, whereas the experiments with two metrics are marked with "Bi". The query running times are very similar. This is not surprising as the number of relaxed arcs does not depend on whether one or two weights are used. For larger graphs there is a slightly larger difference in running times. We believe that this is a cache effect. As the "Bi" variant has twice as many weights, less arcs fit into the L3 cache. For the smaller graphs this effect does not occur because the higher CH levels occupy less memory than the cache's size and thus doubling the memory consumption is non-problematic.

The difference in customization times between the two variants is larger. The number of enumerated triangles is the same, but twice as many instructions are executed per triangle. We would thus expect a factor of 2 difference in the customization running times. However, this factor is only observed on the largest instance. On all smaller instances, the gap is significantly smaller. Again, this is most likely the result of cache effects.

Conclusions

We have extended Contraction Hierarchies (CH) to a three-phase customization approach and demonstrated in an extensive experimental

Table 3.23: Instance sizes and experimental results for the additional DIMACS road graphs graphs. The instances are weighted by travel time. They are undirected, i. e., no one-way streets exist and the weight of an arc corresponds to its backward arc's weight. The "Uni" numbers exploit this and have one weight per CH arc, whereas the "Bi" numbers do not and have two weights per CH arc. We report the number of vertices, directed arcs after removing multi-arcs, arcs in the CH search graph, the time needed to do a full non-amortized customization with 1 and 16 threads, the average running time of Dijkstra's algorithm with stopping criterion, and the average running time of an elimination-tree distance query. The order were computed with KaHIP. We averaged results for 10 000 queries where s and t were picked uniformly at random. We only do a basic customization and no perfect customization.

				Customization [ms]					CH Query	
	Vertices	Arcs	CH Arcs	1 thread		16 threads		Dijkstra	[µs]	
Graph	[$\cdot 10^3$]	[$\cdot 10^3$]	[$\cdot 10^3$]	Uni	Bi	Uni	Bi	[µs]	Uni	Bi
NY	264	730	1 547	46	52	11	12	16 303	34	34
BAY	321	795	1 334	29	39	7	8	17 964	20	20
COL	436	1 042	1 692	40	51	12	12	25 505	35	41
FLA	1 070	2 688	4 239	93	117	25	32	63 497	30	26
NW	1 208	2 821	4 266	88	110	24	31	73 045	27	27
NE	1 524	3 868	6 871	195	255	54	57	96 628	68	64
CAL	1 891	4 630	7 587	195	250	61	64	114 047	43	43
LKS	2 758	6 795	12 829	478	646	75	87	175 084	138	149
E	3 599	8 708	14 169	395	514	85	96	233 511	86	88
W	6 262	15 120	24 115	682	894	121	132	425 244	82	84
CTR	14 082	33 867	57 222	2 656	3 592	392	416	1 050 314	276	285
USA	23 947	57 709	97 902	3 617	7 184	698	979	1 883 053	264	286

evaluation that our Customizable Contraction Hierarchies approach is practicable and efficient not only on real world road graphs but also on game maps. We have proposed new algorithms that improve on the state-of-the-art for nearly all stages of the tool chain: Using our contraction graph data structure, a metric-independent CH can be constructed faster than with the established approach based on dynamic arrays. We have shown that the customization phase is essentially a triangle enumeration algorithm. We have provided two variants of the customization: The basic variant yields faster customization running times, while perfect customization and witness search computes CHs with a provable minimum number of shortcuts within seconds given a metric-independent vertex order. We proposed an elimination-tree based query that unlike previous approaches is not based on Dijkstra's algorithm and thus does not use a priority queue. This results in significantly lower overhead per visited arc, enabling faster queries.

Because of the connection to treewidth, further investigation into algorithms [CZ00; PWK12] that explicitly exploit treewidth, seems promising. In that regard, determining the precise treewidth of road networks could prove useful.

Good separators are the foundation of Customizable Contraction Hierarchies: Finding better separators directly improves both customization as well as query performance. For our purposes, the time required to compute good separators was of no primary concern (we do it once per graph). Hence, *our* nested dissection implementation *based* on KaHIP [SS13] was not optimized for speed but rather to demonstrate that good separators exist. If this may seem too slow, not that, since we performed our experiments, significant improvements have been made in this domain [HS15; SS15; Weg14], resulting in decreased query and customization times, and reduced memory consumption [HS15] for Customizable Contraction Hierarchies.

3.2 TIME-DEPENDENT CUSTOMIZABLE ROUTE PLANNING

While customization can easily handle live traffic data, for better travel time estimation on medium and long-distance path, one should consider short-term traffic predictions [DKKT13] and long-term traffic patterns (e. g., rush hours, observed from historic traffic data). In the following, we consider a *dynamic, time-dependent route planning* problem, where time-dependent functions are updated to respect current traffic incidents or short-term traffic predictions.

As discussed in the previous section, both CRP and CCH provide good customization and query search space bounds on networks of recursive small and balanced separators. In practice, this translates to a strong robustness towards different input metrics. However, this analysis only holds for scalar metrics: In case of multi-variate arc costs (e. g., multiple criteria, time-dependency, electric vehicle constraints), growth in shortcut complexity easily dominates the overall effort and space requirements. For Electric Vehicle Customizable Route Planning (EVCRP) [BDPW13], fortunately, this was experimentally shown not to be case.

However, as examined in the following, time-dependent travel times are a much more challenging non-scalar metric when applied to partition-based route planning. Integrating profile search into the customization phase and computing time-dependent overlays, we observe that, unlike for EVCRP [BDPW13], a naïve implementation fails: Shortcuts on higher-level overlays become too expensive to be kept in memory (and too expensive to evaluate during queries). Therefore, we cannot follow the approach of Time-Dependent Contraction Hierarchies (TCH) [BGSV13], where approximation, if at all, was applied in a post-processing step. Instead, in order to reduce functional complexity, we propose to iteratively approximate overlay arcs after each level's customization. Then, in accordance to theory [FHS14], even slight approximation suffices to make our approach practical. The resulting algorithmic framework enables interactive queries with low average and maximum error in a very realistic scenario consisting of live traffic, short-term traffic predictions, and historic traffic patterns, while also supporting and being robust to user preferences.

The rest of this section is organized as follows. Section 3.2.1 introduces necessary notation and foundations, while our approach is presented in Section 3.2.2. In Section 3.2.3, we provide an extensive experimental evaluation.

3.2.1 *Preliminaries*

Throughout Section 3.2, we use the following notation and concepts:

A road network is modeled as a directed *graph* $G = (V, A)$ with $n = |V|$ *vertices* and $m = |A|$ *arcs*, where vertices $v \in V$ correspond to in-

tersections and arcs $(u, v) \in A$ to road segments. An s–t-*path* P (in G) is a sequence $P_{s,t} = [v_1 = s, v_2, \ldots, v_k = t]$ of vertices such that $(v_i, v_{i+1}) \in A$. If s and t coincide, we call P a *cycle*. Every arc a has assigned a *periodic travel-time function* $f_a \colon \Pi \to \mathbb{R}^+$, mapping departure time within period $\Pi = [0, \pi]$ to travel time. Given a departure time τ at s, the (time-dependent) travel time $\tau_{[s,\ldots,t]}$ of an s–t-path is obtained by consecutive function evaluation, i. e., $\tau_{[s,\ldots,v_i]} = f_{(v_{i-1},v_i)}(\tau_{[s,\ldots,v_{i-1}]})$. We assume that functions are piecewise linear and represented by *breakpoints*. We denote by $|f|$ the number of breakpoints of a function f. Moreover, we define $f^{\max}$ as the maximum value of f, i.e., $f^{\max} = \max_{\tau \in \Pi} f(\tau)$. Analogously, $f^{\min}$ is the minimum value of f. A function f is *constant* if $f \equiv c$ for some $c \in \Pi$. We presume that functions fulfill the *FIFO property*, i. e., for arbitrary $\sigma \leqslant \tau \in \Pi$, the condition $\sigma + f(\sigma) \leqslant \tau + f(\tau)$ holds (waiting at a vertex never pays off). Unless waiting is allowed at vertices, the shortest-path problem becomes $\mathcal{NP}$-hard if this condition is not satisfied for all arcs [Dea04; SOS98]. Given two functions f, g, the *link* operation is defined as $\text{link}(f, g) := f + g \circ (\text{id} + f)$, where id is the identity function and $\circ$ is function composition. The result $\text{link}(f, g)$ is piecewise linear again, with at most $|f| + |g|$ breakpoints (namely, at departure times of breakpoints of f and backward projections of departure times of points of g). We also define *merging* of f and g by $\text{merge}(f, g) := \min(f, g)$. The result of merging piecewise linear functions is piecewise linear, and the number of breakpoints is in $\mathcal{O}(|f| + |g|)$ (containing breakpoints of the two original functions and at most one intersection per linear segment). Linking and merging are implemented by coordinated linear sweeps over the breakpoints of the corresponding functions.

The *(travel-time) profile* of a path $P = [v_1, \ldots, v_k]$ is the function $f_P \colon \Pi \to \mathbb{R}^+$ that maps departure time τ at v_1 to travel time on P. Starting at $f_{[v_1,v2]} = f_{(v_1,v_2)}$, we obtain the desired profile by consecutively applying the link operation, i. e., $f_{[v_1,\ldots,v_i]} = \text{link}(f_{[v_1,\ldots,v_{i-1}]}, f_{(v_{i-1},v_i)})$. Given a set $\mathcal{P}$ of s–t-paths, the corresponding s–t-*profile* is $f_{\mathcal{P}}(\tau) = \min_{P \in \mathcal{P}} f_P(\tau)$ for $\tau \in \Pi$, i. e., the *minimum profile* over all paths in $\mathcal{P}$. The s–t-profile maps departure time to minimum travel time for the given paths. It is obtained by (iteratively) merging the respective paths.

A *partition* of V is a set $\mathcal{C} = \{C_1, \ldots, C_k\}$ of disjoint vertex sets with $\bigcup_{i=1}^{k} C_i = V$. More generally, a *nested multi-level partition* consists of sets $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$ such that $\mathcal{C}^\ell$ is a partition of V for all $\ell \in \{1, \ldots, L\}$, and additionally for each cell C_i in $\mathcal{C}^\ell, \ell < L$, there is a partition $\mathcal{C}^{\ell+1}$ at level $\ell + 1$ containing a cell C_j with $C_i \subseteq C_j$. We call C_j the *supercell* of C_i. For consistency, we define $\mathcal{C}^0 = \{\{v\} \mid v \in V\}$ and $\mathcal{C}^{L+1} = \{V\}$. Vertices u and v are *boundary vertices* on level ℓ if they are in different cells of $\mathcal{C}^\ell$. The corresponding arc $(u, v) \in A$ is a *boundary arc* on level ℓ.

QUERY VARIANTS AND ALGORITHMS. Given a departure time dep and vertices s and t, an *earliest-arrival (EA)* query asks for the minimum

travel time from s to t when departing at time dep. Similarly, a *latest-departure (LD)* query asks for the minimum travel time of an s–t-path arriving at time dep. A *profile query* for given source s and target t asks for the minimum travel time at every possible departure time dep, i. e., a profile $f_{s,t}$ from s to t (over all s–t-paths in G). EA queries can be handled by a time-dependent variant of Dijkstra's algorithm [Dre69], which we refer to as *TD-Dijkstra*. It maintains (scalar) *arrival time labels* $d(\cdot)$ for each vertex, initially set to dep for the source s (∞ for all other vertices). In each step, a vertex u with minimum $d(u)$ is extracted from a priority queue (initialized with s). Then, the algorithm *relaxes* all outgoing arcs (u, v): if $d(u) + f_{(u,v)}(d(u))$ improves $d(v)$, it updates $d(v)$ accordingly and adds (or updates) v in the priority queue. LD queries are handled analogously by running the algorithm from t, relaxing incoming instead of outgoing arcs, and maintaining departure time labels.

Profile queries can be solved by *Profile-Dijkstra* [DW09b], which is based on linking and merging. It generalizes Dijkstra's algorithm, maintaining s–v profiles f_v at each vertex $v \in V$. Initially, it sets $f_s \equiv 0$, and $f_v \equiv \infty$ for all other vertices. The algorithm continues along the lines of TD-Dijkstra, using a priority queue with scalar keys $f_v^{\min}$. For extracted vertices u, arc relaxations propagate profiles rather than travel times, computing $g := \text{link}(f_u, f_{(u,v)})$ and $f_v := \text{merge}(f_v, g)$ for outgoing arcs (u, v). As shown by Foschini et al. [FHS14], the number of breakpoints of the profile of an s–v-paths can be superpolynomial, and hence, so is space consumption *per vertex label* and the running time of Profile-Dijkstra in the worst case. Accordingly, it is not feasible for large-scale instances, even in practice [DW09b].

3.2.2 *Our Approach*

We propose *Time-Dependent CRP (TDCRP)*, a speedup technique for time-dependent route planning allowing fast integration of user-dependent metric changes. Additionally, we enable current and/or predicted traffic updates with limited departure time horizon (accounting for the fact that underlying traffic situations resolve over time). To take historic knowledge of traffic patterns into account, we use functions of departure time at arcs. This conceptual change has important consequences: For plain CRP, the topology data structures is fixed after preprocessing, enabling several micro-optimizations with significant impact on customization and query [DGPW15]. In our case, functional complexity is metric-dependent (influenced by, e. g., user preferences) and has to be handled dynamically during customization. Hence, for adaptation to dynamic time-dependent scenarios, we require new data structures and algorithmic changes during customization. Below, we recap the three-phase workflow of CRP [DGPW15] that allows fast integration of user-dependent routing preferences, describing

its extension to TDCRP along the way. In particular, we incorporate *profile queries* into the customization phase to obtain *time-dependent* shortcuts. Moreover, we adapt the query phase to efficiently compute time-dependent shortest routes.

3.2.2.1 *Preprocessing*

The (metric-independent) preprocessing step of CRP computes a multi-level partition of the vertices, with given number L of levels. Several graph partition algorithms tailored to road networks exist, providing partitions with balanced cell sizes and small cuts [DGRW11; HS15; SS12b; SS15]. For each level $\ell \in \{1, \dots, L\}$, the respective partition $\mathcal{C}^\ell$ induces an *overlay graph* H^ℓ, containing all boundary vertices and boundary arcs in $\mathcal{C}^\ell$ and *shortcut* arcs between boundary vertices within each cell $C_i^\ell \in \mathcal{C}^\ell$. We define $\mathcal{C}^0 = \{\{v\} \mid v \in V\}$ and $H^0 := G$ for consistency. Building the overlay, we use the clique matrix representation, storing cliques of boundary vertices in matrices of contiguous memory [DGPW15]. Matrix entries represent *pointers* to functions (whose complexity is not known until customization). This dynamic data structure rules out some optimizations for plain CRP, such as microcode instructions, that require preallocated ranges of memory for the metric [DGPW15]. To improve locality, all functions are stored in a single array, such that profiles corresponding to outgoing arcs of a boundary vertex are in contiguous memory.

3.2.2.2 *Customization*

In the customization phase, costs of all shortcuts (added to the overlay graphs during preprocessing) are computed. We run profile searches to obtain these time-dependent costs. In particular, we require, for each boundary vertex u (in some cell C_i at level $\ell \geqslant 1$), the time-dependent distances for all $\tau \in \Pi$ to all boundary vertices $v \in C_i$. To this end, we run a profile query on the overlay $H^{\ell-1}$. By design, this query is restricted to *subcells* of C_i, i. e., cells C_j on level $\ell - 1$ for which $C_j \subseteq C_i$ holds. This yields profiles for all outgoing (shortcut) arcs (u, v) in C_i from u. On higher levels, previously computed overlays are used for faster computation of shortcuts. Unfortunately, profile queries are expensive in terms of both running time and space consumption. Below, we describe improvements to remedy these effects, mostly by tuning the profile searches.

IMPROVEMENTS. The main bottleneck of profile search is performing link and merge operations, which require linear time in the function size (cf. Section 3.2.1). To avoid unnecessary operations, we explicitly compute and store the minimum $f^{\min}$ and the maximum $f^{\max}$ of a profile f in its corresponding label and in shortcuts of overlays. These values are used for early pruning, avoiding costly link and

merge operations: Before relaxing an arc (u, v), we check whether $f_u^{\min} + f_{(u,v)}^{\min} > f_v^{\max}$, i. e., the minimum of the linked profile exceeds the maximum of the label at v. If this is the case, the arc (u, v) does not need to be relaxed. Otherwise, the functions are linked. We distinguish four cases, depending on whether the first or second function are constant, respectively. If both are constant, linking becomes trivial (summing up two integers). If one of them is constant, simple shift operations suffice (we need to distinguish two cases, depending on which of the two functions is constant). Only if no function is constant, we apply the link operation.

After linking $f_{(u,v)}$ to f_u, we obtain a tentative label $\tilde{f}_v$ together with its minimum $\tilde{f}_v^{\min}$ and maximum $\tilde{f}_v^{\max}$. Before merging f_v and $\tilde{f}_v$, we run additional checks to avoid unnecessary merge operations. First, we perform bound checks: If $\tilde{f}_v^{\min} > f_v^{\max}$, the function f_v remains unchanged (no merge necessary). Note that this may occur although we checked bounds before linking. Conversely, if $\tilde{f}_v^{\max} < f_v^{\min}$, we simply replace f_v by $\tilde{f}_v$. If the checks fail, and one of the two functions is constant, we must merge. But if f_v and $\tilde{f}_v$ are both nonconstant, one function might still dominate the other. To test this, we do a coordinated linear-time sweep over the breakpoints of each function, evaluating the current line segment at the next breakpoint of the other function. If during this test $\tilde{f}_v(\tau) < f_v(\tau)$ for any point $(\tau, \cdot)$, we must merge. Otherwise we can avoid the merge operation and its numerically unstable line segment intersections.

Additionally, we use *clique flags*: For a vertex v, define its *parents* as all direct predecessors on paths contributing to the profile at the current label of v. For each vertex v of an overlay H^ℓ, we add a flag to its label that is true if *all* parents of v belong to the same cell at level ℓ. This flag is set to true whenever the corresponding label f_v is replaced by the tentative function $\tilde{f}_v$ after relaxing a clique arc (u, v), i. e., the label is set for the first time or the label f_v is dominated by the tentative function $\tilde{f}_v$. It is set to false if the vertex label is partially improved after relaxing a boundary arc. For flagged vertices, we do not relax outgoing clique arcs, as this cannot possibly improve labels within the same cell (due to the triangle inequality and the fact that we use full cliques).

PARALLELIZATION. Cells on a given level are processed independently, so customization can be parallelized naturally, assigning cells to different threads [DGPW15]. In our scenario, however, workload is strongly correlated with the number of time-dependent arcs in the search graph. It may differ significantly between cells: In realistic data sets, the distribution of time-dependent arcs is clearly not uniform, as it depends on the road type (highways vs. side roads) and the area (rural vs. urban). To balance load, we parallelize per boundary vertex (and not per cell).

Shortcut profiles are written to dynamic containers, as the number of breakpoints is not known in advance. Thus, we must prohibit parallel (writing) access to these data structure. One way to solve this is to make use of locks. However, this is expensive if many threads try to write profiles at the same time. Instead, we use thread-local profile containers, i. e., each thread uses its own container to store profiles. After customization of each level, we synchronize data by copying profiles to the global container sequentially. To improve spatial locality during queries, we maintain the relative order of profiles wrt. the matrix layout (so profiles of adjacent vertices are likely to be contiguous in memory). Since relative order within each thread-local containers is maintained easily (by running queries accordingly), we can use merge sort when writing profiles to the global container.

APPROXIMATION. On higher levels of the partition, shortcuts represent larger parts of the graph. Accordingly, they contain more breakpoints and consume more space. This makes profile searches fail on large graphs due to insufficient memory, even on modern hardware. Moreover, running time is strongly correlated to the complexity of profiles. To save space and time, we *simplify* functions during customization. To this end, we use the algorithm of Imai and Iri [II87]. For a maximum (relative or absolute) error bound ε, it computes an approximation of a given piecewise linear function with minimum number of breakpoints. In TCH [BGSV13], this technique is applied after preprocessing to reduce space consumption. Instead, we use the algorithm to simplify profiles after computing all shortcuts of a certain level. Therefore, searches on higher levels use approximated functions from lower levels, leading to slightly less accurate profiles but faster customization. The bound ε is a tuning parameter: Larger values allow faster customization, but decrease quality. Also, approximation is not necessarily applied on all levels, but can be restricted to the higher ones. Note that after approximating shortcuts, the triangle inequality may no longer hold for the corresponding overlay. This is relevant when using clique flags: They yield faster profile searches, but slightly decrease quality (additional arc relaxations may improve shortcut bounds).

3.2.2.3 *Live Traffic and Short-Term Traffic Predictions*

Updates due to, e. g., live traffic, require that we rerun parts of the customization. Clearly, we only have to run customization for *affected* cells, i. e., cells containing arcs for which an update is made. We can do even better if we exploit that live traffic and short-term updates only affect a limited *time horizon*. Thus, we do not propagate updates to boundary vertices that cannot reach an affected arc before the end of its time horizon.

We assume that short-term updates are *partial functions* $f\colon [\pi', \pi''] \to \mathbb{R}^+$, where $\pi' \in \Pi$ and $\pi'' \in \Pi$ are the *beginning* and *end* of the time horizon, respectively. Let $a_1 = (u_1, v_1), \ldots, a_k = (u_k, v_k)$ denote the *updated* arcs inside some cell C_i at level ℓ, and let $f_1, \ldots, f_k$ be the corresponding partial functions representing time horizons. Moreover, let τ be the current point in time. To update C_i we run, on its induced subgraph, a backward *multi-target* latest departure (LD) query from the tails of all updated arcs. In other words, we initially insert the vertices $u_1, \ldots, u_k$ into the priority queue. For each $i \in \{1, \ldots, k\}$, the label of u_i is set to π_i'', i. e., the end of the time horizon $[\pi_i', \pi_i'']$ of the partial function f_i. Consequently, the LD query computes, for each vertex of the cell C_i, the latest possible departure time such that some affected arc is reached before the end of its time horizon. Whenever the search reaches a boundary vertex of the cell, it is marked as *affected* by the update. We stop the search as soon as the departure time label of the current vertex is below τ. (Recall that LD visits vertices in decreasing order of departure time.) Thereby, we ensure that only such boundary vertices are marked from which an updated arc can be reached in time.

Afterwards, we run profile searches for C_i as in regular customization, but only from affected vertices. For profiles obtained during the searches, we test whether they improve the corresponding stored shortcut profile. If so, we add the *affected* interval of the profile for which a change occurs to the set of time horizons of the next level. If shortcuts are approximations, we test whether the change is *significant*, i. e., the maximum difference between the profiles exceeds some bound. We continue the update process on the next level accordingly.

3.2.2.4 *Queries*

The query algorithm makes use of shortcuts computed during customization to reduce the search space. Given a source s and a target t, the search graph consists of the overlay graph induced by the top-level partition $\mathcal{C}^L$, all overlays of cells of lower levels containing s or t, and the level-0 cells in the input graph G that contain s or t. Note that the search graph does not have to be constructed explicitly, but can be obtained on-the-fly [DGPW15]: At each vertex v, one computes the highest levels $\ell_{s,v}$ and $\ell_{v,t}$ of the partition such that v is not in the same cell of the partition as s or t, respectively (or zero, if v is in the same level-1 cell as s or t). Then, one relaxes outgoing arcs of v only at level $\min\{\ell_{s,v}, \ell_{v,t}\}$ (recall that $H^0 = G$).

To answer EA queries, we run TD-Dijkstra on this search graph. For faster queries, we make use of the minimum values $f^{\min}_{(u,v)}$ stored at arcs: We do not relax an arc (u, v) if $d(u) + f^{\min}_{(u,v)}$ does not improve $d(v)$. Thereby, we avoid costly function evaluation. We could also use clique flags for EA queries. However, in our implementation, when combined

with approximated profiles, we have observed rare but high maximum errors. Hence we de-activate clique flags for all query experiments.

To answer profile queries, Profile-Dijkstra can be run on the CRP search graph, using the same optimizations as described in Section 3.2.2.2.

3.2.2.5 *Implementation Details*

Implementation of the functional operations (evaluation, link, merge, and subroutines) requires considerable effort and is a great source for numerical imprecision. We tested several implementation variants and settled on using fixed-point arithmetics at milliseconds resolution. We also found it beneficial to use half-plane tests (which require no division) to determine whether an intersection between line segments occurs, before computing the actual intersection.

We reorder vertices of the graph, moving boundary vertices to the front, and ordering vertices of the same level by cell IDs [DGPW15]. This improves locality of subsequent memory accesses. In a naïve implementation, each vertex requires its own vertex label. Since the search graph is known in advance, we can reduce the number of vertex labels to save space. Instead of explicitly extracting the search graph, we compute the ranges of IDs of vertices of the current cell for each level (note that there is at most one range per level, due to vertex reordering). Then, we can remap the ranges of each level (for source and target cell, respectively) to a smaller, global range of vertex indices. The size of the global range depends on the maximum cell size, which is known after preprocessing. The following (mixed) variant worked best in our experiments: We only remap bottom-level inner-cell indices (the majority of vertices), while keeping a distinct vertex label for every boundary vertex in the graph. Thereby, we save a significant amount of space, improve locality, but keep vertex mapping overhead limited during queries.

To reset labels between queries, rather than using standard approaches (like timestamps), we again exploit the vertex reordering: We store, for every cell, its vertex ranges, and reset only the (at most two) cells per level touched during a query, along with all top level vertices. With labels being on adjacent ranges, this can be done efficiently in practice.

Finally, we can save some space by storing cell IDs only at boundary vertices. Before running the actual query algorithm, the source and target cell can be retrieved by running an additional DFS in their respective (bottom-level) cells with negligible overhead.

3.2.3 *Experiments*

We implemented all algorithms in C++ using g++ 4.8 (flag -O3) as compiler. Experiments were conducted on a dual 8-core Intel Xeon

Table 3.24: Network properties. We report the number of vertices and arcs of the routing graph, and as a measure of time-dependency, the total amount of break points in the whole network as well as the average time-dependent arc complexity.

				# Break points	
Network	# Vertices	# Arcs	Time-dep. arcs	Total	Avg.
Berlin	443 191	988 493	271 688 (27.5%)	3 464 241	12.8
Germany	4 692 091	10 805 429	777 984 (7.2%)	12 714 370	16.3
Europe	18 010 173	42 188 664	2 609 651 (6.2%)	29 362 693	11.3

E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. We ran customization in parallel (using all 16 threads) and queries sequentially.

INPUT DATA AND METHODOLOGY. Our test instances are based on the road network of Germany ($n = 4.7$ million, $m = 10.8$ million) and Western Europe ($n = 18$ million, $m = 42.2$ million), kindly provided by PTV AG,[12] and the road network of Berlin/Brandenburg ($n = 443$ k, $m = 988$ k), kindly provided by TomTom. All inputs contain time-dependent data. For the Germany and Berlin instance, data stems from historical traffic patterns. For both instances, we extracted the 24 hour profile of a Tuesday. For Western Europe, travel time functions were generated synthetically [NDLS12]. See Table 3.24 for details on inputs. For partitioning, we used PUNCH [DGRW11], which is explicitly developed for road networks and aims at minimizing the number of boundary arcs. For Germany, we use a 5-level partition, with maximum cell sizes of $2^{[4:8:12:15:18]}$. For Europe, we use a 6-level partition, with maximum cell sizes $2^{[4:8:11:14:17:20]}$. Finally, we use a 5-level partition for Berlin, with cell size restricted to $2^{[4:8:11:14:17]}$. Compared to plain CRP, we use partitions with more levels, to allow fine-grained approximation. Computing the partition took 20 seconds for Berlin, 5 minutes for Germany, and 23 minutes for Europe. Given that road topology changes rarely, this is sufficiently fast in practice.

EVALUATING CUSTOMIZATION. Table 3.25 details customization for different approximation parameters ε on the Europe instance. We report, for several choices of ε and for each level of the partition, figures on the complexity of shortcuts in the overlays and the parallelized customization time. The first block shows figures for exact profile computation. Customization had to be aborted after the fourth level, because the 64 GiB of main memory were not sufficient to store the profiles of all vertex labels. For remaining levels, we clearly

12 The Germany and Europe instances can be obtained easily for scientific purposes, see http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php

Table 3.25: Customization performance on Europe for varying approximation parameters (ε). We report, per level, the number of breakpoints (bps, in millions) in the resulting overlay, the percentage of clique arcs that are time-dependent (td.clq.arcs), average complexity of time-dependent arcs (td.arc.cplx), as well as customization time. Without approximation, Levels 5 and 6 cannot be computed as they do not fit into main memory.

ε		Lvl1	Lvl2	Lvl3	Lvl4	Lvl5	Lvl6	Total
—	bps [10^6]	99.1	398.4	816.4	1 363.4	—	—	2 677.4
	td.clq.arcs [%]	17.0	52.6	76.0	84.2	—	—	—
	td.arc.cplx	21.0	68.9	189.0	509.3	—	—	—
	time [s]	11.4	52.0	152.9	206.2	—	—	375.7
0.01%	bps [10^6]	75.7	182.7	244.6	240.8	149.3	59.2	952.2
	td.clq.arcs [%]	17.0	52.6	76.0	84.2	85.2	82.5	—
	td.arc.cplx	16.0	31.6	56.6	90.0	108.6	108.0	—
	time [s]	4.5	18.0	32.7	82.1	150.3	151.5	439.1
0.1%	bps [10^6]	60.7	107.5	111.5	87.9	47.9	17.6	432.9
	td.clq.arcs [%]	17.0	52.7	76.0	84.2	85.2	82.5	—
	td.arc.cplx	12.9	18.6	25.8	32.8	34.8	32.1	—
	time [s]	4.2	16.0	21.4	40.7	62.4	55.0	199.7
1.0%	bps [10^6]	45.7	58.0	45.6	29.2	14.7	5.4	198.5
	td.clq.arcs [%]	17.0	52.7	76.0	84.2	85.2	82.5	—
	td.arc.cplx	9.7	10.0	10.6	10.9	10.7	9.8	—
	time [s]	4.1	14.1	14.8	22.7	29.6	24.1	109.2

see the strong increase in the total number of breakpoints per level. Also, the relative amount of time-dependent arcs rises with each level, since shortcuts become longer. Customization time clearly correlates with profile complexity, from 10 seconds on the lowest level, to more then three minutes on the fourth. When approximating, we see that customization becomes faster for larger values of ε. We apply approximation to all levels of the partition (using it only on the topmost levels did not provide significant benefits in preliminary experiments). Recall that higher levels work on approximated shortcuts of previous levels, so ε does not provide a bound on the error of the shortcuts. We see that even a very small value (0.01 %) yields a massive drop of profile complexity (more than a factor 5 at Level 4), and immediately allows full customization. For reasonably small values ($\varepsilon = 0.1\,\%$, $\varepsilon = 1.0\,\%$), we see that customization becomes much faster (less than two minutes for $\varepsilon = 1.0\,\%$). In particular, this is fast enough for traffic updates. Even for larger values of ε, the higher levels are far more expensive:

Table 3.26: Query performance on Europe as a trade-off between customization effort and approximation. For customization, we set different approximation parameters (ε) and disable ($\circ$) or enable ($\bullet$) clique flags (Cl.). For the different settings, we report query performance in terms of number of vertices extracted from the queue, scanned arcs, evaluated function breakpoints (# Bps), running time, and average and maximum error, each averaged over 100 000 random queries. As we employ approximation per level, resulting query errors can be higher than the input parameter.

Customization			Query					
Approx. ε	Cl.	Time [s]	# Vertices	# Arcs	# Bps	Time [ms]	Err. [%] avg.	max.
0.01 %	$\circ$	1 155.1	3 499	541 091	433 698	14.69	<0.01	0.03
0.01 %	$\bullet$	439.1	3 499	541 090	434 704	14.53	<0.01	0.03
0.10 %	$\circ$	533.0	3 499	541 088	96 206	7.63	0.04	0.28
0.10 %	$\bullet$	199.7	3 499	541 088	99 345	6.47	0.04	0.29
1.00 %	$\circ$	284.4	3 499	541 080	67 084	5.66	0.51	3.15
1.00 %	$\bullet$	109.2	3 499	541 058	70 202	5.75	0.54	3.21

This is due to the increasing amount of time-dependent arcs, slowing down profile search.

EVALUATING CUSTOMIZATION AND QUERIES. In Table 3.26, we show query performance for different values of the approximation parameter ε on the Europe instance. We also show the effect of using clique flags during customization: they improve customization performance by about a factor of 2.6, while having a negligible influence on query results. For each value of ε, we report timings as well as average and maximum error over 100 000 point-to-point queries. For each query, the source and target vertex and the departure time were picked uniformly at random. Similar to customization, the data shows that query times decrease with higher approximation ratio. Again, this is due to the smaller number of breakpoints in profiles (observe that the number of visited vertices and arcs is almost identical in all cases). As expected, the average and maximum error increase with ε. However, we see that even for the parameter choice $\varepsilon = 1.0\,\%$, the maximum error is very low (about 3 %). Moreover, query times are quite practical for all values of ε, ranging from 5 ms to 15 ms. In summary, our approach allows query times that are fast enough for interactive applications, if a reasonable, small error is allowed. Given that input functions are based on statistical input with inherent inaccuracy, the error of TDCRP is more than acceptable for realistic applications.

Table 3.27: Robustness comparison for TCH [BGSV13] and TDCRP. For different input instances, we report timing of metric-dependent preprocessing (always run on 16 cores) and sequential queries. Query times are averaged over the same 100 000 random queries as in Table 3.26.

	TCH		TDCRP	
Network	Prepro. [s]	Query [ms]	Custom. [s]	Query [ms]
Europe	1 479	1.37	109	5.75
Europe, bad traffic	7 772	5.87	208	8.01
Europe, avoid highways	8 956	19.54	127	8.29

EVALUATING ROBUSTNESS. We also evaluate robustness of our approach against dynamic updates and user-dependent custom metrics. The first scenario (bad traffic) simulates a highly congested graph: for every time-dependent arc in the Western Europe instance with associated travel-time function f, we replace f by f' defined as $f'(\tau) := 2(f(\tau) - f^{\min}(\tau)) + f^{\min}(\tau)$, while maintaining the FIFO property on f'. In the second scenario, we consider user restrictions (avoid highways). For each scenario, customization and the same set of 100 000 random queries as before are run on the respective modified instance. (Hence, we do not remove highways for the second scenario, setting very high costs instead.) Table 3.27 compares results of the original instance (Europe) to the modified ones. Besides our approach, which is run using parameter $\varepsilon = 1.0$ for customization, we also evaluate TCH [BGSV13], the fastest known approach for time-dependent route planning. All measurements for TCH are based on this freely available implementation: `https://github.com/GVeitBatz/KaTCH`.

While TCH allows faster queries on the original instance, we see that running times increase significantly for the modified ones. Preprocessing time also increases to several hours in both cases. In the first scenario (bad traffic), this can be explained by a larger number of paths that are relevant at different points in time (more congested roads need to be bypassed). Consequently, customization time of TDCRP rises as well but by a much smaller factor.

In the second scenario (avoid highways), the TCH hierarchy clearly deteriorates. While TDCRP is quite robust to this change (both customization and query times increase by less than 50 %), TCH queries slow down by more than an order of magnitude.

While possibly subject to implementation, our experiment indicates that underlying vertex orderings of TCH are not robust against less well-behaved metrics. Similar effects can be shown for scalar Contraction Hierarchies (CH) on metrics reflecting, e. g., travel distance [DGPW15; GSSV12]. In summary, TDCRP is much more robust in both scenarios.

Table 3.28: Comparison of time-dependent speedup techniques on instances of Berlin, Germany, and Europe. We present figures *as reported* for variants of TDCALT [DN12], SHARC [Del11], TCH and ATCH [BGSV13], FLAT [KMP+16], and TDCRP; see Table 3.29 for scaled results. For preprocessing, customization, and live traffic updates, we show the number of threads used (Thr.). For EA queries, we present average numbers on queue extractions (# Vert.), scanned arcs, *sequential* running time in milliseconds, and average and maximum relative error. For FLAT, the Berlin instance differs slightly from ours.

			Preprocessing		Customization		Traffic	EA Queries				
			Time	Space	Time	Space	Time			Time	Err. [%]	
Algorithm	Inst.	Thr.	[h:m:s]	[B/n]	[m:s]	[B/n]	[m:s]	# Vert.	# Arcs	[ms]	avg.	max.
TDCALT	Ger	1	9:00	50	—	—	n/a	3 190	12 255	5.36	—	—
TDCALT-K1.15	Ger	1	9:00	50	—	—	n/a	1 593	5 339	1.87	0.05	13.84
eco L-SHARC	Ger	1	1:18:00	219	—	—	—	2 776	19 005	6.31	—	—
heu SHARC	Ger	1	3:26:00	137	—	—	—	818	1 611	0.69	n/a	0.61
TCH	Ger	8	6:18	995	1:14	995	—	520	5 820	0.75	—	—
ATCH (1.0)	Ger	8	6:18	239	1:14	239	—	588	7 993	1.24	—	—
inex. TCH (0.1)	Ger	8	6:18	286	1:14	286	—	642	7 138	0.70	0.02	0.10
inex. TCH (1.0)	Ger	8	6:18	214	1:14	214	—	654	7 271	0.69	0.27	1.01
inex. TCH (2.5)	Ger	8	6:18	172	1:14	172	—	668	7 429	0.72	0.79	2.44
inex. TCH (10.0)	Ger	8	6:18	113	1:14	113	—	898	10 109	1.06	3.84	9.75
FLAT/FCA	Ger	6	≈30:00:00	≈10 000	—	—	0:37	1 122	n/a	1.27	n/a	1.53
TDCRP (0.1)	Ger	16	04:33	29	0:16	166	0:16	2 152	167 263	1.92	0.05	0.25
TDCRP (1.0)	Ger	16	04:33	29	0:08	77	0:08	2 152	167 305	1.66	0.68	2.85
TDCALT	Eur	1	1:00:00	61	—	—	1:01	60 961	356 527	121.4	—	—
TDCALT-K1.05	Eur	1	1:00:00	61	—	—	1:01	32 405	n/a	62.5	0.01	3.94
TDCALT-K1.10	Eur	1	1:00:00	61	—	—	1:01	12 777	n/a	21.9	0.09	7.88
TDCALT-K1.15	Eur	1	1:00:00	61	—	—	1:01	6 365	32 719	9.2	0.26	8.69
eco L-SHARC	Eur	1	6:49:00	198	—	—	—	18 289	165 382	38.29	—	—
heu SHARC	Eur	1	22:12:00	127	—	—	—	5 031	8 411	2.94	n/a	1.60
TCH	Eur	8	45:44	599	8:02	599	—	1 021	13 681	2.11	—	—
ATCH (1.0)	Eur	8	45:44	208	8:02	208	—	1 223	20 336	2.89	—	—
inex. TCH (0.1)	Eur	8	45:44	239	8:02	239	—	1 722	24 389	2.70	0.02	0.15
inex. TCH (1.0)	Eur	8	45:44	195	8:02	195	—	1 782	25 361	2.76	0.20	1.50
inex. TCH (2.5)	Eur	8	45:44	175	8:02	175	—	1 875	26 948	2.94	0.48	3.37
inex. TCH (10.0)	Eur	8	45:44	144	8:02	144	—	1 801	25 692	2.92	2.88	16.21
TDCRP (0.1)	Eur	16	22:33	32	3:20	237	3:20	3 499	541 088	6.47	0.04	0.29
TDCRP (1.0)	Eur	16	22:33	32	1:49	133	1:49	3 499	541 058	5.75	0.54	3.21
FLAT/FCA	Ber*	6	≈4:30:00	≈60 000	—	—	0:21	93	n/a	0.08	n/a	0.78
TDCRP (0.1)	Ber	16	00:20	28	0:01	134	0:01	840	23 478	0.40	0.08	0.29
TDCRP (1.0)	Ber	16	00:20	28	<0:01	66	<0:01	840	23 479	0.38	0.86	3.14

Table 3.29: Comparison of time-dependent speedup techniques as in Table 3.28. For better comparability across different hardware, we scale *sequential* timings for TDCALT [DN12], SHARC [Del11], TCH and ATCH [BGSV13] to our machine; see Table 3.30 for factors. For FLAT [KMP+16], sequential timings for preprocessing and live traffic updates were not reported (n/a).

		Preprocessing		Customization		Traffic	EA Queries				
		Scaled	Space	Scaled	Space	Scaled			Scaled	Err. [%]	
Algorithm	Inst.	[h:m:s]	[B/n]	[m:s]	[B/n]	[m:s]	# Vert.	# Arcs	[ms]	avg.	max.
TDCALT	Ger	3:14	50	—	—	n/a	3 190	12 255	1.93	—	—
TDCALT-K1.15	Ger	3:14	50	—	—	n/a	1 593	5 339	0.67	0.05	13.84
eco L-SHARC	Ger	28:03	219	—	—	—	2 776	19 005	2.27	—	—
heu SHARC	Ger	1:14:06	137	—	—	—	818	1 611	0.25	n/a	0.61
TCH	Ger	34:10	995	6:59	995	—	520	5 820	0.69	—	—
ATCH (1.0)	Ger	34:10	239	6:59	239	—	588	7 993	1.15	—	—
inex. TCH (0.1)	Ger	34:10	286	6:59	286	—	642	7 138	0.65	0.02	0.10
inex. TCH (1.0)	Ger	34:10	214	6:59	214	—	654	7 271	0.64	0.27	1.01
inex. TCH (2.5)	Ger	34:10	172	6:59	172	—	668	7 429	0.67	0.79	2.44
inex. TCH (10.0)	Ger	34:10	113	6:59	113	—	898	10 109	0.98	3.84	9.75
FLAT/FCA	Ger	n/a	≈10 000	—	—	n/a	1 122	n/a	1.51	n/a	1.53
TDCRP (0.1)	Ger	54:30	29	3:30	166	3:30	2 152	167 263	1.92	0.05	0.25
TDCRP (1.0)	Ger	54:30	29	1:43	77	1:43	2 152	167 305	1.66	0.68	2.85
TDCALT	Eur	21:35	61	—	—	0:22	60 961	356 527	43.67	—	—
TDCALT-K1.05	Eur	21:35	61	—	—	0:22	32 405	n/a	22.48	0.01	3.94
TDCALT-K1.10	Eur	21:35	61	—	—	0:22	12 777	n/a	7.88	0.09	7.88
TDCALT-K1.15	Eur	21:35	61	—	—	0:22	6 365	32 719	3.31	0.26	8.69
eco L-SHARC	Eur	2:27:07	198	—	—	—	18 289	165 382	13.77	—	—
heu SHARC	Eur	7:59:08	127	—	—	—	5 031	8 411	1.06	n/a	1.60
TCH	Eur	4:23:41	599	48:07	599	—	1 021	13 681	1.95	—	—
ATCH (1.0)	Eur	4:23:41	208	48:07	208	—	1 223	20 336	2.68	—	—
inex. TCH (0.1)	Eur	4:23:41	239	48:07	239	—	1 722	24 389	2.50	0.02	0.15
inex. TCH (1.0)	Eur	4:23:41	195	48:07	195	—	1 782	25 361	2.56	0.20	1.50
inex. TCH (2.5)	Eur	4:23:41	175	48:07	175	—	1 875	26 948	2.72	0.48	3.37
inex. TCH (10.0)	Eur	4:23:41	144	48:07	144	—	1 801	25 692	2.70	2.88	16.21
TDCRP (0.1)	Eur	4:30:38	32	47:10	237	—	3 499	541 088	6.47	0.04	0.29
TDCRP (1.0)	Eur	4:30:38	32	25:16	133	—	3 499	541 058	5.75	0.54	3.21
FLAT/FCA	Ber*	n/a	≈60 000	—	—	n/a	93	n/a	0.10	n/a	0.78
TDCRP (0.1)	Ber	4:03	28	0:14	134	0:14	840	23 478	0.40	0.08	0.29
TDCRP (1.0)	Ber	4:03	28	0:09	66	0:09	840	23 479	0.38	0.86	3.14

Table 3.30: Scaling factors for different machines, used in Table 3.28. Each machine's score has been determined by a shared Dijkstra implementation on the same graph. The benchmark tool [BDG+15] is available at http://i11www.iti.uni-karlsruhe.de/~pajor/survey/. These factors have to be taken with a grain of salt, since Dijkstra's algorithm is not a good indicator of cache performance. For example, when scaling on TDCRP performance, instead, we observe a factor of 2.06–2.18 for the Opteron 2218 (which we have access to), depending on the instance.

Machine	Approaches Evaluated	Score [ms]	Scaling
2× 8-core Intel Xeon E5-2670, 2.6 GHz	TDCRP	36 582	—
AMD Opteron 2218, 2.6 GHz	TDCALT [DN12], SHARC [Del11]	101 552	2.78
2× 4-core Intel Xeon X5550, 2.66 GHz	TCH, ATCH [BGSV13]	39 684	1.08
6-core Intel Xeon E5-2643v3, 3.4 Ghz	FLAT/FCA [KMP+16]	30 901	0.84

COMPARISON WITH RELATED WORK. Finally, Table 3.28 provides an overview comparing our results to the most relevant existing approaches for time-dependent route planning. We evaluated our approach on all benchmark instances (Berlin-Brandenburg, Germany, and Europe) for the two fastest variants, i. e., $\varepsilon = 0.1$ and $\varepsilon = 1.0$. For the related work, we show measurements as reported in the respective publication, in the fastest reported variant (e. g., if parallelized). For comparison, we also provide in Table 3.29 single-core performance timings taken from the literature, which we scale to our hardware as detailed in Table 3.30.

For TCH and ATCH [BGSV13], preprocessing can be further split into *node order computation* and *contraction*. Since it has been shown in [BGSV13] that node orders can be re-used for certain other metrics (e. g., other week days), we report running times of the contraction as rudimentary customization times. Recall, however, that our robustness tests in Table 3.27 suggest that there is a limit to the applicability of such a customization approach based on current TCH orders.

We see that our approach competes very well with the previous techniques: While providing query times similar to the fastest existing approaches, TDCRP has by far the lowest metric-dependent preprocessing time (i. e., customization time) and a good parallel speedup (factor 13.9 to 14.2 on Europe for 16 threads). At the same time, resulting average and maximum errors (due to approximating profiles during customization) are similar to previous results and low enough for practical purposes. When parallelized, customization of the whole network is fast enough for regular live-traffic updates: 1 s on Berlin, 8 to 16 s on Germany, and 2 to 3 min on Europe. Note, however, that other approaches are also able to handle live traffic by providing *partial updates* of the preprocessed data: For example, by exploiting the fact that effects of live traffic are locally and temporally limited,

FLAT [KMP+16]) achieves partial update times of 21 s on Berlin and 37 s on Germany. For TDCALT [DN12], partial updates are equally fast on Europe (60.94 s reported, 22 s scaled).

When accounting for differences in hardware and implementation, TDCALT can be preprocessed similarly fast as TDCRP. This makes it an interesting alternative candidate for our scenario (metric customization); since it is mostly based on lower bounds and only light contraction, it might be fairly robust to sensible, user-defined metrics (unlike TCH, cf. Table 3.27). Note, however, that TDCALT on Europe requires a significantly higher approximation to achieve a similar level of query performance (even scaled), yielding a high maximum error. Furthermore, in the evaluated variant, landmarks are chosen after the graph contraction routine, making it hard to parallelize the preprocessing (which also has not been attempted). Additionally, TDCALT allows no practical profile search on large instances [Del11; DN12], making it a less versatile approach.

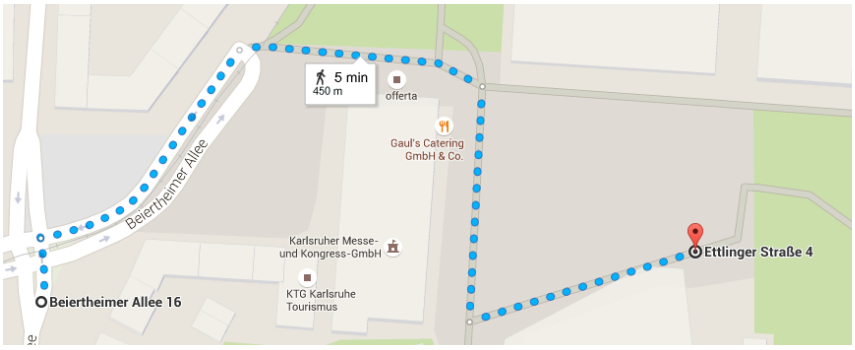
To summarize, we see that TDCRP clearly broadens the state-of-the-art of time-dependent route planning, handling a wider range of practical requirements (e. g., fast metric-dependent preprocessing, robustness to user preferences, live traffic) with a query performance close to the fastest known approaches.

Conclusions

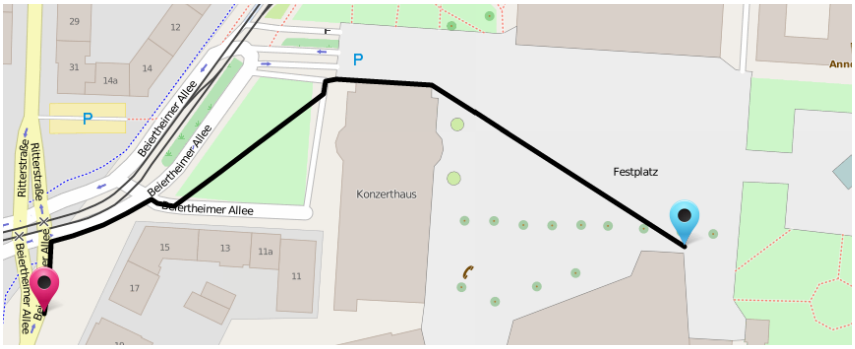
We introduced TDCRP, a separator-based approach for dynamic, time-dependent route planning. We showed that, unlike its closest competitor (A)TCH, TDCRP is robust against user-dependent metric changes, very much like CRP is more robust than CH. Most importantly, unlike scalar CRP, we have to deal with time-dependent shortcuts, and a strong increase in functional complexity on higher levels; To reduce memory consumption, we approximate the overlay arcs at each level, accelerating customization and query times. As a result, we obtain an approach that enables fast near-optimal, time-dependent queries, with quick integration of user preferences, live traffic, and traffic predictions.

There are several aspects of future work. First, functional complexity growth of time-dependent shortcuts is problematic, and from what we have seen, it is much stronger than the increase in the number of corresponding paths. This might explain why TDCALT is surprisingly competitive, so re-evaluation seems fruitful (e. g., exploiting insights from [EP13]). Revisiting hierarchical preprocessing techniques that are not based on shortcuts [Gut04; MCB14] could also be interesting. Additionally, we are interested in alternative customization approaches (avoiding expensive profile searches). This could be achieved, e. g., by using kinetic data structures [FHS14], or balanced contraction [BGSV13] within cells. While we customized time-dependent

overlay arcs with both historic travel time functions (changes seldom) and user preferences (changes often) at once, in practice, it might pay off to separate this into two further phases (yielding a 4-phase approach). One could also aim at exact queries based on approximated shortcuts as in ATCH. Finally, it would be interesting to re-evaluate (A)TCH in light of Customizable CH [DSW14; DSW16].



(a) Google Maps.



(b) Our approach.

Figure 3.13: In contrast to current approaches, our route in this example makes use of sidewalks (avoiding unnecessary street crossings), begins on a plaza and traverses it in a natural way.

3.3 PEDESTRIAN ROUTE PLANNING

In this section, we address the unique challenges that come with computing pedestrian routes. In order to obtain as realistic routes as possible, we propose to first augment the underlying street network model, and then to apply a tailored routing algorithm on top of it. After setting some basic definitions (Section 3.3.1), we propose geometric approaches for automatically adding sidewalks, calculating realistic crossing penalties for major roads, and preprocessing plazas and parks in order to traverse them in a natural way (Section 3.3.2). Our integrated routing algorithm seamlessly handles node-to-node queries and queries whose origin or destination is an arbitrary geographic location inside a plaza or park (Section 3.3.3). To efficiently support long-range queries, we also adapt the *Customizable Route Planning* (CRP) algorithm [DGPW15]—a well-known speed-up technique for computing driving directions in road networks—to our scenario. We evaluate our approach on OpenStreetMap data of Berlin and the state of Baden-Württemberg, Germany (Section 3.3.4). Our algorithm runs in the order of milliseconds, which is practical for interactive applications. We observe that we are able to compute pedestrian routes that are much more appealing than those by state-of-the-art route

planners, such as shown in Figure 3.13. Section 3.3.4.2 shows further examples and an illustrated comparison of our method with three popular external services.

3.3.1 *Preliminaries*

Throughout Section 3.3, we use the following notation and concepts:

We model the street network as a *undirected graph* $G = (V, E)$ with a set V of *nodes* and a set $E \subseteq \binom{V}{2}$ of *edges*. A node that is incident to exactly two edges is called a *2-node*. For a specific subset of edges $E' \subseteq E$ the *induced graph* $G[E'] = (V', E')$ contains E' and the nodes V', which are incident to the edges of E'. An s–t *path* in G is a node sequence $P_{s,t} = (s = v_1, \ldots, v_k = t)$, with each $e_i = \{v_i, v_{i+1}\}$ contained in E. A graph G is *planar* if a crossing-free drawing of G in the plane exists. A specific *embedding* of G maps each node to a coordinate in the plane. The embedding of G subdivides the plane into disjoint polygonal regions called *faces* bounded by the edges of G. Note that in our street networks each node $v \in V$ corresponds to a physical location. Likewise, each edge $e \in E$ represents a street segment. The *cost* of e is given by $c\colon E \to \mathbb{R}_+$, where $c(e)$ is the time (in seconds) a pedestrian requires to traverse e. This value may, e.g., depend on the street category and the segment's physical length. For source and target nodes s and t, Dijkstra's algorithm [Dij59] computes a *shortest* s–t *path* $P_{s,t}$, i.e., an s–t path whose cost $\sum_{i=1}^{k-1} c(e_i)$ is minimal.

Besides the street network, we consider the *walkable area* of public open spaces such as *plazas* and *parks*. We represent them by polygons, as follows. A *(simple) polygon* $Q \subset \mathbb{R}^2$ is defined as the interior of a sequence of *vertices* $Q = (p_1, \ldots, p_n), p_i \in \mathbb{R}^2$ sorted clockwise and connected by non-self-intersecting *segments* $\overline{p_1p_2}, \overline{p_2p_3}, \ldots, \overline{p_n, p_1}$. (We distinguish the *nodes* of a graph from the *vertices* of a polygon.) A *polygon with holes* Q is defined by a *boundary cycle* b_Q and *holes* $h_Q^1, \ldots, h_Q^k$ in the interior of b_Q, where b_Q and h_Q^i again define simple polygons and their vertices are the vertices of Q. The *interior* of Q is $b_Q \setminus (\cup_i h_Q^i)$ and a point o or a segment s is *within* Q if o or s lie within this interior. The *visibility graph* $VG(Q)$ of a polygon with holes Q is a geometric graph that consists of all vertices $p_1, \ldots, p_n$ of Q and all segments $\overline{p_ip_j}$ which lie within Q. The *visibility polygon* $VP(p, Q)$ of a point $p \in Q$ with respect to the containing polygon Q is the region within Q that is *visible* from p, i.e., for each $q \in VP(p, Q)$ the segment $\overline{pq} \subseteq Q$. With $|Q|$ we denote the number of vertices of Q.

For many geometric computations, we use functionality of the computational geometry library CGAL [Cgal15], in particular for computing line segment intersections, polygon unions and differences, visibility graphs and polygons, range queries, and point-in-polygon queries; see [BCKO08] for descriptions of the algorithms. To apply

these geometric algorithms to our street data, we map geographic coordinates to points in $\mathbb{R}^2$ using the Mercator projection. We use Euclidean distances $\|p - q\|$ between points p and q.

We implemented our own sweep line algorithm for a *batched point in polygon test*. Given a set of disjoint polygons (without holes) and a set of query points, our algorithm determines for each point the polygon that contains it (if any). The algorithm works by sweeping the query points and the end points of the polygon segments, which define our event points, from left to right, maintaining a self-balancing binary tree of the segments which intersect the current (vertical) sweep line. Whenever the current event point is a query point $o = (x, y)$, we find the two segments s_a and s_b, which lie vertically directly above and below o. If s_a and s_b belong to a polygon $Q = (p_1, \ldots, p_n)$ with leftmost vertex p_1, we check whether $s_a = \overline{p_i p_{i+1}}$, $s_b = \overline{p_j p_{j+1}}$, and $j > i$. If so, o lies within Q, otherwise it is not contained in any polygon. If m is the number of polygon edges and n is the number of points and polygon vertices then this algorithm requires $O(n \log m)$ time. For a set of polygons with holes we may use the same approach once for the boundaries and once for the holes.

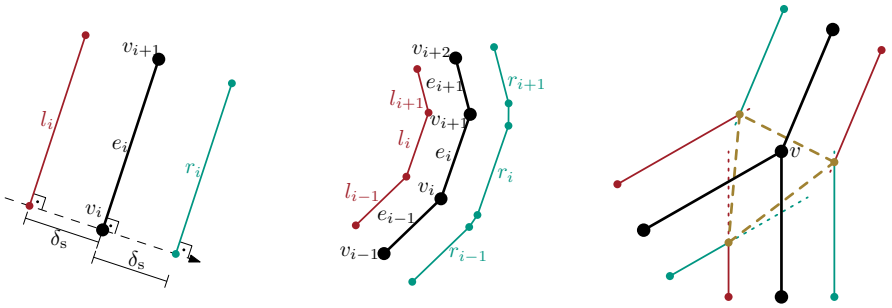
3.3.2 *Augmented Graph Model for Pedestrian Routing*

We consider three key aspects where pedestrian routes differ from those of vehicles: (a) sidewalks are preferred over streets, if present; (b) plazas can be traversed freely; (c) in parks pedestrians may walk freely on the lawn, but park walkways are preferred. In this section, we present algorithms that process the street network in order to accommodate these differences. (We then discuss queries in Section 3.3.3.) Most of this preprocessing is independent of the edge costs in the network, hence, new costs can be integrated with little effort.

3.3.2.1 *Sidewalks and Street Crossings*

Unlike features, such as street direction, turn restrictions and separation into lanes, sidewalk data is often lacking (or inconsistently modeled) in popular street databases, such as OpenStreetMap.[13] As a result, state-of-the art pedestrian route planners mostly use the streets themselves and not their sidewalks. However, this may lead to unnecessary street crossings, which can either be costly (due to traffic lights), or even be impossible. In contrast, when sidewalks are considered properly, a seeming detour may actually be the shorter path, see Figure 3.14. We therefore propose to replace some streets (as given by the input) with automatically generated sidewalks. We distinguishing between three street types: *Highways* represent streets that are inaccessible for pedestrians, hence, they have no sidewalks; *regular*

13 http://openstreetmap.org



(a) Sidewalks for single segment.

(b) Connecting sidewalk paths.

(c) Adjusting sidewalks at a street intersection.

Figure 3.15: Generating sidewalks. Regular street segments are replaced by two sidewalk edges (a). Subsequent pairs of sidewalk edges are then connected along each 2-node path (b). Finally, the resulting sidewalks are adjusted at street intersections (dotted parts removed), and crossing edges (dashed) are added (c).

streets, such as city streets, have sidewalks; and *walkways* are footpaths and streets small enough to require no sidewalks.

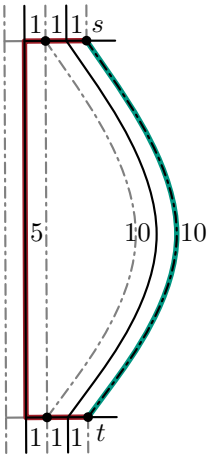


Figure 3.14: Using streets (left) or sidewalks (right) changes shortest path.

STREET POLYGONS. Naïvely, one could add sidewalks to the left and to the right of every regular street in the input [PV03]. Unfortunately, this results in sidewalks being placed in the middle of multilane streets or in median strips, which is clearly unwanted. We therefore propose to avoid areas enclosed by regular or highway streets that are too small or thin to hold sidewalks.

To achieve this, we first compute a set $\mathcal{S}$ of *street polygons*, representing such areas without sidewalks. Consider the embedded graph G_{hr} induced by the set of highway and regular street edges. We obtain the planarization G'_{hr} of G_{hr} using a standard sweep-line algorithm for line segment intersections [BCKO08]. Let f be a face in G'_{hr} and let a_f and p_f denote its area and perimeter, respectively. Then f is considered a street polygon, if $a_f/p_f \leqslant \beta_r$ (too thin) or $a_f \leqslant \beta_a$ (too small) for suitably chosen thresholds β_r, β_a.

SIDEWALKS. Our goal is to place sidewalks to the left and to the right of each street edge at some offset, unless they would be placed inside a street polygon. They should also follow curves and handle street intersections correctly, see Figure 3.15. To do so, we consider the embedded graph G_r, induced by the regular street edges. Recall that in G_r street intersections are modeled by nodes v of degree $\deg(v) \geqslant 3$, while the street's curvature is modeled as paths of 2-nodes. For each maximal 2-node path $(v_1, \ldots, v_k)$ and its adjacent intersection nodes v_0 and v_{k+1} (where we treat dead ends as intersection nodes, too), we consider the edge sequence $(e_0, \ldots, e_k)$, where $e_i = \{v_i, v_{i+1}\}$. For each edge e_i, we create two sidewalk edges l_i and r_i, and offset

them (from e_i) by a distance δ_s; see Figure 3.15a. In order to form correct paths along bends, these edges need to be trimmed or linked via auxiliary edges, depending on the bend angles; see Figure 3.15b.

At each street intersection $v \in G_r$ with $\deg(v) \geqslant 3$, we sort the incident edges in cyclic order. This order yields adjacent sidewalks, which we again trim at their respective intersection points or link by an auxiliary edge; see Figure 3.15c. For each street edge e incident to v, we also add an edge between the two sidewalks at v associated with e, which allows to cross e at v; see again Figure 3.15c.

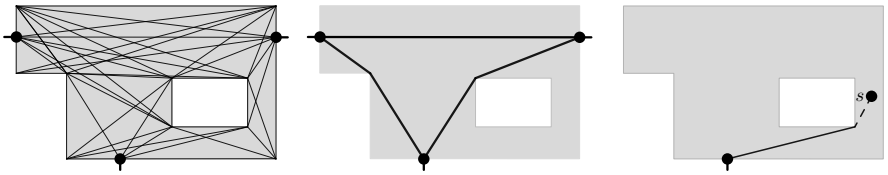
Next, we remove all sidewalk portions contained in street polygons of $\mathcal{S}$. Using a standard line segment intersection algorithm [BCKO08], we first subdivide sidewalks at the boundaries of street polygons. Then, we use our point-in-polygon algorithm (see Section 3.3.1) to remove all sidewalk segments with both endpoints inside a polygon of $\mathcal{S}$. This results in (at most) two sidewalks per street, as opposed to two sidewalks per lane.

Finally, we assemble the *routing graph* G induced by sidewalk, crossing and walkway edges (but not highway and regular street edges). For connectivity, we add nodes at the intersections of sidewalks with walkways, subdividing the intersecting edges, again by running a line segment intersection algorithm [BCKO08].

CROSSING PENALTIES. We may further utilize the street polygons $\mathcal{S}$ in order to penalize certain street crossings where waiting times can be expected. As the area covered by parallel street lanes is represented in $\mathcal{S}$, an edge e of G which passes through a multi-lane street also has a portion within $\mathcal{S}$, and we may penalize this portion in our cost function. We use two types of penalties. The "one-time" penalty α_e models a general waiting time, either for a pedestrian light or for traffic to clear. We add α_e to the cost of each edge that *enters* $\mathcal{S}$. More precisely, an edge $e = \{u, v\}$ in G enters $\mathcal{S}$ if u is outside $\mathcal{S}$ and the segment of e has common points with $\mathcal{S}$. The second penalty, denoted α_w, is a *penalty per unit of length* spent within $\mathcal{S}$. It reflects that wider streets generally require longer waiting times to cross. We find the edge portions of G within $\mathcal{S}$ while we remove sidewalks within $\mathcal{S}$. We use our sweep line algorithm (cf. Section 3.3.1) to find edges starting outside $\mathcal{S}$. Such edges with portions within $\mathcal{S}$ also enter the street polygons.

3.3.2.2 *Plazas*

Pedestrians may traverse plazas freely. However, somewhat surprisingly, most state-of-the-art pedestrian navigation services route around such walkable areas, not through them. We propose to utilize visibility graphs to remedy this shortcoming. We assume that the street network database provides traversable plazas as a set $\mathcal{P}$ of *plaza polygons*, possibly with holes due to obstacles. Given $\mathcal{P}$ and the previously obtained routing graph G, we compute the *entry nodes* of each plaza: These lie on



(a) A visibility graph. (b) Shortest path visibility edges. (c) Route origin in plaza polygon.

Figure 3.16: Small polygon Q with a hole and all visibility edges (a) and the ones that are also on shortest paths (b). Routing from within Q requires all visibility edges (c).

the intersection of a plaza polygon's boundary and the routing graph and are obtained by a line segment intersection algorithm [BCKO08]. We add each entry node both to the plaza polygon (as a vertex) and to the routing graph. For each polygon $Q \in \mathcal{P}$, we then compute the *visibility graph* $\mathrm{VG}(Q)$. If Q has no holes, we require quadratic time [Cgal15; OW88], otherwise cubic time. (Since we encounter only very few polygons with holes in practice, we did not implement a more efficient algorithm, such as [AW88].) Let $E_{\mathrm{vis}}(Q)$ be the visibility edges of $\mathrm{VG}(Q)$, and $E_{\mathrm{vis}} = \cup_{Q \in \mathcal{P}} E_{\mathrm{vis}}(Q)$. We add E_{vis} as further pedestrian edges to the routing graph G.

Since the number of visibility edges $E_{\mathrm{vis}}(Q)$ of a plaza polygon $Q \in \mathcal{P}$ is generally quite high (see Figure 3.16a), routing through plazas can become expensive. We therefore mark the subset $E^{\mathrm{sp}}_{\mathrm{vis}} \subset E_{\mathrm{vis}}$ of visibility edges that are part of shortest paths between any pair of entry nodes (the query may then ignore unmarked edges); see Figure 3.16b. We do so by running Dijkstra's algorithm from each entry node, only relaxing visibility edges of the node's plaza. Note that $E^{\mathrm{sp}}_{\mathrm{vis}}$ suffices to route *across* plazas, but queries that begin or end on a plaza may still require all edges in E_{vis}; see Figure 3.16c and Section 3.3.3. Also note that computing $E^{\mathrm{sp}}_{\mathrm{vis}}$ requires knowledge of the routing cost function (all other preprocessing does not). However, since the necessary shortest path queries are restricted to each plaza and the number of entry nodes is typically small, this step is not costly compared to the total preprocessing effort.

3.3.2.3 *Parks*

Unlike plazas, parks have designated walkways, which we favor by routing on walkable park areas (such as lawn) only at the beginning or end of a route. In order to quickly locate nearby walkways during queries, we precompute the faces of a park induced by its walkways.

Similarly to plazas, we assume that the walkable area of parks is given as the set $\mathcal{L}$ of *park polygons* (possibly with holes) by the street network database. We compute the entry nodes the same way we do for plazas. We then use our algorithm from Section 3.3.1 to

compute the set E_L of edges in G contained in each $L \in \mathcal{L}$ (in a single sweep). Thus, $G_L = G[E_L]$ contains exactly the park walkways within L. We add the boundary of L to G_L (as nodes and edges) and planarize G_L. We define the set of *park faces* F_L to be the faces of G_L, and $\mathcal{F} = \bigcup_{L \in \mathcal{L}} F_L$. During queries, we will use $\mathcal{F}$ for locating park walkways and routing to/from them (see Section 3.3.3).

3.3.3 *Computing Pedestrian Routes*

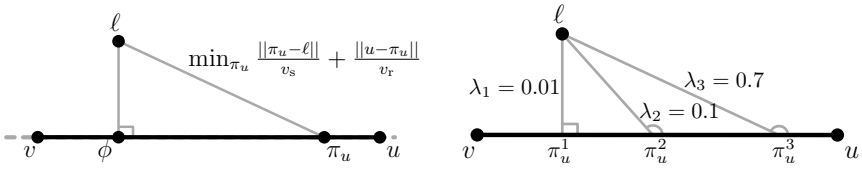
We now discuss how we leverage our model from Section 3.3.2 to compute realistic pedestrian routes. We are generally interested in queries between arbitrary locations ℓ_o (origin) and ℓ_d (destination). Usually, one handles such *location-to-location queries* by first mapping the locations to their nearest nodes (or edges) of the network, and then invoking a shortest path algorithm between those. However, for locations inside plazas and parks this method would result in inaccurate routes. Instead, we propose the following approach. First, we test whether $\ell \in \{\ell_o, \ell_d\}$ is located inside a plaza or a park. In either case, we first connect ℓ to G with sensible edges and then run Dijkstra's algorithm between ℓ_o and ℓ_d on this augmented graph. If neither is the case, we just find the nearest nodes in G using a k-d tree [Ben75], as in the classic scenario. We discuss more details next.

PLAZAS. To test whether the origin or destination location ℓ is on a plaza, we simply perform a point-in-polygon test [BCKO08]. Now, assume that $Q \in \mathcal{P}$ is the polygon, which contains ℓ. We compute the visibility polygon $VP(\ell, Q)$ of ℓ with respect to Q by applying the recent algorithm of Bungiu et al. [BHH+14]. We also use our sweep line algorithm from Section 3.3.1 to obtain the nodes V_Q^ℓ in $VG(Q) \subset G$ that are located within $VP(\ell, Q)$. We then simply connect ℓ to each node $p \in V_Q^\ell$ by adding edges $\{\ell, p\}$ to the graph G.

Recall from Section 3.3.2.2 that to route across plazas, the visibility edges in E_{vis}^{sp} suffice. Hence, we ignore edges $e \in E_{vis} \setminus E_{vis}^{sp}$ during the query, unless $e \in VG(Q)$ for the polygon Q containing ℓ, in which case it is required for correctness.

PARKS. For the case that ℓ is contained in a park, we first obtain the enclosing park face f (similarly to the plaza case). We now consider two different walking speeds: the regular walking speed v_r, and another (slower) one v_s for park faces (e. g., lawn). We set $\lambda = v_s/v_r$ (with $\lambda \in (0, 1]$) as a query time parameter; values $\lambda < 1$ penalize walking on the lawn, with smaller λ values leading to higher penalization.

Taking this into account, our goal is to connect ℓ to the walkways of f, such that the total walking duration is minimized. We thereby compute the optimal path toward each edge $e = \{u, v\} \in f$ separately,



(a) Reaching park walkways via park area.

(b) Different penalties for walking on park area.

Figure 3.17: Using the park area and walkway. We minimize the walking time on the park area plus that on the walkway (left). The manner in which the park area is utilized varies with v_s (right).

as follows. Consider a point π_u on e. To reach u from ℓ via π_u, one requires total walking time $w = \frac{||\pi_u-\ell||}{v_s} + \frac{||u-\pi_u||}{v_r}$; see Figure 3.17a. For a given λ, the minimum walking time w^* is achieved by the *projection point* $\pi_u^* = \phi + \frac{\lambda}{1-\lambda^2} \cdot \frac{||\ell-\phi||}{||u-\phi||} \cdot (u-\phi)$, where ϕ is the perpendicular projection of ℓ on the line through e; see [And12] for a derivation of this formula. As seen in Figure 3.17b, a small value of λ causes a perpendicular projection: walking on the lawn is costly and therefore minimized. A larger value of λ allows for a more direct, target-aimed projection, saving distance but using more of the walkable park area.

We now use the aforementioned formula to compute for each edge $e = \{u, v\} \in f$ the projection points π_u^* and π_v^*. To check whether a segments $s_u = \overline{\pi_u^* \ell}$ is walkable within the park, we test whether the point π_u^* lies within the visibility polygon $\mathrm{VP}(\ell, Q)$. If so, we add the edge $\{\ell, u\}$ with cost $\frac{||\pi_u^*-\ell||}{v_s} + \frac{||u-\pi_u^*||}{v_r}$ to G. Node v is handled analogously.

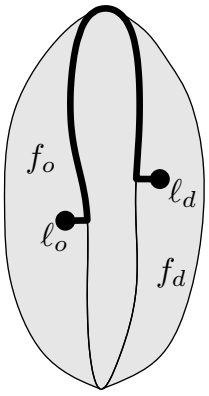


Figure 3.18: Detour due to long thin face.

Note that since we directly connect the origin ℓ_o and destination ℓ_d to the edges of their enclosing faces, we are unable to route around obstacles in parks. Moreover, we are unable to walk across other park faces (except the ones containing the origin and destination locations). However, this may result in unnatural routes, if origin and destination are in the same park separated by a thin face; see Figure 3.18. We solve this issue by introducing a radius parameter ϵ, and additionally compute edges to the boundaries of all faces (of the same park) that have vertices within distance ϵ of ℓ. We use range queries [Cgal15] to obtain those faces. If, both, origin ℓ_o and destination ℓ_d are in the same park and within distance ϵ, we additionally consider the direct route $\overline{\ell_o \ell_d}$ with cost $\frac{||\ell_d-\ell_o||}{v_s}$ explicitly.

CUSTOMIZABLE ROUTE PLANNING. Typical pedestrian routes are very short, thus, one might argue that Dijkstra's algorithm computes them sufficiently fast. Still, a practical routing engine should be robust against long-distance queries as well. We therefore propose to make use of the *Customizable Route Planning* (CRP) algorithm [DGPW15]. It is a state-of-the-art speedup technique, developed for computing

driving directions in road networks. CRP employs three phases: The *preprocessing phase* uses a nested multilevel partition to compute (for each level) a metric-independent overlay graph over the boundary nodes of the partition. The *customization phase* takes a cost function as input and computes the actual edge weights of the overlay graph. Finally, the *query phase* runs bidirectional Dijkstra's algorithm, using the overlay graph to the effect of "skipping" over large parts of the network. See [DGPW15] for details.

Adapting CRP to our scenario requires little effort. We use the routing graph G for computing both the multilevel partition and the overlay graph. To easily support queries beginning or ending within parks or plazas, we enforce that nodes within the same park or plaza are never put into different cells of the partition. (We do this by running the partitioner on a slightly modified graph, in which we contract all nodes associated with the same park or plaza.) To see why this is correct, recall that the temporary edges added by the query only point to nodes within the park or plaza which contains the origin (or destination) location. By construction these nodes are all part of the same cell (on every level of the partition), therefore, the distances in the overlay graph are unaffected and still correct.

Note that in our CRP query we do not bother ignoring visibility edges in G that are not on shortest paths: They are only present on the bottom level, therefore, the query skips over them automatically in most cases.

3.3.4 *Experiments*

We implemented all algorithms in C++ using g++ 4.8.3 (flag -O3) and CGAL 4.6. We conducted our experiments on a single core of a 4-core Intel Xeon E5-1630v3 CPU clocked at 3.7 GHz with 128 GiB of DDR4-2133 RAM. Our data set was extracted from OpenStreetMap (OSM) on May 15, 2015, and includes roads, plazas and parks.[14] We use two instances: Berlin (BE) and the state of Baden-Württemberg (BW), both in Germany. While BE is an eclectic city with plenty of large streets, parks and plazas (making it interesting for evaluating pedestrian routes), we use BW to demonstrate the scalability of our approach.

We first present a quantitative evaluation of our approach, then compare the quality of our routes to the state of the art in a case study.

3.3.4.1 *Quantitative Evaluation*

We determined sensible values for the parameters of our preprocessing (cf. Section 3.3.2) by running preliminary experiments. We set the

14 Note that OSM offers a tag for indicating availability of sidewalks at streets, however, it has not been widely adopted as of now, cf. http://taginfo.openstreetmap.org/keys/?key=sidewalk.

Table 3.31: Size figures before and after preprocessing. Besides graph size, we report the total number of vertices for plaza, park, and obstacle polygons. Preprocessing time is given in [m:s].

	OSM Input					Pedestrian Output				
	Nodes	Edges	Plaza	Park	Obst.	Nodes	Edges	Plaza	Park	Time
BE	378 298	890 682	9 727	33 072	1 116	452 586	1 132 928	7 276	19 903	1:26
BW	8 235 762	17 740 940	74 547	86 380	4 439	10 209 641	22 750 644	63 300	43 632	32:45

Table 3.32: Detailed preprocessing figures. Besides running time, we report the number of added sidewalks, substituted streets, avg. vertices per plaza polygon (Plaza avg.), visibility edges (Vis.) and the fraction of them on shortest paths (SP. [%]), avg. vertices per park (Park avg.), avg. faces per park (Faces/park), avg. vertices per park face (Face avg.), and the vertices of all park faces (Faces total).

	Sidewalks			Plazas				Parks				
	Added sidewalks	Subst. streets	Time [s]	Plaza avg.	Vis. total	SP. [%]	Time [s]	Park avg.	Faces/ park	Face avg.	Faces total	Time [s]
BE	266 336	105 146	32.6	15.38	86 912	9.5	23.0	22.4	10.4	10.68	98 108	31.6
BW	5 580 842	1 824 185	743.7	17.45	772 416	7.7	563.6	20.8	6.6	11.26	155 840	657.4

sidewalk offset to $\delta_s = 3\,\text{m}$, and set values for sidewalks suppression of small and thin street polygons to $\beta_a = 1000\,\text{m}^2$ and $\beta_r = 3.17\,\text{m}$. For queries we assume a regular walking speed of $v_r = 1.4\,\frac{\text{m}}{\text{s}}$ [BBHK06], and we set $v_s = 0.9\,\frac{\text{m}}{\text{s}}$ for walkable park areas, i. e., $\lambda \approx 0.6$. We also set the park face expansion value to $\epsilon = 20\,\text{m}$. Regarding intersections, we set the crossing penalties to $\alpha_e = 10\,\text{s}$ and $\alpha_w = 1\,\frac{\text{s}}{\text{m}}$, which leads to about 30 s of expected waiting time for typically-sized intersections.

Note that though we set these parameter values uniformly for our experiments, the approach would easily allow setting specific values per intersection or park face, if such detailed data was available. Also note that in our instances we do not add crossing edges within street polygons, i. e., at large multi-lane intersections (cf. Section 3.3.2). In fact, OpenStreetMap provides these already, and adding further crossings may result in dangerous paths, forcing the pedestrian to cross several lanes without the aid of traffic regulations.

PREPROCESSING. Table 3.31 presents size figures for the input and output of our preprocessing. Note that BW is significantly larger than BE (factor of 20 in graph size and factor of 9 in plaza polygons). This is reflected by the preprocessing effort, which takes about 23 times longer on BW. However, the graph size increases by less

than 30 % (nodes and edges) by our preprocessing. Unfortunately, polygons representing walkable areas (parks and plazas) in OSM may overlap and, moreover, polygons with holes are not supported. Instead, obstacles are represented as an additional type of polygon. We therefore first compute the union of overlapping polygons and then subtract potential obstacles from it [Cgal15]. This explains the (somewhat peculiar) drop of 50 % in the number of park polygon vertices in our output. Note that only less than 3 % of the resulting plaza polygons have holes in them (not reported in the table).

Table 3.32 presents more detailed figures. We observe that each part of our preprocessing requires a similar amount of time. Regarding sidewalks, only a small subset (12 %) of the roads is actually substituted. (Recall that we replace neither highways nor walkways.) However, the number of sidewalk edges per substituted road segment is more than two on average, due to complex intersections and other effects (cf. Section 3.3.2). Regarding plazas, we observe that the number of visibility edges is only a small fraction of the graph (less than 10 %), with less than 10 % of those actually being on shortest paths. The necessary shortest path computations take less than 3 seconds on BW (not reported in the table). For parks, we observe that including walkways (to compute park faces) increases the number of park vertices by a factor of 5 ("Faces total" in the table). While this results in a high average number of vertices per entire park (111 for BE), the number of vertices per park face remains small, which is the influential performance figure for queries that begin or end in a park.

QUERIES. We now evaluate the query performance. Recall that our query algorithm takes as input two arbitrary locations, which may be inside a plaza or park, and in which case the query will route from the precise location to the vertices of its surrounding polygon. Table 3.33 separately evaluates our algorithm for each scenario of placing the origin or destination on a street node (s), inside a plaza (p), or a park face (f). Per scenario, we generated 1,000 queries, choosing origin and destination (i.e., node, plaza polygon or park face) uniformly at random. For the plaza or park case, we further chose an interior point at random.

The query is oblivious to the specific scenario, i.e., we only pass geographic locations as input, and it needs to perform the necessary checks to figure out the right scenario itself. However, at below 80 µs these checks (including the determination of the specific street node or enclosing polygon) take negligible time. The initialization stage for plazas (computing additional visibility edges) or parks (computing and testing projections) is considerably more expensive, but still runs well below a millisecond, orders of magnitude faster than the subsequent run of Dijkstra's algorithm. Note that in our implementation we never add any edges explicitly (cf. Section 3.3.3), but rather simply

Table 3.33: Evaluating the query performance of our approach. We distinguish each combination of the origin/destination being on a street node (s), plaza polygon (p), or park face (f). We report the time in milliseconds to check for each of these cases (Localization), the time for our initialization stage (Initialization) or not applicable (—), and the time for running Dijkstra's algorithm (Dij.).

	BE						BW					
	Localization			Initialization		Dij.	Localization			Initialization		Dij.
Query	Plaza	Park	Street	Plaza	Park	[ms]	Plaza	Park	Street	Plaza	Park	[ms]
s–s	0.021	0.027	0.004	—	—	31.8	0.033	0.040	0.005	—	—	808.0
s–p	0.021	0.016	0.002	0.165	—	30.4	0.032	0.020	0.003	0.173	—	871.6
p–p	0.016	—	—	0.264	—	27.9	0.027	—	—	0.351	—	889.4
s–f	0.021	0.022	0.002	—	0.359	28.3	0.029	0.026	0.002	—	0.310	758.7
p–f	0.017	0.011	—	0.145	0.362	30.6	0.027	0.014	—	0.178	0.303	810.1
f–f	0.020	0.021	—	—	0.733	27.6	0.029	0.027	—	—	0.622	733.6

initialize Dijkstra's algorithm with all vertices (and their respective distances) to which these temporary edges would point.

CUSTOMIZABLE ROUTE PLANNING. We finally evaluate the combination of our query algorithm with the Customizable Route Planning (CRP) approach [DGPW15] on our larger BW network. For partitioning, we use PUNCH [DGRW11] set to compute five nested levels with at most $[2^8, 2^{11}, 2^{14}, 2^{17}, 2^{20}]$ vertices per cell. (This is the same configuration as in [DGPW15].) We compute the partition on the routing graph (that is output by our preprocessing), however, we temporarily replace nodes of the same plaza or park by a single supernode. This keeps polygons from spreading over cell boundaries and simplifies the CRP query. Computing the metric-independent partition takes several minutes and the subsequent customization phase takes about five seconds. Note that to integrate a new cost function, e.g., due to different crossing penalties, only the customization phase has to be rerun, which is very fast.

Figure 3.19 compares the performance of CRP with Dijkstra's algorithm using the *Dijkstra rank* methodology [SS05]: When running Dijkstra's algorithm from node s, node u has *rank* x, if it is the x-th node taken from the priority queue. By selecting random origin and destination pairs according to ranks $2^1, 2^2, \ldots, 2^{\lfloor \log |V_r| \rfloor}$ (we select 1,000 queries per bucket), the plot simultaneously captures short- mid- and long-range queries. We observe that for short-range queries the performance of Dijkstra's algorithm is very similar to that of CRP (below 200 µs on average). However, from rank 2^{10} onward, Dijkstra's algorithm becomes significantly slower (rising to more than a second),

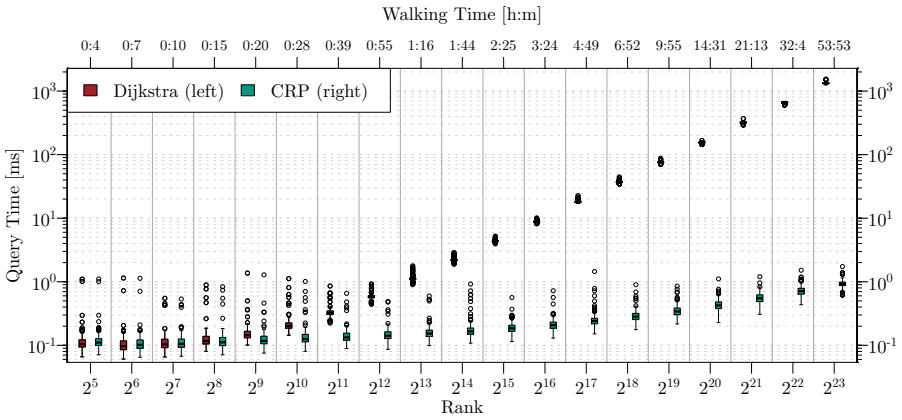


Figure 3.19: Dijkstra rank plot on our BW instance, comparing the performance of Dijkstra's algorithm with CRP. The top axis shows the average walking time for the queries in each bucket.

while the average running time of CRP remains below 1 ms at any rank. Note that while most pedestrian queries are likely of short range, a production system must nevertheless be robust against any query.

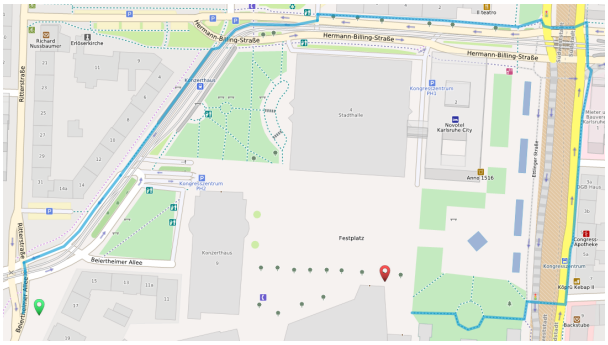
3.3.4.2 *Case Study*

We now present a case study, which compares the output of our approach to OpenRouteService[15], Google Maps[16] and Nokia HERE[17].

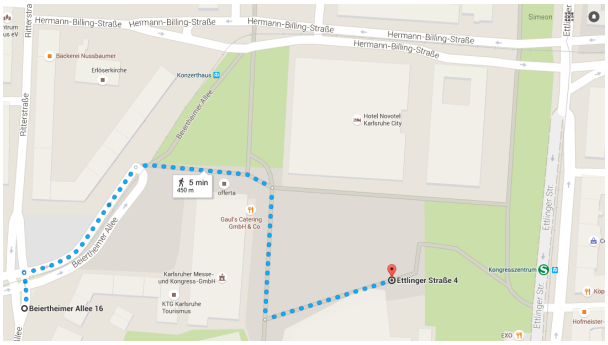
Figure 3.20 shows an example in the city of Karlsruhe, Germany. It highlights the importance of, both, the presence of sidewalks and being able to route across walkable areas. Clearly, OpenRouteService has the worst result, as it does not consider the boundary of the plaza (Festplatz) for routing, which results in a large detour. Because of improper sidewalk data, the routes of Google Maps and HERE suggest to go across the same street (Beiertheimer Allee) twice, which is unnatural and unnecessary. While Nokia HERE is the only competing approach that has some additional edges for walking across open areas (thus yielding a more realistic route), the utilization of these edges seems to be heuristic, still yielding an (unnatural) detour. In contrast, our route has no unnecessary street crossings (because of our generated sidewalk data), and the plaza is traversed in a natural way.

Figure 3.21 shows an example of a route starting on a plaza between buildings and ending in a park (Berlin, Germany). Unlike the previous example, OpenRouteService is able to route around (but not across) the plaza, because the plaza's boundary has been tagged as walkable. On the other hand, GoogleMaps seems to lack information in that region and so maps the query locations to the nearest street network node (which is actually blocked by the building structure). HERE

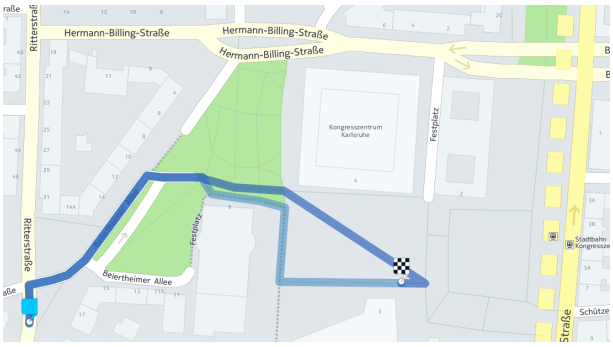
15 http://www.openrouteservice.org/
16 https://maps.google.de/
17 https://www.here.com/



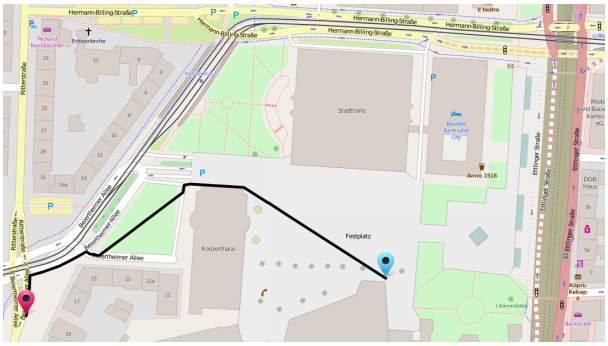
(a) OpenRouteService.



(b) Google Maps.



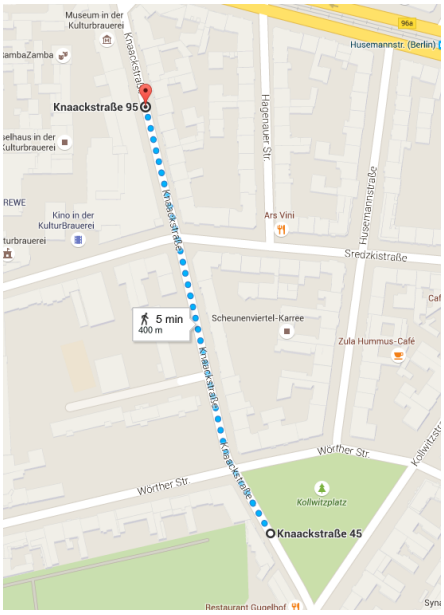
(c) Nokia's HERE.



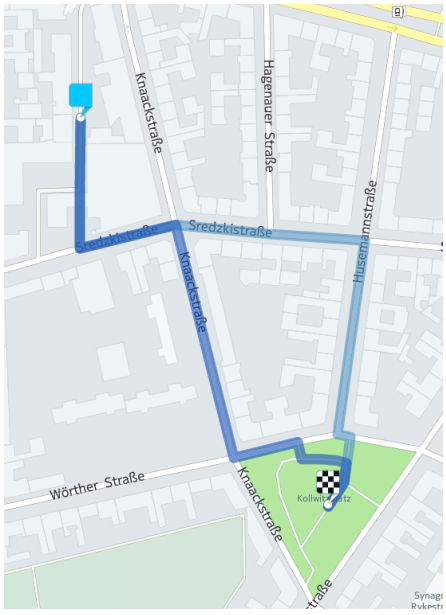
(d) Our approach.

Figure 3.20: Comparison with several readily available pedestrian route planning services. The origin of the route is a street address, and its destination is inside a plaza.

(a) OpenRouteService.



(b) Google Maps.



(c) Nokia's HERE.

(d) Our approach.

Figure 3.21: Comparison with several readily available pedestrian route planning services. The route begins in a plaza and ends in a park.

has walkways on the plaza, but also uses a shortcut which passes through a cinema; it yields a shorter path but is obscure and unlikely: guiding the pedestrian to a door is puzzling, the building may be closed, etc. As before, the route of our approach traverses the plaza without detours.

Towards the destination, the routes of OpenRouteService, Google Maps and HERE are all incomplete: they find the nearest node and simply use it as the query target. In contrast, our approach allows walking directly across the lawn and avoids the small detours introduced by the other approaches.

Conclusions

We presented an approach for quickly computing realistic pedestrian routes. We proposed geometric algorithms to automatically augment the street network with sensible sidewalks and edges in plazas, making it possible to walk across them in a natural way. Our query algorithm extends classic node-to-node queries by allowing the origin or destination to be an arbitrary location inside a park or plaza. We also combined our algorithm with the well-known Customizable Route Planning technique, which enabled us to compute appealing pedestrian routes within milliseconds, fast enough for interactive applications.

Future work includes more realistic models (e.g., for traffic lights or more precise human walking behavior); leveraging of building layouts [BPS11]; and additional optimization criteria like elevation and stairs, which have been used in the context of bicycle routing [Sto12]. We would also be interested in using our sidewalk generation algorithm in a semi-automatic tool for adding sidewalk data back to OpenStreetMap.

4

PUBLIC TRANSIT JOURNEY PLANNING

Research on journey planning in public transit networks is still very active, see the discussion in Chapter 2. While many publications opt to ignore traffic and rush hours (with a significant but bearable error on the calculated travel time) for road networks, transit networks are inherently time-dependent due to the underlying timetable. Ignoring the schedule may lead to journeys that have infeasible transfers between vehicles, which most likely results in getting the user stranded.

To capture the time-dependency, early approaches suggested different encodings of the schedule in a graph representation of the transit network [DKP12; DPW09b; PSWZ08]. One of these, the *time-expanded model (TE)*, expands time in the sense that it creates a vertex for each *time event* in the timetable (such as a vehicle departing or arriving at a stop), inserting arcs to connect events in the sequence they are served by a vehicle.

Newer approaches, like RAPTOR [DPW12a; DPW14], moved from graph representations to applying dynamic programming directly on the timetable. Very recently, however, graph representations were again considered by a trip-based approach [Wit15].

One of the degrees of freedom in public transit modeling is the way periodicity is handled. Undoubtedly, there are many recurring events in a timetable (e. g., small bus operators typically have only three schedules throughout the year: for Monday through Friday, for Saturday, and for Sunday). For larger-scale networks, however, periodicity is seldom perfect (as witnessed by the long list of exceptions in the footer of the station timetable of, e. g., the German railways). Aperiodic timetables therefore consider each singular day in the calendar as it is scheduled to happen.

In this chapter, we revisit the time-expanded graph model. For aperiodic timetables, the TE graph is directed and acyclic, a property that we exploit in this chapter. In particular, we find that by topologically sorting (in the most straightforward way) the timetable connections represented in the TE graph, we can completely avoid explicitly building the TE graph to begin with. In the second part of this chapter, these properties (directed, acyclic) enable the transfer of research results on fast reachability queries.

CHAPTER OUTLINE. Section 4.1 introduces the Connection Scan Algorithm (CSA), a novel algorithmic framework to compute journeys directly on the timetable. It organizes data as a single array of connections, which it scans once per query, enabling fast earliest arrival

and multicriteria profile search. Section 4.2 introduces Public Transit Labeling (PTL), a preprocessing approach that provides simple and efficient algorithms for earliest arrival, profile, and multicriteria queries that are orders of magnitude faster than the state of the art.

4.1 CONNECTION SCAN ALGORITHM

In this section, we present the *Connection Scan Algorithm* (CSA). In its basic variant, it solves the earliest arrival problem, and is, like RAPTOR, not graph-based. It is centered around elementary *connections*, which are the most basic building block of a timetable. CSA organizes them as one single array, which it then scans once (linearly) to compute journeys to all stops of the network. The algorithm turns out to be intriguingly simple with excellent spatial data locality.

The rest of this section is organized as follows. Section 4.1.1 sets necessary notion, and Section 4.1.2 presents our new algorithm. Section 4.1.3 extends it to multicriteria profile queries. The experimental evaluation is available in Section 4.1.4.

4.1.1 *Preliminaries*

Throughout Section 4.1, we use the following notation and concepts:

Our public transit networks are defined in terms of their aperiodic *timetable*, consisting of a set of *stops*, a set of *connections*, and a set of *footpaths*. A *stop* p corresponds to a location in the network where a passenger can enter or exit a vehicle (such as a bus stop or train station). Stops may have associated minimum transfer times, denoted $\tau_{ch}(p)$, which represent the minimum time required to transfer between vehicles at the same stop p. A *connection* c models a vehicle departing at a stop $p_{dep}(c)$ at time $\tau_{dep}(c)$ and arriving at stop $p_{arr}(c)$ at time $\tau_{arr}(c)$ without intermediate halt. Connections that are subsequently operated by the same vehicle are grouped into *trips*. We identify them by $t(c)$. We denote by c_{next} the next connection (after c) of the same trip, if available. Trips can be further grouped into *routes*. A route is a set of trips serving the exact same sequence of stops. For correctness, we require trips of the same route to not overtake each other. *Footpaths* enable walking transfers between nearby stops. Each footpath consists of two stops with an associated walking duration. Note that our footpaths are transitively closed. A *journey* is a sequence of connections and footpaths. If two subsequent connections are not part of the same trip, their arrival-departure time-difference must be at least the minimum transfer time of the stop. Because our footpaths are transitively closed, a journey never contains two subsequent footpaths.

In the following, we consider several well-known problems. In the *earliest arrival problem* we are given a source stop p_s, a target stop p_t, and a departure time τ. It asks for a journey that departs from p_s

no earlier than τ and arrives at p_t as early as possible. The *profile problem* asks for the set of all earliest arrival journeys (from p_s to p_t) for every departure at p_s. Besides arrival time, we also consider the number of transfers as criterion: In multicriteria scenarios one is interested in computing a *Pareto set* of nondominated journeys. Here, a journey J_1 *dominates* a journey J_2 if it is better with respect to every criterion. Nondominated journeys are also called to be *Pareto-optimal*. Finally, the *multicriteria profile problem* requests a set of Pareto-optimal journeys (from p_s to p_t) for all departures (at p_s).

Usually, these problems have been solved by (variants of) Dijkstra's algorithm on an appropriate graph (representing the timetable). Most relevant for us is the realistic *time-expanded model* [PSWZ08]. It expands time in the sense that it creates a vertex for each *event* in the timetable (such as a vehicle departing or arriving at a stop). Then, for every connection it inserts an arc between its respective departure/arrival events, and also arcs that link subsequent connections. Arcs are always weighted by the time difference of their linked events. Special vertices may be added to respect minimum transfer times at stops. See [MSWZ07; PSWZ08] for details.

4.1.2 *Earliest Arrival Queries*

We now introduce the Connection Scan Algorithm (CSA), our approach to public transit route planning. We describe it for the earliest arrival problem and extend it to more complex scenarios in Sections 4.1.3. Our algorithm builds on the following property of public transit networks: We call a connection c *reachable* iff either the user is already traveling on a preceding connection of the same trip $t(c)$, or, he is standing at the connection's departure stop $p_{dep}(c)$ on time, i. e., before $\tau_{dep}(c)$. In fact, the time-expanded approach encodes this property into a graph G, and then uses Dijkstra's algorithm to obtain optimal sequences of reachable connections [PSWZ08]. Unfortunately, Dijkstra's performance is affected by many priority queue operations and suboptimal memory access patterns. However, since our timetables are aperiodic, we observe that G is acyclic. Thus, its arcs may be sorted topologically, e. g., by departure time. Dijkstra's algorithm on G, actually, scans (a subsequence of) them in this order.

Instead of building a graph, our algorithm assembles the timetable's connections into a single array C, sorted by departure time. Given source stop p_s and departure time τ as input, it maintains for each stop p a label $\tau(p)$ representing the earliest arrival time at p. Labels $\tau(\cdot)$ are initialized to all-infinity, except $\tau(p_s)$, which is set to τ. The algorithm scans all connections $c \in C$ (in order), testing if c can be *reached*. If this is the case and if $\tau_{arr}(c)$ improves $\tau(p_{arr}(c))$, CSA *relaxes* c by updating $\tau(p_{arr}(c))$. After scanning the full array, the labels $\tau(\cdot)$ provably hold earliest arrival times for all stops.

REACHABILITY, MINIMUM TRANSFER TIMES AND FOOTPATHS. To account for minimum transfer times in our data, we check a connection c for reachability by testing if $\tau(p_{dep}(c)) + \tau_{ch}(p_{dep}(c)) \leqslant \tau_{dep}(c)$ holds. Additionally, we track whether a preceding connection of the same trip $t(c)$ has been used. We, therefore, maintain for each connection a flag, initially set to 0. Whenever the algorithm identifies a connection c as reachable, it sets the flag of c's subsequent connection c_{next} to 1. Note that for networks with $\tau_{ch}(\cdot) = 0$, trip tracking can be disabled and testing reachability simplifies to $\tau(p_{dep}(c)) \leqslant \tau_{dep}(c)$. To handle footpaths, each time the algorithm relaxes a connection c, it scans all outgoing footpaths of $p_{arr}(c)$.

IMPROVEMENTS. Clearly, connections departing before time τ can never be reached and need not be scanned. We do a binary search on C to identify the first relevant connection and start scanning from there (*start criterion*). If we are only interested in *one-to-one queries*, the algorithm may stop as soon as it scans a connection whose departure time exceeds the target stop's earliest arrival time. Also, as soon as one connection of a trip is reachable, so are all subsequent connections of the same trip (and preceding connections of the trip have already been scanned). We may, therefore, keep a flag (indicating reachability) per trip (instead of per connection). The algorithm then operates on these *trip flags* instead. Note that we store all data sequentially in memory, making the scan extremely cache-efficient. Only accesses to stop labels and trip flags are potentially costly, but the number of stops and trips is small in comparison. To further improve spatial locality, we subtract from each connection $c \in C$ the minimum transfer time of $p_{dep}(c)$ from $\tau_{dep}(c)$, but keep the original ordering of C. Hence, CSA requires random access only on small parts of its data, which mostly fits in low-level cache.

4.1.3 *Profile and Multicriteria Queries*

CSA can be extended to profile queries. Given the timetable and a source stop p_s, a profile query computes for every stop p the set of all earliest arrival journeys to p for every departure from p_s, discarding dominated journeys. Such queries are useful for preprocessing techniques, but also for users with flexible departure (or arrival) time. We refer to the solution as a Pareto set of $(\tau_{dep}(p_s), \tau_{arr}(p_t))$ pairs.

In the following, we describe the *reverse* p–p_t-profile query. The forward search works analogously. Our algorithm, pCSA (p for profile), scans once over the array of connections sorted by *decreasing* departure time. For every stop it keeps a partial (tentative) profile. It maintains the property that the partial profiles are correct wrt. the subset of already scanned connections. Every stop is initialized with an empty profile, except p_t, which is set to a constant identity-profile. When

scanning a connection c, pCSA *evaluates* the partial profile at the arrival stop $p_{arr}(c)$: It asks for the earliest arrival time τ^* at p_t over all journeys departing at $p_{arr}(c)$ at $\tau_{arr}(c)$ or later. It then *updates* the profile at $p_{dep}(c)$ by potentially adding the pair $(\tau_{dep}(c), \tau^*)$ to it, discarding newly dominated pairs, if necessary.

MAINTAINING PROFILES. We describe two variants of maintaining profiles. The first, pCSA-P (P for Pareto), stores them as arrays of Pareto-optimal (τ_{dep}, τ_{arr}) pairs ordered by decreasing arrival (departure) time. Since new candidate entries are generated in order of decreasing departure time, profile updates are a constant-time operation: A candidate entry is either dominated by the last entry or is appended to the array. Profile evaluation is implemented as a linear scan over the array. This is quick in practice, since, compared to the timetable's period, connections usually have a short duration. The identity profile of p_t is handled as a special case. By slightly modifying the data structure, we obtain pCSA-C (C for constant), for which evaluation is also possible in constant time: When updating a profile, pCSA may append a candidate entry, even if it is dominated. To ensure correctness, we set the candidate's arrival time τ^* to that of the dominating entry. We then observe that, independent of the input's source or target stop, profile entries are always generated in the same order. Moreover, each connection is associated with only two such entries, one at its departure stop, relevant for updating, and, one at its arrival stop, relevant for evaluation. For each connection, we precompute *profile indices* pointing to these two entries, keeping them with the connection. Furthermore, its associated departure time and stop may be dropped. Note that the space consumption for keeping all (even suboptimal) profile entries is bounded by the number of connections. Following [DKP12], we also collect—in a quick preprocessing step—at each stop all arrival times (in decreasing order). Then, instead of storing arrival times in the profile entries, we keep *arrival time indices*. For our scenarios, these can be encoded using 16 (or fewer) bits. We call this technique *time indexing*, and the corresponding algorithm pCSA-CT.

MINIMUM TRANSFER TIMES AND FOOTPATHS. We incorporate minimum transfer times by evaluating the profile at a stop p for time τ at $\tau + \tau_{ch}(p)$. The trip bit is replaced by a trip arrival time, which represents the earliest arrival time at p_t when continuing with the trip. When scanning a connection c, we take the minimum of the trip arrival time and the evaluated profile at $p_{arr}(c)$. We update the trip arrival time and the profile at $p_{dep}(c)$, accordingly. *Footpaths* are handled as follows. Whenever a connection c is relaxed, we scan all incoming footpaths at $p_{dep}(c)$. However, this no longer guarantees that profile entries are generated by decreasing departure time, mak-

ing profile updates a non-constant operation for pCSA-P. Also, we can no longer precompute profile indices for pCSA-C. Therefore, we expand footpaths into *pseudoconnections* in our data, as follows. If p_a and p_b are connected by a footpath, we look at all reachable (via the footpath) pairs of incoming connections c_{in} at p_a and outgoing connections c_{out} at p_b. We create a new pseudoconnection (from p_a to p_b, departure time $\tau_{arr}(c_{in})$, and arrival time $\tau_{dep}(c_{out})$) iff there is no other pseudoconnection with a later or equal departure time and an earlier or equal arrival time. Pseudoconnections can be identified by a simultaneous sweep over the incoming/outgoing connections of p_a and p_b. During query, we handle footpaths toward p_t as a special case of the evaluation procedure. Footpaths at p_s are handled by merging the profiles of stops that are reachable by foot from p_s.

ONE-TO-ONE QUERIES. So far we described *all-to-one profile queries*, i. e., from all stops to the target stop p_t. If only the *one-to-one profile* between stops p_s and p_t is of interest, a well-known pruning rule [DKP12; MSWZ07] can be applied to pCSA-P: Before inserting a new profile entry at any stop, we check whether it is dominated by the last entry in the profile at p_s. If so, the current connection cannot possibly be extended to a Pareto-optimal solution at the source, and, hence, can be pruned. However, we still have to continue scanning the full connection array.

MULTICRITERIA. CSA can be extended to compute *multicriteria* profiles, optimizing triples $(\tau_{dep}(p_s), \tau_{arr}(p_t), \#t)$ of departure time, arrival time and number of taken trips. We call this variant mcpCSA-CT. We organize these triples by mapping arrival time $\tau_{arr}(p_t)$ onto *bags* of $(\tau_{dep}(p_s), \#t)$ pairs. Thus, we follow the general approach of pCSA-CT, but now maintain profiles as ($\tau_{arr}(p_t)$, bag) pairs. Evaluating a profile, thus, returns a bag. Where pCSA-CT computes the minimum of two departure times, mcpCSA-CT *merges* two bags, i. e., it computes their union and removes dominated entries. When it scans a connection c, $\#t$ is increased by one for each entry of the evaluated bag, unless c is a pseudoconncetion. It then merges the result with the bag of trip $t(c)$, and updates the profile at $p_{dep}(c)$, accordingly. Exploiting that, in practice, $\#t$ only takes small integral values, we store bags as fixed-length vectors using $\#t$ as index and departure times as values. Merging bags then corresponds to a component-wise minimum, and increasing $\#t$ to shifting the vector's values. A variant, mcpCSA-CT-SSE, uses SIMD-instructions for these operations.

4.1.4 *Experiments*

We ran experiments pinned to one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB

Table 4.1: Size figures for our timetables including figures of the time-dependent (TD), colored time-dependent (TD-col), and time-expanded (TE) graph models [DKP12; MSWZ07; PSWZ08].

Figures	London	Germany	Europe
Stops	20 843	6 822	30 517
Trips	125 537	94 858	463 887
Connections	4 850 431	976 678	4 654 812
Routes	2 135	9 055	42 547
Footpaths	45 652	0	0
Expanded Footpaths	8 436 763	0	0
TD Vertices	97 k	114 k	527 k
TD Arcs	272 k	314 k	1 448 k
TD-col Vertices	21 k	20 k	79 k
TD-col Arcs	71 k	86 k	339 k
TE Vertices	9 338 k	1 809 k	8 778 k
TE Arcs	34 990 k	3 652 k	17 557 k

of L3 and 256 KiB of L2 cache. We compiled our C++ code using g++ 4.7.1 with flags `-O3 -mavx`.

We consider three realistic inputs whose sizes are reported in Table 4.1. They are also used in [DKP12; DPW14; Gei10], but we additionally filter them for (obvious) errors, such as duplicated trips and connections with non-positive travel time. Our main instance, London, is available at [Lds]. It includes tube (subway), bus, tram, Dockland Light Rail (DLR) and is our only instance that also includes footpaths. However, it has no minimum transfer times. The German and European networks were kindly provided by HaCon [HaC84]. Both have minimum transfer times. The German network contains long-distance, regional, and commuter trains operated by Deutsche Bahn during the winter schedule of 2001/02. The European network contains long-distance trains, and is based on the winter schedule of 1996/97. To account for overnight trains and long journeys, our (aperiodic) timetables cover one (London), two (Germany), and three (Europe) consecutive days.

We ran for every experiment 10 000 queries with source and target stops chosen uniformly at random. Departure times are chosen at random between 0:00 and 24:00 (of the first day). We report the running time and the number of label comparisons, counting an SSE operation as a single comparison. Note that we disregard comparisons in the priority queue implementation.

EARLIEST ARRIVAL. In Table 4.2, we report performance figures for several algorithms on the London instance. Besides CSA, we ran re-

Table 4.2: Figures for the earliest arrival problem on our London instance. Indicators are: • enabled, ○ disabled, – not applicable. "Sta." refers to the start criterion. "Trp." indicates the method of trip tracking: connection flag (○), trip flag (•), none (×). "One." indicates one-to-one queries by either using the stop criterion or pruning. We also show results for TE+ALT as reported in [CDD+14] for a slightly different instance and machine.

Alg.	Sta.	Trp.	One.	#Scanned Arcs/Con.	#Reachable Arcs/Con.	#Relaxed Arcs/Con.	#Scanned Footpaths	#L.Cmp. p. Stop	Time [ms]
TE	–	–	○	20 370 117	—	5 739 046	—	977.3	876.2
TD	–	–	○	262 080	—	115 588	—	11.9	18.9
TD-col	–	–	○	68 183	—	21 294	—	3.2	7.3
CSA	○	○	○	4 850 431	2 576 355	11 090	11 500	356.9	16.8
CSA	•	○	○	2 908 731	2 576 355	11 090	11 500	279.7	12.4
CSA	•	•	○	2 908 731	2 576 355	11 090	11 500	279.7	9.7
TE	–	–	•	1 391 761	—	385 641	—	66.8	64.4
TD	–	–	•	158 840	—	68 038	—	7.2	10.9
TD-col	–	–	•	43 238	—	11 602	—	2.1	4.1
TE+ALT	–	–	•	n/a	—	n/a	—	n/a	9.4
CSA	•	•	•	420 263	126 983	5 574	7 005	26.6	2.0
CSA	•	×	•	420 263	126 983	5 574	7 005	26.6	1.8

duced, realistic time-expanded Dijkstra (TE) with two vertices per connection [PSWZ08] and footpaths [MSWZ07], realistic time-dependent Dijkstra (TD), and time-dependent Dijkstra using the optimized coloring model [DKP12] (TD-col). For CSA, we distinguish between scanned, reachable and relaxed connections. Algorithms in Table 4.2 are grouped into blocks.

The first considers one-to-all queries, and we see that basic CSA scans *all* connections (4.8 M), only half of which are reachable. On the other hand, TE scans about half of the graph's arcs (20 M). Still, this is a factor of four more entities due to the modeling overhead of the time-expanded graph. Regarding query time, CSA greatly benefits from its simple data structures and lack of priority queue: It is a factor of 52 faster than TE. Enabling the start criterion reduces the number of scanned connections by 40 %, which also helps query time. Using trip bits increases spatial locality and further reduces query time to 9.7 ms. We observe that just a small fraction of scanned arcs/connections actually improve stop labels. Only then CSA must consider footpaths. The second block considers one-to-one queries. Here, the number of connections scanned by CSA is significantly smaller; journeys in London rarely have long travel times. Since our London instance does not have minimum transfer times, we may remove trip tracking

from the algorithm entirely. This yields the best query time of 1.8 ms on average. Although CSA compares significantly more labels, it outperforms Dijkstra in almost all cases (also see Table 4.4 for other inputs). Only for one-to-all queries on London TD-col is slightly faster than CSA.

FURTHER ENGINEERING OF THE TIME-EXPANDED APPROACH. In our implementation of TE, we already use the reduced graph representation with two vertices per connection. However, it is well known that TE can be further accelerated by, e. g., (1) *node blocking* [DPW09b] and *restricted node exploration* [CDD+14], essentially skipping later connections to already discovered stops, (2) *extended arc relaxation* [CDD+14], where only arrival vertices are inserted into the queue, and (3) a combination with speedup techniques such as *ALT* [GH05]. Note that such techniques are not easily applied to CSA: by design, it does not use a queue to maintain a search front, and skipping a connection by means of checking a flag is not much faster than actually scanning the connection. In [CDD+14], a combination of above techniques (called TE+ALT in Table 4.2) yields query times below 10 ms on a 4-core Intel i5-2500K clocked at 3.30 GHz; Since the considered instance of London has three times as many connections but only half as many stops as our London instance, a precise comparison is difficult. However, in [DPW09b] similar speedups of factor 2.5–10 over basic TE are achieved by applying such improvements. Hence, it seems safe to say that the TE graph model and query algorithms can be engineered to achieve performance within the same order of magnitude as the non graph-based approaches. Nonetheless, CSA offers competitive EA query times with an exceptionally simple algorithm that is much easier to implement and test.

PROFILE AND MULTICRITERIA QUERIES. In Table 4.3 we report experiments for (multicriteria) profile queries on London. Other instances are available in Table 4.4. We compare CSA to SPCS-col [DKP12] (an extension of TD-col to profile queries) and rRAPTOR [DPW14] (an extension of RAPTOR to multicriteria profile queries). Note that in [DPW14], rRAPTOR is evaluated on two-hours range queries, whereas we compute full profile queries. A first observation is that, regarding query time, one-to-all SPCS is outperformed by all other algorithms, even those which additionally minimize the number of transfers. Similarly to our previous experiment, CSA generally does more work than the competing algorithms, but is, again, faster due to its cache-friendlier memory access patterns. We also observe that one-to-all pCSA-C is slightly faster than pCSA-P, even with target pruning enabled, although it scans 2.7 times as many connections because of expanded footpaths. Note, however, that the figure for pCSA-C does not include the postprocessing that removes dominated journeys. Time indexing

Table 4.3: Figures for the (multicriteria) profile problem on London. "#Tr." is the max. number of trips considered. "Arr." indicates minimizing arrival time, "Tran." transfers. "Prof." indicates profile queries. "#Jn." is the number of Pareto-optimal journeys.

Algorithm	#Tr.	Arr.	Tran.	Prof.	One.	#Jn.	#L.Cmp. p. Stop	Time [ms]
SPCS-col	–	•	○	•	○	98.2	477.7	1 262
SPCS-col	–	•	○	•	•	98.2	372.5	843
pCSA-P	–	•	○	•	○	98.2	567.6	177
pCSA-P	–	•	○	•	•	98.2	436.9	161
pCSA-C	–	•	○	•	–	98.2	1 912.5	134
pCSA-CT	–	•	○	•	–	98.2	1 912.5	104
rRAPTOR	8	•	•	•	○	203.4	1 812.5	1 179
rRAPTOR	8	•	•	•	•	203.4	1 579.6	878
rRAPTOR	16	•	•	•	•	206.4	1 634.0	922
mcpCSA-CT	8	•	•	•	–	203.4	15 299.8	255
mcpCSA-CT-SSE	8	•	•	•	–	203.4	1 912.5	221
mcpCSA-CT-SSE	16	•	•	•	–	206.4	3 824.9	466

Table 4.4: Evaluating other instances. Start criterion and trip flags are always used.

						Germany			**Europe**		
Algorithm	#Tr.	Arr.	Tran.	Prof.	One.	#Jn.	#L.Cmp. p. Stop	Time [ms]	#Jn.	#L.Cmp. p. Stop	Time [ms]
TE	–	•	○	○	○	1.0	317.0	117.1	0.9	288.6	624.1
TD-col	–	•	○	○	○	1.0	11.9	3.5	0.9	10.0	21.6
CSA	–	•	○	○	○	1.0	228.7	3.4	0.9	209.5	19.5
TE	–	•	○	○	•	1.0	29.8	11.7	0.9	56.3	129.9
TD-col	–	•	○	○	•	1.0	6.8	2.0	0.9	5.3	11.5
CSA	–	•	○	○	•	1.0	40.8	0.8	0.9	74.2	8.3
pCSA-CT	–	•	○	•	–	20.2	429.5	4.9	11.4	457.6	46.2
rRAPTOR	8	•	•	•	○	29.4	752.1	161.3	17.2	377.5	421.8
rRAPTOR	8	•	•	•	•	29.4	640.1	123.0	17.2	340.8	344.9
mcpCSA-CT-SSE	8	•	•	•	–	29.4	429.5	17.9	17.2	457.6	98.2

further accelerates pCSA-C, indicating that the algorithm is, indeed, memory-bound. Regarding multicriteria profile queries, doubling the number of considered trips also doubles both CSA's label comparisons and its running time. For rRAPTOR the difference is less (only 12 %)—most work is spent in the first eight rounds. Indeed, journeys with more than eight trips are very rare. This justifies mcpCSA-CT-SSE with eight trips, which is our fastest algorithm (221 ms on average). Note that using an AVX2 processor (announced for June 2013), one will be able to process 256 bit-vectors in a single instruction. We, therefore, expect mcpCSA-CT-SSE to perform better for greater numbers of trips in the future.

Conclusions

We have studied the Connection Scan framework of algorithms (CSA) for several public transit route planning problems. One of its strengths is the conceptual simplicity, allowing easy implementation. Yet, it is sufficiently flexible to handle complex scenarios, such as multicriteria profile queries. Our experiments on the metropolitan network of London revealed that CSA is faster than existing, non-preprocessed approaches. All scenarios considered are fast enough for interactive applications. For future work, since CSA does not use a priority queue, parallel extensions seem promising.

4.2 PUBLIC TRANSIT LABELING

Since for aperiodic timetables, the TE model yields a *directed acyclic graph* (DAG), several public transit query problems translate to *reachability* problems. Different methodologies exist to enable fast reachability computation [CHWF13; JW13; MS14; SABW13; YAIY13; YCZ10; ZLWX14]. In particular, the *2-hop labeling* [CHKZ03] scheme associates with each vertex two labels (forward and backward); reachability (or shortest-path distance) can be determined by intersecting the source's forward label and the target's backward label. Modern implementations of this scheme [ADGW12] yield the fastest known query times on road networks, as discussed in Chapter 2.

In this section, we adapt 2-hop labeling to public transit networks, improving query performance by orders of magnitude over previous methods, while keeping preprocessing time practical. Starting from the time-expanded graph model (Section 4.2.2), we extend the labeling scheme by carefully exploiting properties of public transit networks (Section 4.2.3). Besides earliest arrival and profile queries, we address multicriteria and location-to-location queries, as well as reporting the full journey description quickly (Section 4.2.4). We validate our Public Transit Labeling (PTL) algorithm by careful experimental evaluation on large metropolitan and national transit networks (Section 4.2.5), achieving queries within microseconds.

Please note that a very similar approach has appeared in [WLY+15], independently of this work, which originally appeared in [DDPW15].

4.2.1 *Preliminaries*

Throughout Section 4.2, we use the following notation and concepts:

Let $G = (V, A)$ be a (weighted) *directed graph*, where V is the set of vertices and A the set of arcs. An arc between two vertices $u, v \in V$ is denoted by (u, v). A *path* is a sequence of adjacent vertices. A vertex v is *reachable* from a vertex u if there is a path from u to v. A *DAG* is a graph that is both directed and acyclic.

We consider *aperiodic* timetables that consist of sets of stops S, events E, trips T, and footpaths F. *Stops* are distinct locations where one can board a transit vehicle (such as bus stops or subway platforms). *Events* are the scheduled departures and arrivals of vehicles. Each event $e \in E$ has an associated stop $\texttt{stop}(e)$ and time $\texttt{time}(e)$. Let $E(p) = \{e_0(p), \ldots, e_{k_p}(p)\}$ be the list (ordered by time) of events at a stop p. We set $\texttt{time}(e_i(p)) = -\infty$ for $i < 0$, and $\texttt{time}(e_i(p)) = \infty$ for $i > k_p$. For simplicity, we may drop the index of an event (as in $e(p) \in E(p)$) or its stop (as in $e \in E$). A *trip* is a sequence of events served by the same vehicle. A pair of a consecutive departure and arrival events of a trip is a *connection*. *Footpaths* model transfers between nearby stops, each with a predetermined walking duration.

A journey planning algorithm outputs a set of *journeys*. A journey is a sequence of trips (each with a pair of pick-up and drop-off stops) and footpaths in the order of travel. Journeys can be measured according to several criteria, such as arrival time or number of transfers. A journey j_1 *dominates* a journey j_2 if and only if j_1 is no worse in any criterion than j_2. In case j_1 and j_2 are equal in all criteria, we break ties arbitrarily. A set of non-dominated journeys is called a *Pareto set*. Multicriteria Pareto optimization is NP-hard in general, but practical for natural criteria in public transit networks [DPSW13; DPW14; MW06; PSWZ08]. A journey is *tight* if there is no other journey between the same source and target that dominates it in terms of departure and arrival time, e. g., that departs later and arrives earlier.

Given a timetable, stops s and t, and a departure time τ, the (s, t, τ)-*earliest arrival* (EA) problem asks for an s–t journey that arrives at t as early as possible and departs at s no earlier than τ. The (s, t)-*profile* problem asks for a Pareto set of all tight journeys between s and t over the entire timetable period. Finally, the (s, t, τ)-*multicriteria* (MC) problem asks for a Pareto set of journeys departing at s no earlier than τ and minimizing the criteria arrival time and number of transfers. We focus on computing the *values* of the associated optimization criteria of the journeys (i. e., departure time, arrival times, number of transfers), which is enough for many applications. Section 4.2.4 discusses how the full journey description can be obtained with little overhead.

2-HOP LABELING. Our algorithms are based on the 2-hop labeling scheme for directed graphs [CHKZ03]. It associates with every vertex v a *forward label* $L_f(v)$ and a *backward label* $L_b(v)$. In a *reachability labeling*, labels are subsets of V, and vertices $u \in L_f(v) \cup L_b(v)$ are *hubs* of v. Every hub in $L_f(v)$ must be reachable from v, which in turn must be reachable by every hub in $L_b(v)$. In addition, labels must obey the *cover property*: for any pair of vertices u and v, the intersection $L_f(u) \cap L_b(v)$ must contain at least one hub on a u–v path (if it exists). It follows from this definition that $L_f(u) \cap L_b(v) \neq \emptyset$ if and only if v is reachable from u.

In a *shortest path labeling*, each hub $u \in L_f(v)$ also keeps the associated distance $\text{dist}(u, v)$ (or $\text{dist}(v, u)$, for backward labels), and the cover property requires $L_f(u) \cap L_b(v)$ to contain at least one hub on a *shortest* u–v path. If labels are kept sorted by hub ID, a *distance label query* efficiently computes $\text{dist}(u, v)$ by a coordinated linear sweep over $L_f(u)$ and $L_b(v)$, finding the hub $w \in L_f(u) \cap L_b(v)$ that minimizes $\text{dist}(u, w) + \text{dist}(w, v)$. In contrast, a *reachability label query* can stop as soon as any matching hub is found.

In general, smaller labels lead to less space and faster queries. Many algorithms to compute labelings have been proposed [ADGW12; AIY13; CHWF13; JW13; YAIY13; ZLWX14], often for restricted graph

classes. We use (as a black box) the recent RXL algorithm [DGPW14], which efficiently computes small shortest path labelings for a variety of graph classes at scale. It is a sampling-based greedy algorithm that builds labels one hub at a time, with priority to vertices that cover as many relevant paths as possible. More precisely, RXL has two main ingredients: (1) the generation of a vertex order based on (frequently updated) sampling; and (2) the generation of labels from this order using pruned labeling [AIY13].

PRUNED LABELING. A labeling is *hierarchical* if the relation "u is a hub of v" constitutes a partial order. Conversely, a given vertex ordering $\text{rank}(\cdot)$ can be turned into a consistent hierarchical labeling, the smallest of which is called *canonical labeling* [ADGW12]. The *Pruned Landmark Labeling* (PLL) algorithm [AIY13] computes a canonical labeling from a given vertex order, processing vertices from most to least important (in terms of their rank). We describe it in terms of shortest path labelings, but reachability labelings are computed analogously. It starts with an empty labeling. Processing vertex v, it runs a modified forward and backward Dijkstra-search from v. We describe the forward run, the other works analogously. For every vertex w removed from the queue with distance $d(w)$, it checks the distance $\text{dist}_L(w)$ obtained from a v–w-label query on the partially computed labels. If $\text{dist}_L(w) \leqslant d(w)$, a shortest path v–w must already be covered by a higher-ranked vertex and the search is pruned at w. Otherwise, v is added as a hub $(v, \text{dist}(v, w))$ to the backward label $L_b(w)$ of w, and the outgoing arcs of w are scanned. Note that, in case of non-unique shortest paths, PLL breaks ties in favor of higher-ranked vertices. Moreover, for the partial label queries to work efficiently, it is crucial to assign hub ids by rank, which guarantees that partial labels remain sorted. We note that RXL uses PLL not only to generate the labels from an order, but also as a subroutine when producing the order itself [DGPW14].

Different approaches for transforming a timetable into a graph exist (see [PSWZ08] for an overview). Here, we focus on the *time-expanded model*. Since it uses scalar arc costs, it is a natural choice for adapting the labeling approach. In contrast, the *time-dependent model* (another popular approach) associates functions with the arcs, which makes adaption more difficult.

4.2.2 *Basic Earliest Arrival and Profile Queries*

We build the time-expanded graph from the timetable as follows. We group all departure and arrival events by the stop where they occur. We sort all events at a stop by time, merging events that happen at the same stop and time. We then add a vertex for each unique event, a *waiting arc* between two consecutive events of the same stop,

and a *connection arc* for each connection (between the corresponding departure and arrival event). The cost of arc (u, v) is $\texttt{time}(v) - \texttt{time}(u)$, i. e., the time difference of the corresponding events. To account for footpaths between two stops a and b, we add, from each vertex at stop a, a *foot arc* to the first reachable vertex at b (based on walking time), and vice versa. As events and vertices are tightly coupled in this model, we use the terms interchangeably.

Any label generation scheme (we use RXL [DGPW14]) on the time-expanded graph creates two (forward and backward) *event labels* for every vertex (event), enabling *event-to-event queries*. For our application *reachability* labels [YAIY13], which only store hubs (without distances), suffice. First, since all arcs point to the future, time-expanded graphs are DAGs. Second, if an event e is reachable from another event e' (i. e., $L_f(e') \cap L_b(e) \neq \emptyset$), we can compute the time to get from e' to e as $\texttt{time}(e) - \texttt{time}(e')$. In fact, *all* paths between two events have equal cost.

In practice, however, event-to-event queries are of limited use, as they require users to specify both departure *and* arrival times, one of which is usually unknown. Therefore, we discuss earliest arrival and profile queries, which *optimize* arrival time and are thus more meaningful. See Section 4.2.4 for multicriteria queries.

EARLIEST ARRIVAL QUERIES. Given event labels, we answer an (s, t, τ)-EA query as follows. We first find the earliest event $e_i(s) \in E(s)$ at the source stop s that suits the departure time, i. e., with $\texttt{time}(e_i(s)) \geqslant \tau$ and $\texttt{time}(e_{i-1}(s)) < \tau$. Next, we search at the target stop t for the earliest event $e_j(t) \in E(t)$ that is reachable from $e_i(s)$ by testing if $L_f(e_i(s)) \cap L_b(e_j(t)) \neq \emptyset$ and $L_f(e_i(s)) \cap L_b(e_{j-1}(t)) = \emptyset$. Then, $\texttt{time}(e_j(t))$ is the earliest arrival time. One could find $e_j(t)$ using linear search (which is simple and cache-friendly), but binary search is faster in theory and in practice. To accelerate queries, we *prune* (skip) all events $e(t)$ with $\texttt{time}(e(t)) < \tau$, since $L_f(e_i(s)) \cap L_b(e(t)) = \emptyset$ always holds in such cases. Moreover, to avoid evaluating $L_f(e_i(s))$ multiple times, we use *hash-based queries* [DGPW14]: we first build a hash set of the hubs in $L_f(e_i(s))$, then check the reachability for an event $e(t)$ by probing the hash with hubs $h \in L_b(e(t))$.

PROFILE QUERIES. To answer an (s, t)-profile query, we perform a coordinated sweep over the events at s and t. For the current event $e_i(s) \in E(s)$ at the source stop (initially set to the earliest event $e_0(s) \in E(s)$), we find the first event $e_j(t) \in E(t)$ at the target stop that is reachable, i. e., such that $L_f(e_i(s)) \cap L_b(e_j(t)) \neq \emptyset$ and $L_f(e_i(s)) \cap L_b(e_{j-1}(t)) = \emptyset$. This gives us the earliest arrival time $\texttt{time}(e_j(t))$. To identify the latest departure time from s for that earliest arrival event (and thus have a tight journey), we increase i until $L_f(e_i(s)) \cap L_b(e_j(t)) = \emptyset$, then add $(\texttt{time}(e_{i-1}(s)), \texttt{time}(e_j(t)))$

to the profile. We repeat the process starting from the events $e_i(s)$ and $e_{j+1}(t)$. Since we increase either i or j after each intersection test, the worst-case time to find all tight journeys is linear in the number of events (at s and t) multiplied by the size of their largest label.

4.2.3 *Improvements*

Our approach can be refined to exploit features specific to public transit networks. As described so far, our labeling scheme maintains reachability information for *all pairs* of events (by covering all paths of the time-expanded graph, breaking ties arbitrarily). However, in public transit networks we actually are only interested in *certain paths*. In particular, the labeling does *not* need to cover any path ending at a departure event (or beginning at an arrival event). We can thus discard forward labels from arrival events and backward labels from departure events.

TRIMMED EVENT LABELS. Moreover, we can disregard paths representing dominated journeys that depart earlier and arrive later than others (i. e., journeys that are not tight, cf. Section 4.2.1). Consider all departure events of a stop. If a certain hub is reachable from event $e_i(s)$, then it is also reachable from $e_0(s), \dots, e_{i-1}(s)$, and is thus potentially added to the forward labels of all these earlier events. In fact, experiments show that on average the same hub is added to 1.8–5.0 events per stop (depending on the network). We therefore compute *trimmed event labels* by discarding all but the latest occurrence of each hub from the forward labels. Similarly, we only keep the earliest occurrence of each hub in the backward labels. (In preliminary experiments, a much slower algorithm that greedily covers tight journeys explicitly [ADGW12; DGPW14] produced very similar label sizes.)

Unfortunately, we can no longer just apply the query algorithms from Section 4.2.2 with trimmed event labels: if the selected departure event at s does not correspond to a tight journey toward t, the algorithm will not find a solution (though one might exist). One could circumvent this issue by also running the algorithm from subsequent departure events at s, which however may lead to quadratic query complexity in the worst case (for both EA and profile queries).

STOP LABELS. We solve this problem by introducing *stop labels*: For each stop p, we merge all forward event labels $L_f(e_0(p)), \dots, L_f(e_k(p))$ into a forward stop label $SL_f(p)$, and all backward event labels into a backward stop label $SL_b(p)$. Similar to distance labels, each stop label $SL(p)$ is a list of pairs $(h, \texttt{time}_p(h))$, each containing a hub and a time, sorted by hub. For a forward label, $\texttt{time}_p(h)$ encodes the latest departure time from p to reach hub h. More precisely, let h be a hub in an event label $L_f(e_i(p))$: we add the pair $(h, \texttt{time}(e_i(p)))$ to the stop

label $SL_f(p)$ only if $h \notin L_f(e_j(p)), j > i$, i. e., only if h does not appear in the label of another event with a later departure time at the stop. Analogously, for backward stop labels, $\texttt{time}_p(h)$ encodes the earliest arrival time at p from h.

By restricting ourselves to these entries, we effectively discard dominated (non-tight) journeys to these hubs. It is easy to see that these stop labels obey a *tight journey cover property*: for each pair of stops s and t, $SL_f(s) \cap SL_b(t)$ contains at least one hub on each tight journey between them (or any equivalent journey that departs and arrives at the same time; recall from Section 4.2.1 that we allow arbitrary tie-breaking). This property does *not*, however, imply that the label intersection *only* contains tight journeys: for example, $SL_f(s)$ and $SL_b(t)$ could share a hub that is important for long distance travel, but not to get from s to t. The remainder of this section discusses how we handle this fact during queries.

STOP LABEL PROFILE QUERIES. To run an (s,t)-profile query on stop labels, we perform a coordinated sweep over both labels $SL_f(s)$ and $SL_b(t)$. For every matching hub h, i. e., $(h, \texttt{time}_s(h)) \in SL_f(s)$ and $(h, \texttt{time}_t(h)) \in SL_b(t)$, we consider the journey induced by $(\texttt{time}_s(h), \texttt{time}_t(h))$ for output. However, since we are only interested in reporting tight journeys, we maintain (during the algorithm) a tentative set of tight journeys, removing dominated journeys from it on-the-fly. (We found this to be faster than adding all journeys during the sweep and only discarding dominated journeys at the end.) We can further improve the efficiency of this approach in practice by (globally) reassigning hub IDs by the time of day. Note that every hub h of a stop label is still also an event and carries an event time $\texttt{time}(h)$. (Not to be confused with $\texttt{time}_s(h)$ and $\texttt{time}_t(h)$.) We assign sequential IDs to all hubs h in order of increasing $\texttt{time}(h)$, thus ensuring that hubs in the label intersection are enumerated chronologically. Note that this does not imply that journeys are enumerated in order of departure or arrival time, since each hub h may appear anywhere along its associated journey. However, preliminary experiments have shown that this approach leads to fewer insertions into the tentative set of tight journeys, reducing query time. Moreover, as in shortest path labels [DGPW14], we improve cache efficiency by storing the values for hubs and times separately in a stop label, accessing times only for matching hubs.

Overall, stop and event labels have different trade-offs: maintaining the profile requires less effort with event labels (any discovered journey is already tight), but fewer hubs are scanned with stop labels (there are no duplicate hubs).

STOP LABEL EARLIEST ARRIVAL QUERIES. Reassigned hub IDs also enable fast (s, t, τ)-EA queries. We use binary search in $SL_f(s)$ and

$SL_b(t)$ to find the earliest relevant hub h, i. e., with $\texttt{time}(h) \geqslant \tau$. From there, we perform a linear coordinated sweep as in the profile query, finding $(h, \texttt{time}_s(h)) \in SL_f(s)$ and $(h, \texttt{time}_t(h)) \in SL_b(t)$. However, instead of maintaining tentative profile entries $(\texttt{time}_s(h), \texttt{time}_t(h))$, we ignore solutions that depart too early (i. e., $\texttt{time}_s(h) < \tau$), while picking the hub h^* that minimizes the tentative best arrival $\texttt{time}_t(h^*)$. (Note that $\texttt{time}(h) \geqslant \tau$ does not imply $\texttt{time}_s(h) \geqslant \tau$.) Once we scan a hub h with $\texttt{time}(h) \geqslant \texttt{time}_t(h^*)$, the tentative best arrival time cannot be improved anymore, and we stop the query. For practical performance, *pruning* the scan, so that we only sweep hubs h between $\tau \leqslant \texttt{time}(h) \leqslant \texttt{time}_t(h^*)$, is very important.

4.2.4 *Practical Extensions*

So far, we presented stop-to-stop queries, which report the departure and arrival times of the quickest journey(s). In this section, we address multicriteria queries, general location-to-location requests, and obtaining detailed journey descriptions.

MULTICRITERIA OPTIMIZATION AND MINIMUM TRANSFER TIME. Besides optimizing arrival time, many users also prefer journeys with fewer transfers. To solve the underlying multicriteria optimization problem, we adapt our labeling approach by (1) encoding transfers as arc costs in the graph, (2) computing shortest path labels based on these costs (instead of reachability labels on an unweighted graph), and (3) adjusting the query algorithm to find the Pareto set of solutions.

Reconsider the earliest arrival graph from Section 4.2.2. As before, we add a vertex for each unique event, linking consecutive events at the same stop with waiting arcs of cost 0. However, each connection arc (u, w) in the graph is subdivided by an intermediate *connection vertex* v, setting the cost of arc (u, v) to 0 and the cost of arc (v, w) to 1. By interpreting costs of 1 as leaving a vehicle, we can count the number of trips taken along any path. To model staying in the vehicle, consecutive connection vertices of the same trip are linked by zero-cost arcs [PSWZ08]. See Figure 4.1 for an example graph.

A shortest path labeling on this graph now encodes the number of transfers as the shortest path distance between two events, while the duration of the journey can still be deduced from the time difference of the events. Consider a fixed source event $e(s)$ and the arrival events of a target stop $e_0(t), e_1(t), \ldots$ in order of increasing time. The minimum number of transfers required to reach the target stop t never increases with arrival times. (Hence, the whole Pareto set P of multicriteria solutions can be computed with a single Dijkstra run [PSWZ08].)

We exploit this property to compute (s, t, τ)-EA multicriteria (MC) queries from the labels as follows. We initialize P as the empty set. We then perform an (s, t, τ)-EA query (with all optimizations described

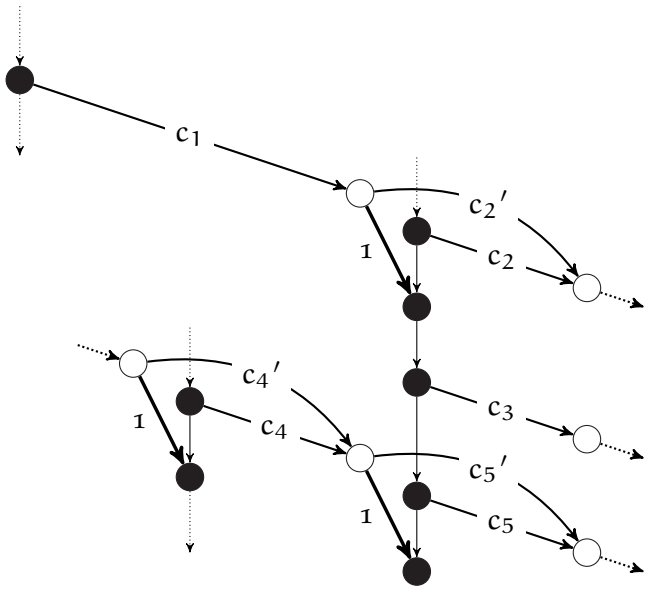


Figure 4.1: Detail of the multicriteria graph (with minimum transfer time). It shows departure and/or arrival event vertices (in black) of three stops, where consecutive events at each stop are connected by vertical waiting arcs. The figure further shows connection vertices (in white) of three trips (trip 1: $c_1, c_2, \ldots$, trip 2: $c_3, \ldots$, trip 3: $c_4, c_5, \ldots$). All arcs have cost 0, except those from connection vertices to event vertices, which have cost 1. This is used to model and count transfers.

in Section 4.2.2) to compute the *fastest* journey in the solution, i. e., the one with most transfers. We add this journey to P. We then check (by performing distance label queries) for each subsequent event at t whether there is a journey with fewer transfers (than the most recently added entry of P), in which case we add the journey to P and repeat. The MC query ends once the last event at the target stop has been processed. We can stop earlier with the following optimization: we first run a distance label query on the *last* event at t to obtain the *smallest* possible number of transfers to travel from s to t. We may then already stop the MC query once we add a journey to P with this many transfers. Note that, since we do not need to check for domination in P explicitly, our algorithm maintains P in constant time per added journey.

MINIMUM TRANSFER TIMES. Transit agencies often model an entire station with multiple platforms as a single stop and account for the time required to change trips inside the station by associating a *minimum transfer time* $\mathrm{mtt}(p)$ with each stop p. To incorporate them into the EA graph, we first locally replace each affected stop p by a *set* of new stops p^*, distributing *conflicting* trips (between which transferring is impossible due to $\mathrm{mtt}(p)$) to different stops of p^*. We then add footpaths between all pairs of stops in p^* with length $\mathrm{mtt}(p)$.

A small set p^* can be computed by solving an appropriate coloring problem [DKP12]. For the MC graph, we need not change the input. Instead, it is sufficient to *shift* each arrival event $e \in E(p)$ by adding $\text{mtt}(p)$ to $\texttt{time}(e)$ before creating the vertices.

LOCATION-TO-LOCATION QUERIES. A query between arbitrary locations s^* and t^*, which may employ walking or driving as the first and last legs of the journey, can be handled by a two-stage approach. It first computes sets $\mathcal{S}$ and $\mathcal{T}$ of relevant stops near the origin s^* and destination t^* that can be reached by car or on foot. With that information, a *forward superlabel* [ADF+12] is built from all forward stop labels associated with $\mathcal{S}$. For each entry $(h, \texttt{time}_p(h)) \in SL_f(p)$ in the label of stop $p \in \mathcal{S}$, we adjust the departure time $\texttt{time}^*_s(h) = \texttt{time}_p(h) - \text{dist}(s^*, p)$ so that the journey starts at s^* and add $(h, \texttt{time}^*_s(h))$ to the superlabel. For duplicate hubs that occur in multiple stop labels, we keep only the latest departure time from s^*. This can be achieved with a coordinated sweep, always adding the next hub of minimum ID. A *backward superlabel* (for $\mathcal{T}$) is built analogously. For location-to-location queries, we then simply run our stop-label-based EA and profile query algorithms using the superlabels. In practice, we need not build superlabels explicitly but can simulate the building sweep during the query (which in itself is a coordinated sweep over two labels). A similar approach is possible for event labels. Moreover, point-of-interest queries (such as finding the closest restaurants to a given location) can be computed by applying known techniques [ADF+12] to these superlabels.

JOURNEY DESCRIPTIONS. While for many applications it suffices to report departure and arrival times (and possibly the number of transfers) per journey, sometimes a more detailed description is needed. We could apply known path unpacking techniques [ADF+12] to retrieve the full sequence of connections (and transfers), but in public transit it is usually enough to report the list of trips with associated transfer stops. For this, we can store with each hub the sequence of trips (and transfer stops) for travel between the hub and its label vertex.

4.2.5 *Experiments*

SETUP. We implemented all algorithms in C++ using Visual Studio 2013 with full optimization. All experiments were conducted on a machine with two 8-core Intel Xeon E5-2690 CPUs and 384 GiB of DDR3-1066 RAM, running Windows 2008R2 Server. All runs are *sequential*. We use at most 32 bits for distances.

We consider four realistic inputs: the metropolitan networks of London (`data.london.gov.uk`) and Madrid (`emtmadrid.es`), and the national networks of Sweden (`trafiklab.se`) and Switzerland (`gtfs.`

Table 4.5: Size of timetables and the earliest arrival (EA) and multicriteria (MC) graphs.

						EA Graph		MC Graph	
Instance	Stops	Conns	Trips	Footp.	Dy.	|V|	|A|	|V|	|A|
London	20.8 k	5,133 k	133 k	45.7 k	1	4,719 k	51,043 k	9,852 k	72,162 k
Madrid	4.7 k	4,527 k	165 k	1.3 k	1	3,003 k	13,730 k	7,530 k	34,505 k
Sweden	51.1 k	12,657 k	548 k	1.1 k	2	8,151 k	34,806 k	20,808 k	93,194 k
Switzerland	27.1 k	23,706 k	2,198 k	29.8 k	2	7,979 k	49,656 k	31,685 k	170,503 k

Table 4.6: Preprocessing figures. Label sizes are averages of forward and backward labels.

	Earliest Arrival						**Multicriteria**			
		Event Labels			Stop Labels			Event Labels		
Instance	RXL [h:m]	Hubs p. lbl	Hubs p. stop	Space [MiB]	Hubs p. stop	Space [MiB]	RXL [h:m]	Hubs p. lbl	Hubs p. stop	Space [MiB]
London	0:54	70	15,480	1,334	7,075	1,257	49:19	734	162,565	26,871
Madrid	0:25	77	49,247	963	9,830	403	10:55	404	258,008	10,155
Sweden	0:32	37	5,630	1,226	1,536	700	36:14	190	29,046	12,637
Switzerland	0:42	42	11,189	1,282	2,970	708	61:36	216	58,022	12,983

geops.ch). London includes all modes of transport, Madrid contains only buses, and the national networks contain both long-distance and local transit. We consider 24-hour timetables for the metropolitan networks, and two days for national ones (to enable overnight journeys). Footpaths were generated using a known heuristic [DKP12] for Madrid; they are part of the input for the other networks. See Table 4.5 for size figures of the timetables and resulting graphs. The average number of unique events per stop ranges from 160 for Sweden to 644 for Madrid. (Recall from Section 4.2.2 that we merge all coincident events at a stop.) Note that no two instances dominate each other (wrt. number of stops, connections, trips, events per stop, and footpaths).

PREPROCESSING. Table 4.6 reports preprocessing figures for the unweighted earliest arrival graph (which also enables profile queries) and the multicriteria graph. For earliest arrival (EA), preprocessing takes well below an hour and generates about one gigabyte, which is quite practical. Although there are only 37–70 hubs per label, the total number of hubs per stop (i. e., the combined size of all labels) is quite large (5,630–49,247). By eliminating redundancy (cf. Section 4.2.3), stop labels have only a fifth as many hubs (for Madrid). Even though they

Table 4.7: Evaluating earliest arrival queries. Bullets (•) indicate different features: profile query (Prof.), stop labels (St. lbs.), pruning (Prn.), hashing (Hash), and binary search (Bin.). The column "=" indicates the average number of matched hubs.

Prof.	St. lbs.	Prn.	Hash	Bin.	London				Sweden				Switzerland			
					Lbls.	Hubs	=	[µs]	Lbls.	Hubs	=	[µs]	Lbls.	Hubs	=	[µs]
○	○	○	○	○	108.4	6,936	1	14.7	68.0	2,415	1	6.9	89.0	3,485	1	8.7
○	○	•	○	○	16.1	1,360	1	5.9	34.4	1,581	1	5.4	33.5	1,676	1	5.8
○	○	•	•	○	16.1	1,047	1	4.2	34.4	1,083	1	3.6	33.5	1,151	1	3.8
○	○	•	•	•	7.0	332	4	2.8	6.5	179	3	2.1	7.6	204	4	2.1
○	•	○	○	○	2.0	13,037	1,126	54.8	2.0	2,855	81	10.0	2.0	5,707	218	20.4
○	•	•	○	○	2.0	861	62	6.2	2.0	711	16	3.6	2.0	699	19	3.8
•	○	○	○	○	658.5	40,892	211	141.7	423.7	13,590	118	39.4	786.6	29,381	240	81.4
•	•	○	○	○	2.0	13,037	1,126	74.3	2.0	2,855	81	12.1	2.0	5,707	218	24.5

need to store an additional distance value per hub, total space usage is still smaller. In general, *average* labels sizes (though not total space) are higher for metropolitan instances. This correlates with the higher number of daily journeys in these networks.

Preprocessing the multicriteria (MC) graph is much more expensive: times increase by a factor of 26.2–54.8 for the metropolitan and 67.9–88 for the national networks. On Madrid, Sweden, and Switzerland labels are five times larger compared to EA, and on London the factor is even more than ten. This is immediately reflected in the space consumption, which is up to 26 GiB (London).

QUERIES. We now evaluate query performance. For each algorithm, we ran 100,000 queries between random source and target stops, at random departure times between 0:00 and 23:59 (of the first day). Table 4.7 reports detailed figures, organized in three blocks: event label EA queries, stop label EA queries, and profile queries (with both event and stop labels). We discuss MC queries later.

We observe that event labels result in extremely fast EA queries (6.9–14.7 µs), even without optimizations. As expected, pruning and hashing reduce the number of accesses to labels and hubs (see columns "Lbls." and "Hubs"). Although binary search cannot stop as soon as a matching hub is found (see the "=" column), it accesses fewer labels and hubs, achieving query times below 3 µs on all instances.

Using stop labels (cf. Section 4.2.3) in their basic form is significantly slower than using event labels. With pruning enabled, however, query times (3.6–6.2 µs) are within a factor of two of the event labels, while saving a factor of 1.1–2.4 in space. For profile queries, stop labels are

Table 4.8: Comparison with the state of the art. Presentation largely based on [BDG+15], with some additional results taken from [BS14]. The first block of techniques considers the EA problem, the second the MC problem and the third the profile problem.

	Instance				Criteria					
Algorithm	Name	Stops [$\cdot 10^3$]	Conns [$\cdot 10^6$]	Dy.	Arr.	Tran.	Prof.	Prep. [h]	Jn.	Query [ms]
CSA [DPSW13]	London	20.8	4.9	1	•	○	○	—	n/a	1.8
ACSA [SW14]	Germany	252.4	46.2	2	•	○	○	0.2	n/a	8.7
CH [Gei10]	Europe (LD)	30.5	1.7	p	•	○	○	<0.1	n/a	0.3
TP [BDG+15]	Madrid	4.6	4.8	1	•	○	○	19	n/a	0.7
TP [BS14]	Germany	248.4	13.9	1	•	○	○	249	0.9	0.2
PTL	London	20.8	5.1	1	•	○	○	0.9	0.9	0.0028
PTL	Madrid	4.7	4.5	1	•	○	○	0.4	0.9	0.0030
PTL	Sweden	51.1	12.7	2	•	○	○	0.5	1.0	0.0021
PTL	Switzerland	27.1	23.7	2	•	○	○	0.7	1.0	0.0021
RAPTOR [DPW14]	London	20.8	5.1	1	•	•	○	—	1.8	5.4
TP [BDG+15]	Madrid	4.6	4.8	1	•	•	○	185	n/a	3.1
TP [BS14]	Germany	248.4	13.9	1	•	•	○	372	1.9	0.3
PTL	London	20.8	5.1	1	•	•	○	49.3	1.8	0.0266
PTL	Madrid	4.7	4.5	1	•	•	○	10.9	1.9	0.0643
PTL	Sweden	51.1	12.7	2	•	•	○	36.2	1.7	0.0276
PTL	Switzerland	27.1	23.7	2	•	•	○	61.6	1.7	0.0217
CSA [DPSW13]	London	20.8	4.9	1	•	○	•	—	98.2	161.0
ACSA [SW14]	Germany	252.4	46.2	2	•	○	•	0.2	n/a	171.0
CH [Gei10]	Europe (LD)	30.5	1.7	p	•	○	•	<0.1	n/a	3.7
TP [BS14]	Germany	248.4	13.9	1	•	○	•	249	16.4	3.3
PTL	London	20.8	5.1	1	•	○	•	0.9	81.0	0.0743
PTL	Madrid	4.7	4.5	1	•	○	•	0.4	110.7	0.1119
PTL	Sweden	51.1	12.7	2	•	○	•	0.5	12.7	0.0121
PTL	Switzerland	27.1	23.7	2	•	○	•	0.7	31.5	0.0245

clearly the best approach. It scans up to a factor of 5.1 fewer hubs and is up to 3.3 times faster, computing the profile of the full timetable period in under 80 µs on all instances. The difference in factors is due to the overhead of maintaining the Pareto set during the query.

COMPARISON. Table 4.8 compares our new algorithm (indicated as *PTL*, for Public Transit Labeling) to the state of the art and also evaluates multicriteria queries. In this experiment, PTL uses event labels with pruning, hashing and binary search for earliest arrival (and multicriteria) queries, and stop labels for profile queries. We compare PTL to CSA [DPSW13] and RAPTOR [DPW14] (currently the fastest algorithms without preprocessing), as well as Accelerated CSA (ACSA) [SW14], Timetable Contraction Hierarchies (CH) [Gei10], and Transfer Patterns (TP) [BCE+10; BS14] (which make use of preprocessing). Since RAPTOR always optimizes transfers (by design), we only include it for the MC problem. Note that the following evaluation should be taken with a grain of salt, as no standardized benchmark instances exist, and many data sets used in the literature are proprietary. Although precise numbers are not available for several competing methods, it is safe to say they use less space than PTL, particularly for the MC problem.

Table 4.8 shows that PTL queries are very efficient. Remarkably, they are faster on the national networks than on the metropolitan ones: the latter are smaller in most aspects, but have more frequent journeys (that must be covered). Compared to other methods, PTL is 2–3 orders of magnitude faster on London than CSA and RAPTOR for EA (factor 643), profile (factor 2,167), and MC (factor 203) queries. We note, however, that PTL is a point-to-point algorithm (as are ACSA, TP, and CH); for one-to-all queries, CSA and RAPTOR would be faster.

PTL has 1–2 orders of magnitude faster preprocessing and queries than TP for the EA and profile problems. On Madrid, EA queries are 233 times faster while preprocessing is faster by a factor of 48. Note that Sweden (PTL) and Germany (TP) have a similar number of connections, but PTL queries are 95 times faster. (Germany does have more stops, but recall that PTL query performance depends more on the frequency of trips.) For the MC problem, the difference is smaller, but both preprocessing and queries of PTL are still an order of magnitude faster than TP (up to 48 times for MC queries on Madrid).

Compared to ACSA and CH (for which figures are only available for EA and profile problems), PTL has slower preprocessing but significantly faster queries (even accounting for different network sizes).

4.2.5.1 *Alternative Preprocessing Techniques*

Before settling on using RXL [DGPW14] as a black box, we tried several other techniques (and variants) for computing reachability within our PTL algorithm. We shortly describe our experience here.

OTHER LABELING TECHNIQUES. Since we mainly deal with reachability labeling (except for multicriteria queries), it is natural to try alternative orders (as an input to pruned labeling) based on the vast reachability literature [CHWF13; JW13; MS14; SABW13; YAIY13; YCZ10; ZLWX14], with the goal of obtaining smaller labels. In particular, we found that orders based on vertex degrees, which work well for unweighted social networks [JW13; YAIY13], do not offer much guidance on the time-expanded timetable graph, which has a very uniform degree distribution. We also tried the order obtained from topological folding (TF) [CHWF13] and the order implicit in the contraction scheme from PReaCH [MS14]. PReaCH is a refined online search (non-labeling) reachability technique that reports excellent results using vertex contraction as an ingredient. It works by iteratively contracting sources and sinks of the DAG (which creates new sources and sinks). We thus tried the reverse order for labeling, selecting the vertex contracted last as the first hub, all the way down to the original sources and sinks. However, TF and PReaCH orders both slowed down label generation by at least an order of magnitude relative to RXL (we aborted the experiment).

TIGHT JOURNEY COVER. In Section 4.2.3, we introduced stop labels, which obey a tight journey cover property: for each pair of stops s and t, the label intersection $SL_f(s) \cap SL_b(t)$ contains at least one hub on each tight journey between them (or any equivalent journey that departs and arrives at the same time). We proposed to obtain stop labels by trimming (reachability-based) event labels, disregarding paths that correspond to dominated journeys (that depart earlier and arrive later than other journeys). To validate our approach, we have also evaluated an algorithm that (greedily) covers tight journeys explicitly.

Similar to the Offline Path Greedy algorithm from [DGPW14], the approach computes the set of all tight journeys (breaking ties arbitrarily) between all stops of the timetable (we used a variant of profile CSA, cf. Section 4.1, but other algorithms such as RAPTOR [DPW14] should work as well), with indices keeping track of which events cover each journey and which journeys are covered by each event. Then, in each iteration, it greedily picks the event that covers most uncovered journeys. It adds this event as a hub to departure and arrival events of the journeys covered, while updating the indices of other events these journeys were covered by. It terminates as soon as all tight journeys are covered. On the small instance (38 195 connections, 4 573 stops, and 1 252 footpaths) we could test (since memory for the indices is the bottleneck), this greedy approach achieves stop labels 5.7 % smaller (on average) than our default approach from Section 4.2.3.

Even for tight journeys, however, public transit networks often have multiple alternative ways of traveling that lead to the same

departure and arrival time. (In contrast, on road networks, shortest paths are mostly unique.) Essentially, waiting times due to difference in frequency (e. g., metro vs. train) can be attributed differently to journey legs.[1] We therefore found it important to break ties in the most favorable way when determining the order, as in the Online Path Greedy algorithm from [DGPW14]. Extending our approach to compute all events that cover a specific pair of departure and arrival event (i. e., computing all tight journeys without prior tie-breaking), we achieve stop labels 20.5 % smaller than our default approach from Section 4.2.3. While this results in slightly faster queries, our default approach offers a much better trade-off between label size and preprocessing effort.

However, the order in which vertices are chosen by the greedy cover algorithms as hubs offers an interesting perspective: when we look at the corresponding stops (or routes) of these hubs, they are very evenly distributed. This is an indication that choosing hubs freely, i. e., not from whole stops (or routes) at once (which, in contrast, would be natural for time-dependent graph models), is important.

OTHER ALGORITHMIC APPROACHES. The time-expanded graph model is not particularly space-efficient. Moreover, graph search-based queries on this model are usually less efficient [BDG+15] than in the time-dependent model, and also than RAPTOR [DPW14] and CSA (see Section 4.1), which are not search-based. Unfortunately, we found it challenging to incorporate these faster techniques within our label-based algorithm.

Since CSA does not maintain a search space (it is just a linear scan of an array), there is not much hope in combining it with pruned labeling (or RXL). Similarly, we do not see how to prune the scan of routes in RAPTOR. We did try running pruned labeling on the time-dependent graph model, but found the pruning test to be an issue. One possibility is to base the pruning test on pre-trimmed stop labels, but this amounts to EA queries, which would be too slow with stop labels (cf. Table 4.7), particularly since we cannot yet reassign hub IDs (for then, partial labels used for pruning, would not be sorted implicitly). An alternative would be to build event labels, but this is essentially the same as using the time-expanded approach. In fact, preliminary experiments indicated that the time-dependent approach was slower than plain pruned labeling on the time-expanded graph.

1 In the time-expanded graph, consider a fixed pair of departure and arrival events. The intersection of the set of all events reachable by the departure event with the set of all events that can reach the arrival event is exactly the set of all events that cover the considered journey.

Conclusions

We have introduced PTL, a new preprocessing-based algorithm for journey planning in public transit networks, by revisiting the time-expanded model and adapting the Hub Labeling approach to it. By further exploiting structural properties specific to timetables, we obtained simple and efficient algorithms that outperform the current state of the art on large metropolitan and country-sized networks by orders of magnitude for various realistic query types. Future work includes developing tailored algorithms for hub computation (instead of using RXL as a black box), compressing the labels (e. g., using techniques from [BS14] and [DGPW14]), exploring other hub representations (e. g., using trips instead of events, as in 3-hop labeling [YAIY13]), using multi-core and instruction-based parallelism for preprocessing and queries, and handling dynamic scenarios (e. g., temporary station closures and train delays or cancellations [CDD+14; MMPZ13]).

5 MULTIMODAL JOURNEY PLANNING

In this chapter, we study the problem of finding multimodal journeys in transportation networks. However, it is important to note that the transition from public transit to multimodal journey planning is fluid, as mentioned in Chapter 2. We find the following categorization (increasing in complexity) helpful, but note that in the literature each variant is sometimes described as being multimodal:

- Public transit (bus, ferry, metro, tram, train, ...) augmented by precomputed footpaths for short transfers,
- Public transit as above, plus initial (from both source and target) limited radius search (involving walking, biking, and driving) for location-location queries (i. e., "door-to-door"),
- Public transit with unrestricted initial walking, biking, driving,
- *Fully multimodal search,* i. e., with unrestricted walking, biking, driving before, between, and after public transit trips,

In this chapter, we examine the latter case. Thereby, as in Chapter 3, we aim at supporting user preferences and offering choice. An interesting challenge in multimodal route planning is that a quickest path may have an infeasible or undesirable (to a particular user) sequence of modal transfers (e. g., taking a private car between train rides). Furthermore, in a multimodal setting, it can be hard for the user to comprehend all available traveling options. A good multimodal journey planner should provide the user with a concise yet diverse set of choices (in order not to overwhelm yet provide good options).

CHAPTER OUTLINE. Section 5.1 introduces User-Constrained Contraction Hierarchies (UCCH), a multimodal route-planning technique that handles mode-sequence constraints as a user input for each query (as opposed to already during preprocessing). Then, Section 5.2 examines fully-multimodal multicriteria search, where we propose using fuzzy logic [FA04; Zad88] to identify, in a principled way, a modest-sized subset of representative journeys.

5.1 USER-CONSTRAINTS ON MULTIMODAL TRANSFERS

Here, we present *User-Constrained Contraction Hierarchies* (UCCH), the first multimodal speedup technique that handles arbitrary mode-sequence constraints as input to the query—a feature unavailable from

previous techniques. Unlike Access-Node Routing, it also answers local queries correctly and requires significantly less preprocessing effort. We revisit one technique, namely *vertex contraction*, that has proven successful in road networks (cf. Chapter 3): By ensuring that shortcuts never span multiple modes of transport, we extend Contraction Hierarchies [GSSV12] in a sound manner. Moreover, we show how careful engineering further helps our scenario. Our experimental study shows that, unlike previous techniques, we can handle an intercontinental instance composed of cars, railways and flights with over 50 million vertices, 125 million edges, and 30 thousand stations. With only 557 MiB of auxiliary data, we achieve query times that are fast enough for practicable applications.

The rest of this section is organized as follows. Section 5.1.1 sets necessary notation, summarizes graph models we use, and precisely defines the problem we are solving. Section 5.1.1.3 introduces our new technique. Finally, Section 5.1.5 presents experiments to evaluate our algorithm.

5.1.1 *Preliminaries*

Throughout Section 5.1, we use the following notation and concepts:

Let $G = (V, E)$ be a *directed graph*, where V is the set of *vertices* and $E \subseteq V \times V$ the set of *edges*. For an edge $(u, v) \in E$, we call u the *tail* and v the *head* of the edge. The *degree* of a vertex $u \in V$ is defined as the number of edges $e \in E$ where u is either head or tail of e. The *reverse* graph $\overleftarrow{G} = (V, \overleftarrow{E})$ of G is obtained from G by flipping all edges, i. e., $(u, v) \in \overleftarrow{E}$ if and only if $(v, u) \in E$. Note that we use the terms graph and network interchangeably. To distinguish between different modes of transport, our graphs are *labeled* by vertex labels $\text{lbl} : V \to \Sigma$ and edge labels $\text{lbl} : E \to \Sigma$. Often Σ is called the *alphabet* and contains the available modes of transport in G, for example, `road`, `rail`, `flight`. All edges in our graphs are *weighted* by periodic time-dependent travel time functions $f : \Pi \to \mathbb{N}_0$ where Π depicts a set of time points (think of it as the seconds of a day). If f is constant over Π, we call f *time-independent*. Respecting periodicity in a meaningful way, we say that a function f has the *FIFO property* if for all $\tau_1, \tau_2 \in \Pi$ with $\tau_1 \leqslant \tau_2$ it holds that $f(\tau_1) \leqslant f(\tau_2) + (\tau_2 - \tau_1)$. In other words, waiting never pays off. Moreover, we require link and merge operations which generalize the summation and minimum operations from scalar values to travel time functions. Thereby, the *link* operation of two functions f_1, f_2 is defined for any departure time τ as $(f_1 \oplus f_2)(\tau) = f_1(\tau) + f_2(\tau + f_1(\tau))$, and depicts the total travel time when first evaluating f_1 (at departure time τ) and then f_2 (at departure time $\tau + f_1(\tau)$, i. e., the arrival time after "traversing" f_1). The *merge* operation $\min(f_1, f_2)(\tau)$ is defined as the element-wise minimum of f_1 and f_2, i. e., $\min(f_1(\tau), f_2(\tau))$. Note that to depict the travel time

function $f(\tau)$ of an edge $e \in E$, we sometimes write $\text{len}(e, \tau)$, or just $\text{len}(e)$ if it is clear from the context that $\text{len}(e, \tau)$ is constant over all choices of τ.

In time-dependent graphs there are two types of queries relevant for us: A *time-query* has as input $s \in V$ and a departure time τ. It computes a shortest path tree to every vertex $u \in V$ when departing from s at time τ. In contrast, a *profile-query* computes a shortest path graph from s to all $u \in V$, consisting of shortest paths for all departure times $\tau \in \Pi$.

Whenever appropriate, we use some notion of formal languages. A finite sequence $w = \sigma_0 \sigma_1 \dots \sigma_k$ of symbols $\sigma_i \in \Sigma$ is called a *word*. A not necessarily finite set of words L is called formal *language* (over Σ). A nondeterministic finite *automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ characterized by the set Q of *states*, the *transition relation* $\delta \subseteq Q \times \Sigma \times Q$, and sets $S \subseteq Q$ of *initial states* and $F \subseteq Q$ of *final states*. A language L is called *regular* if and only if there is a finite automaton $\mathcal{A}_L$ such that $\mathcal{A}_L$ accepts L.

5.1.1.1 *Models*

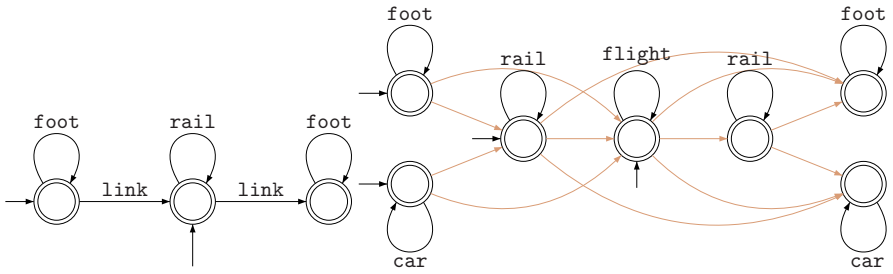
Following [DPW09a], our multimodal graphs are composed of different models for each mode of transportation. We briefly introduce each model and explain how they are combined.

In the *road network*, vertices model intersections and edges depict street segments. We either label edges by `car` for roads or `foot` for pedestrian paths. Our road networks are weighted by the average travel time of the street segment. For pedestrians we assume a walking speed of 4.5 kph. Note that our road networks are time-independent.

Regarding the *railway network*, we use the coloring model [DKP12] which is based on the well-known realistic time-dependent graph model [PSWZ08]. It consists of station vertices connected to route vertices. Trains are modeled between route vertices via time-dependent edges. Different trains use the same route vertex as long as they are not conflicting. In the coloring model conflicting trains are computed explicitly which yields significantly smaller graphs compared to the original realistic time-dependent model (without dropping correctness). Moreover, to enable transfers between trains, some station vertices are interconnected by time-independent foot paths. See [DKP12] for details. We label vertices and edges with `rail`. Note that we also use this model for bus networks.

Finally, to model *flight networks*, we use the time-dependent phase II model [DPWZ09]. It has small size and models airport procedures realistically. Vertices and edges are labeled with `flight`.

Note that the travel time functions in our networks are a special form of piecewise linear functions that can be efficiently evaluated [DKP12; PSWZ08]. Also all edges in our networks have the FIFO property.



(a) `foot-and-rail`. (b) `hierarchical`.

Figure 5.1: Two example automata. In the right figure, light edges are labeled as `link`.

MERGING THE NETWORKS. To obtain an integrated *multimodal* network $\mathbf{G} = (\mathbf{V}, \mathbf{E})$, we merge the vertex and edge sets of each individual network. Detailed data on transfers between modes of transport was not available to us. Thus, we heuristically add link edges labeled `link`. More precisely, we link each station vertex in the railway network to its geographically closest vertex of the road network. We also link each airport vertex of the flight network to their closest vertices in the road and rail networks. Thereby we only link vertices that are no more than distance δ apart, a parameter chosen for each instance. The time to traverse a link edge is computed from its geographical length and a walking speed of 4.5 kph.

5.1.1.2 *Path Constraints on the Sequences of Transport Modes*

Since the naïve approach of using Dijkstra's algorithm on the combined network $\mathbf{G}$ does not incorporate modal constraints, we consider the Label Constrained Shortest Path Problem (LCSPP) [BJMoo]: Each edge $e \in \mathbf{E}$ has a label $\text{lbl}(e)$ assigned to it. The goal is to compute a shortest s-t-path P where the word $w(P)$ formed by concatenating the edge labels along P is element of a language L, a query input.

Modeling sequence constraints is done by specifying L. To represent mode sequence constraints, regular languages of the following form suffice. The alphabet Σ consists of the available transport modes. In the corresponding NFA $\mathcal{A}_L$, states depict one or more transport modes. To model traveling within one transport mode, we require $(q, \sigma, q) \in \delta$ for those transport modes $\sigma \in \Sigma$ that q represents. Moreover, to allow transfers between different modes of transport, states $q, q' \in Q$, $q \neq q'$ are connected by `link` labels, i. e., $(q, \texttt{link}, q') \in \delta$. Finally, states are marked as initial/final if its modes of transport can be used at the beginning/end of the journey. Example automata are shown in Figure 5.1.

We refer to this variant of LCSPP as LCSPP-MS (as in Modal Sequences). In general, LCSPP is solvable in polynomial time, if L is

context-free. In our case, a generalization of Dijkstra's algorithm works [BJMoo].

5.1.1.3 *Contraction Hierarchies (CH)*

Our algorithm is based on Contraction Hierarchies [GSSD08; GSSV12]. Preprocessing works by heuristically ordering the vertices of the graph by an *importance* value (a linear combination of edge expansion, number of contracted neighbors, among others). Then, all vertices are contracted in order of ascending importance. To contract a vertex $v \in V$, it is removed from G, and shortcuts are added between its neighbors to preserve distances between the remaining vertices. The index at which v has been removed is denoted by $\text{rank}(v)$. To determine if a shortcut (u, w) is added, a local search from u is run (without looking at v), until w is settled. If $\text{len}(u, w) \leqslant \text{len}(u, v) + \text{len}(v, w)$, the shortcut (u, w) is not added, and the corresponding shorter path is called a *witness*.

The CH query is a bidirectional Dijkstra search operating on G, augmented by the shortcuts computed during preprocessing. Both searches (forward and backward) go "upward" in the hierarchy: The forward search only visits edges (u, v) where $\text{rank}(u) \leqslant \text{rank}(v)$, and the backward search only visits edges where $\text{rank}(u) \geqslant \text{rank}(v)$. Vertices where both searches meet represent candidate shortest paths with combined length μ. The algorithm minimizes μ, and a search can stop as soon as the minimum key of its priority queue exceeds μ. Furthermore, we make use of *stall-on-demand*: When a vertex v is scanned in either query, we check for all its incident edges $e = (u, v)$ of the *opposite* direction if $\text{dist}(u) + \text{len}(e) < \text{dist}(v)$ holds ($\text{dist}(v)$ denotes the tentative distance at v). If this is the case, we may prune the search at v. See [GSSV12] for details.

PARTIAL HIERARCHY. If the preprocessing is aborted prematurely, i. e., before all vertices are contracted, we obtain a partial contraction hierarchy (PCH). Let $\text{rank}(v) = \infty$ if and only if v is never contracted, then the same query algorithm as for Contraction Hierarchies is applicable and yields correct results [BDS+10]. The induced subgraph of all uncontracted vertices is called the *core*, and the remaining (contracted) subgraph the *component*. Note that both core and component can contain shortcuts not present in the original graph.

PERFORMANCE. Both preprocessing and query performance of CH depend on the number of shortcuts added. It works well if the network has a pronounced hierarchy, i. e., far journeys eventually converge to a "freeway subnetwork" which is of a small fraction in size compared to the total graph [AFGW10]. Note that if computing a complete hierarchy produces too many shortcuts, one can always abort early and compute a partial hierarchy. A possible stopping criterion

is the *average vertex degree* on the core that is approached during the contraction process.

Our Approach

We now introduce our basic approach and show how CH can be used to compute shortest path with restrictions on sequences of transport modes. We first argue that applying CH on the combined multimodal graph $\mathbf{G}$ without careful consideration either yields incorrect results to LCSPP-MS or finalizes the automaton $\mathcal{A}$ during preprocessing. We then introduce UCCH: A practical adaption of Contraction Hierarchies to LCSPP-MS that enables arbitrary modal sequence constraints as query input. Further improvements that help accelerating both preprocessing and queries are presented in Section 5.1.4.

5.1.2 *Contraction Hierarchies for Multimodal Networks*

Let $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ be a multimodal network. Recall that $\mathbf{G}$ is a combination of time-independent and time-dependent networks (for example, of road and rail), hence, contains edges having both constants and travel time functions associated with them. Applying CH to $\mathbf{G}$ already requires some engineering effort: Shortcuts may represent paths containing edges of different type. In order to compute the shortcuts' travel time functions, these edges have to be linked, resulting in inhomogeneous functions that slow down both preprocessing and query performance. More precisely, when a path $P = (e_1, \dots, e_k)$ is composed into a single shortcut edge e', its labels need to be *concatenated* into a super label $\mathrm{lbl}(e') = \mathrm{lbl}(e_1) \cdots \mathrm{lbl}(e_k)$. In particular, if there are subsequent edges e_i, e_j in P where $\mathrm{lbl}(e_i) \neq \mathrm{lbl}(e_j)$, the shortcut induces a modal transfer. Running a query where this particular mode change is prohibited potentially yields incorrect results: The shortcut must not be used but the label constrained path (i. e. the one without this transfer) may have been discarded during preprocessing by the witness search (see Section 5.1.1.3). Note that the partial time-dependent nature of $\mathbf{G}$ further complicates things. A shortcut $e' = (u, v)$ needs to represent the travel time profile from u to v, that is, the underlying path P depends on the time of day. As a consequence, the super label of e' is time-dependent as well.

If the automaton $\mathcal{A}$ is known during preprocessing, we can modify CH preprocessing to yield correct query results with respect to $\mathcal{A}$. While contracting vertex $v \in \mathbf{G}$ and thereby considering to add a shortcut $e' = (u, w)$, we look at its super label $\mathrm{lbl}(e') = \mathrm{lbl}(e_1) \cdots \mathrm{lbl}(e_k)$. To determine if e' has to be inserted, we run multiple witness searches as follows: For each state $q \in \mathcal{A}$ where q represents $\mathrm{lbl}(v)$, we run a multimodal profile-search from u (ignoring v). We run it with q as initial state and all those states $q' \in \mathcal{A}$ as final state, where q' is

reachable from q in $\mathcal{A}$ by applying $\text{lbl}(e')$. Only if for all these profile-searches $\text{dist}(w) \leqslant \text{len}(e')$ holds, the shortcut e' is not required: For every relevant transition sequence of the automaton, there is a shorter path in the graph. Note that shortcuts $e' = (u, w)$ may be required even if an edge from u to w already existed before contraction. This results in parallel edges for different subsequences of the constraint automaton.

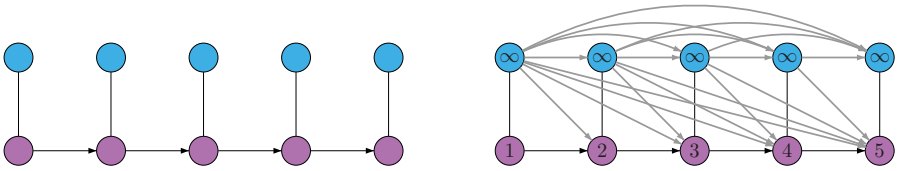
This approach which we call *State-Dependent CH* (SDCH) has some disadvantages, however. First, witness search is slow and less effective than in the unimodal scenario, resulting in many more shortcuts. This hurts preprocessing and query performance. Adding to it the more complicated data structures required for inhomogeneous travel time functions and arbitrary label sequences, SDCH combines challenges of both Flexible CH [GKS10] and Timetable CH [Gei10]. As a result we expect a significant slowdown over unimodal CH on road networks. But most notably, SDCH requires a predetermined automaton $\mathcal{A}$ during preprocessing.

5.1.3 *UCCH: Contraction for User-Constrained Route Planning*

We now introduce User-Constrained Contraction Hierarchies (UCCH). Unlike SDCH, it can handle arbitrary sequence constraint automata during query and has a simpler witness search. We first turn toward preprocessing before we go into detail about the query algorithm.

PREPROCESSING. The main reason behind the disadvantages discussed in Section 5.1.2 is the computation of shortcuts that span over boundaries of different modal networks. Instead, let Σ be the alphabet of labels of a multimodal graph $\mathbf{G}$. We now process each subnetwork independently. We compute—in no particular order—a partial Contraction Hierarchy restricted to the subgraph $G_{\text{lbl}} = (V_{\text{lbl}}, E_{\text{lbl}})$ (for every $\text{lbl} \in \Sigma$). Here, G_{lbl} is exactly the original graph of the particular transportation mode (before merging). We consider the traditional contraction order with the exception of *transfer vertices*: Vertices which are incident to at least one edge labeled `link` in $\mathbf{G}$. We fix the rank of all such vertices v to infinity, i. e., they are never contracted. Note that all other vertices have only incident edges labeled by lbl in $\mathbf{G}$. As a result, shortcuts only span edges within one modal network. Hence, we neither obtain inhomogeneous travel time functions nor "mixed" super labels. We set the label of each shortcut edge e' to $\text{lbl}(e)$, where e is an arbitrary edge along the path, represented by e'.

To determine if a shortcut $e' = (u, w)$ is required (when contracting a vertex v), we restrict the witness search to the modal subnetwork G_{lbl} of v. Restricting the search space of witness searches does not yield incorrect query results: Only *too many* shortcuts might be inserted, but no required shortcuts are *omitted*. In fact, this is a common technique



(a) Input graph. (b) Graph after contraction.

Figure 5.2: Contracting only route vertices in the realistic time-dependent rail model [PSWZ08]. The bottom row of vertices are station vertices, while the top row are route vertices contracted in the order depicted by their labels. Grey edges represent added shortcuts. Note that these shortcuts are required as they incorporate different transfer times (for boarding and exiting vehicles at different stations).

to accelerate CH preprocessing [GSSV12]. Note that broadening the witness search beyond network boundaries is prohibitive in our case: It may find a shorter u-v-path using parts of other modal networks. However, such a path is not necessarily a witness if one of these other modes is forbidden during the query. Thus, we must not take it into account to determine if e' can be dropped.

Our preprocessing results in a partial hierarchy for each modal network of $\mathbf{G}$. Its transfer vertices are not contracted, thus, stay at the top of the hierarchy. Recall that we call the subgraph induced by all vertices v with $\text{rank}(v) = \infty$ the *core*. Because of the added shortcuts, the shortest path between every pair of core vertices is also fully contained in the core. As a result, we achieve independence from the automaton $\mathcal{A}$ during preprocessing.

A PRACTICAL VARIANT. Sequence-constrained contraction is independent for every modal network of $\mathbf{G}$: We can use any combination of partial, full or no contraction. Our *practical variant* only contracts time-independent modal networks, that is, the road networks. Contracting the time-dependent networks is much less effective. Recall that we do not contract station vertices as they have incident link edges. Applying contraction only on the non-station vertices, however, yields too many shortcuts (see Figure 5.2 and [Gei10]). It also hides information encoded in the timetable model (such as railway lines), further complicating query algorithms [BDGM09].

QUERY. Our query algorithm combines the concept of a multimodal Dijkstra algorithm with unimodal CH. Let $s, t \in \mathbf{V}$ be source and target vertices and $\mathcal{A}$ some finite automaton with respect to LCSPP-MS. Our query algorithm works as follows. First, we initialize distance values for all pairs of $(v, q) \in \mathbf{V} \times \mathcal{A}$ with infinity. We now run a bidirectional Dijkstra search from s and t. Each search runs independently and maintains priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$ of tuples (v, q) where $v \in \mathbf{V}$

and $q \in \mathcal{A}$. We explain the algorithm for the forward search; the backward search works analogously. The queue $\overrightarrow{Q}$ is ordered by distance and initialized with (s, q) for all initial states q in $\mathcal{A}$ (the backward queue is initialized with respect to final states). Whenever we extract a tuple (v, q) from Q, we scan all edges $e = (v, w)$ in $\mathbf{G}$. For each edge, we look at all states q' in $\mathcal{A}$ that can be reached from q by $\text{lbl}(e)$. For every such pair (w, q') we check whether its distance is improved, and update the queue if necessary. To use the preprocessed data, we consider the graph $\mathbf{G}$, augmented by all shortcuts computed during preprocessing. We run the aforementioned algorithm, but when scanning edges from a vertex v, the forward search only looks at edges (v, w) where $\text{rank}(w) \geqslant \text{rank}(v)$. Similarly, the backward search only looks at edges (v, w) where $\text{rank}(v) \geqslant \text{rank}(w)$. Note that by these means we automatically search inside the core whenever we reach the top of the hierarchy. Thereby we never reinitialize any data structures when entering the core like it is typically the case for core-based algorithms, e. g., Core-ALT [DSSW09]. The stopping criterion carries over from basic CH: A search stops as soon as its minimum key in the priority queue exceeds the best tentative distance value μ. We also use stall-on-demand, however, only on the component.

Intuitively, the search can be interpreted as follows. We simultaneously search upward in those hierarchies of the modal networks that are either marked as initial or as final in the automaton $\mathcal{A}$. As soon as we hit the top of the hierarchy, the search operates on the common core. Because we always find correct shortest paths between core vertices in *any* modal network, our algorithm supports *arbitrary* automata (with respect to LCSPP-MS) as query input. Note that our algorithm implicitly computes *local* queries which use only one of the networks. It makes the use of a separate algorithm for local queries, as in [DPW09a], unnecessary.

HANDLING TIME-DEPENDENCY. Some of the networks in $\mathbf{G}$ are time-dependent. Weights of time-dependent edges (u, v) are evaluated for a departure time τ. However, running a reverse search on a time-dependent network is non-trivial, since the arrival time at the target vertex is not known in advance. Several approaches, such as using the lower-bound graph for the reverse search, exist [BGNS10; DN08], but they complicate the query algorithm. Recall that in our practical variant we do not contract any of the time-dependent networks, hence, no time-dependent edges are contained in the component. This makes backward search on the component easy for us. We discuss search on the core in the next section.

5.1.4 *Improvements*

We now present improvements to our algorithm, some of which also apply to CH.

AVERAGE VERTEX DEGREE. Recall that whenever we contract a modal network, we never contract transfer vertices, even if they were of low importance in the context of that network. As a result, the number of added shortcuts may increase significantly. Thus, we stop the contraction process as soon as the *average vertex degree* in the core exceeds a value α. By varying α, we trade off the number of core vertices and the number of core edges: Higher values of α produce a smaller core but with more shortcut edges. We evaluate a good value of α experimentally.

EDGE ORDERING. Due to the higher average vertex degree compared to unimodal CH, the search algorithm has to look at more edges. Thus, we improve performance of iterating over incident edges of a vertex v by *reordering* them locally at v: We first arrange all outgoing edges, followed by all bidirected edges, and finally, all incoming edges. By these means, the forward respective backward search only needs to look at their relevant subsets of edges at v. The same optimization is applied to the stalling routine. Preliminary experiments revealed that edge reordering improves query performance up to 21 %.

VERTEX ORDERING. To improve the cache hit rate for the query algorithm, we also reorder vertices such that adjacent vertices are stored consecutively with high probability. We use a DFS-like algorithm to determine the ordering [DGNW11]. Because most of the time is spent on the core, we also move core vertices to the front. This improves query performance up to a factor of 2.

CORE PRUNING. Recall that a search stops as soon as its minimum key from the priority queue exceeds the best tentative distance value μ. This is conservative, but necessary for CH (and UCCH) to be correct. However, UCCH spends a large fraction of the search inside the core. We can easily expand road and transfer edges both forward and backward, but because of the conservative stopping criterion, many core vertices are settled twice. To reduce this amount, we do not scan edges of core vertices v, where v has been settled by both searches and did not improve μ. A path through v is provably not optimal. This increases performance by up to 47 %. Another alternative is not applying bidirectional search on the core at all. The forward search continues regularly, while the backward search does not scan edges incident to core vertices. This approach turns out most effective with a performance increase by a factor of 2.

STATE PRUNING. Recall that our query algorithm maintains distances for *pairs* (v, q) where $v \in \mathbf{V}$ and $q \in \mathcal{A}$. Thus, whenever we scan an edge $(u, v) \in \mathbf{E}$ resulting in some state $q \in \mathcal{A}$, we update the distance value of (v, q) only if it is improved, and discard (or prune) it otherwise. However, we can even make use of a stronger *state pruning* rule: Let q_i and q_j be two states in $\mathcal{A}$. We say that q_i *dominates* q_j if and only if the language $L_{\mathcal{A}}(q_j)$ accepted by $\mathcal{A}$ with modified initial state q_j is a subset of the language $L_{\mathcal{A}}(q_i)$ accepted by $\mathcal{A}$ with modified initial state q_i. In other words, any feasible mode sequence beginning with q_j is also feasible when starting at q_i. As a consequence, when we are about to update a pair (v, q_j), we can additionally prune (v, q_j) if there exists a state q_i that dominates q_j and where (v, q_i) has smaller distance: Any shortest path from v is provably not using (v, q_j). As an example, consider the first automaton in Figure 5.1. Let its states be denoted by $\{q_0, q_1, q_2\}$, from left to right. Here, q_0 dominates q_2 with respect to our definition: Any foot path beginning at state q_2 is also a feasible (foot) path beginning at state q_0. Therefore, any pair (v, q_2) can be pruned if (v, q_0) has better distance than (v, q_2). State pruning improves performance by $\approx 10\,\%$.

STATE-INDEPENDENT SEARCH IN COMPONENT. Automata are used to model sequence constraints, however, by definition their state may only change when traversing link edges. In particular, when searching inside the component, there is never a state transition (recall that all link edges are inside the core). Thus, we use the automaton only on the core. We start with a regular unimodal CH-query. Whenever we are about to insert a core vertex v into the priority queue for the first time on a branch of the shortest path tree, we create labels (v, q) for all initial/final states q (regarding forward/backward search). Because the amount of settled component vertices on average is small compared to the total search space, we do not observe a performance gain. However, on large instances with complicated query automata we save up to 1.1 GiB of RAM during query by keeping only one distance value for each component vertex. Recall that component vertices constitute the major fraction of the graph.

PARALLELIZATION. In general, the multimodal graph $\mathbf{G}$ is composed of more than one contractable modal subnetwork, for instance foot and car. In this case, we have to run the aforementioned unimodal CH-query on every component individually. Because these queries are independent from each other, we are able to parallelize them easily. In a first phase, we allocate one thread for every contracted network which then runs the unimodal CH-query on its respective component until it hits the core. In the second phase, we synchronize the threads, and continue the search on the core sequentially. Note that we only

Table 5.1: Comparing size figures of our input instances. The column "Col." indicates whether we use the coloring approach (see Section 5.1.1.1) to model the railway subnetwork. The bottom two instances are taken from [DPW09a].

	Public Transportation			Road		
Network	Stations	Connections	Col.	Vertices	Edges	Density
`ny-road-rail`	16 897	2 054 896	•	579 849	1 527 594	1 : 56
`de-road-rail`	6 822	489 801	•	5 055 680	12 378 224	1 : 749
`europe-road-rail`	30 517	1 621 111	•	30 202 516	72 586 158	1 : 1 133
`wo-road-rail-flight`	31 689	1 649 371	•	50 139 663	124 625 598	1 : 1 846
`de-road-rail(long)`	498	16 450	○	5 055 680	12 378 224	1 : 10 711
`wo-road-flight`	1 172	28 260	○	50 139 663	124 625 598	1 : 139 277

need to run the first phase on those components that are represented by an initial or final state in the input automaton $\mathcal{A}$.

Combining all improvements yields a speedup of up to factor 4.9. (Section 5.1.5.5 of the experimental evaluation will show detailed figures.)

5.1.5 *Experiments*

We conducted our experiments on an Intel Xeon E5430 processor clocked at 2.66 GHz with 32 GiB of RAM and 12 MiB of L2 cache. The program was compiled with GCC 4.5, using optimization level 3. Our implementation is written in C++ using the STL and Boost. We use our own custom implementations for most data structures. In particular, we represent graphs as adjacency arrays, and as a priority queue we use a 4-ary heap. All runs are sequential for comparison.

INPUTS. We assemble a total of six multimodal networks where two are imported from [DPW09a]. Their size figures are reported in Table 5.1. For `ny-road-rail`, we combine New York's foot network with the public transit network operated by MTA [Met66]. We link bus and subway stops to road intersections that are no more than 500 m apart. The `de-road-rail` network combines the pedestrian and railway networks of Germany. The railway network is extracted from the timetable of the winter period 2000/01. It includes short and long distance trains, and we link stations using a radius of 500 m. The `europe-road-rail` network combines the road (as in car) and railway networks of Western Europe. The railway network is extracted from the timetable of the winter period 1996/97 and stations are linked within 5 km. The `wo-road-rail-flight` network is a combination of

the road networks of North America and Western Europe with the railway network of Western Europe and the flight network of Star Alliance and One World. The flight networks are extracted from the winter timetable 2008. As radius we use 5 km.

Both `de-road-rail(long)` and `wo-road-flight` are from [DPW09a]. The data of the Western European and North American road networks (thus Germany and New York) was kindly provided to us by PTV AG [PTV79] for scientific use. The timetable data of New York is publicly available through General Transit Feeds [Gtfs], while the data of the German and European railway networks was kindly provided by HaCon [HaC84]. Unlike the data from HaCon, the New York timetable did not contain any foot path data for short transfers between nearby stops (as typically defined by the operator). Thus, we generated artificial foot paths with a known heuristic [DKP12].

Our instances vary in the fractional size of their public transit subnetwork with respect to the total network size. We call the fraction of linked vertices in a subgraph *density* (see last column of Table 5.1). Our densest network is `ny-road-rail`, which also has the highest number of connections. On the other hand, `de-road-rail(long)` and `wo-road-flight` are rather sparse. However, we include them to compare our algorithm to Access Node Routing (ANR). Note that we take the figures for ANR from [DPW09a]. Since they used a different machine, we scale the running time figures by comparing the running time of Dijkstra's algorithm on our machine to theirs. Also note that for comparison we do not use the improved coloring model (see Section 5.1.1.1) on these two instances.

We use the following automata as query input. The `foot-and-rail` automaton allows either walking, or walking, taking the railway network and walking again. Similarly, the `car-and-rail` automaton uses the road network instead of walking, while the `car-and-flight` automaton uses the flight network instead of the railway network. The `hierarchical` automaton is our most complicated automaton. It hierarchically combines road, railways and flights (in this order). All modal sequences are possible, except of going up in the hierarchy after once stepping down. For example, if one takes a train after a flight, it is impossible to take another flight. Note that completely disallowing walking is not reasonable. Instead, taking the predefined (by the timetable) transfer foot paths within the `rail` (`flight`) model is always allowed within the `rail` (`flight`) state. Finally, the `everything` automaton allows arbitrary modal sequences in any order. See Figure 5.1 for transition graphs of `foot-and-rail` and `hierarchical`.

METHODOLOGY. We evaluate both preprocessing and query performance. The contraction order is always computed according to the aggressive variant from [GSSV12]. We report the time and the amount of computed auxiliary data. Queries are generated with source, target

Table 5.2: Comparing preprocessing performance of UCCH on `de-road-rail` with varying average core degree limit. For queries we use the `foot` automaton. We also report numbers for unconstrained unimodal CH and partial CH (PCH).

	Preprocessing				Query			
Algorithm	Avg. Core-Degree	Core-Vertices	Shortcut-Edges	Time [min]	Settled Vertices	Relaxed Edges	Touched Edges	Time [ms]
UCCH	10	30 908	42.3 %	6	15 531	27 506	155 776	5.85
	15	16 003	43.1 %	7	8 090	16 844	121 631	3.11
	20	12 239	43.7 %	9	6 240	14 425	124 201	2.82
	25	10 635	44.2 %	10	5 465	13 687	135 151	2.80
	30	9 742	44.7 %	12	5 049	13 486	148 735	2.96
	35	9 171	45.1 %	14	4 794	13 598	163 376	3.15
	40	8 788	45.4 %	15	4 628	13 787	179 483	3.38
PCH	13	10 635	41.7 %	6	5 567	11 402	71 860	1.93
PCH	15	6 750	41.8 %	7	3 636	7 970	53 655	1.37
CH	—	0	41.8 %	9	677	1 290	11 434	0.25

vertices and departure times uniformly picked at random. For Dijkstra we run 1,000 queries, while for UCCH we run a superset of 100,000 queries. We report the average number of: (1) extracted vertices in the implicit product graph from the priority queue (settled vertices), (2) priority queue update operations (relaxed edges), (3) touched edges, (4) the average query time, and (5) the speedup over Dijkstra. Note that we only report the time to compute the length of the shortest path. Unpacking of shortcuts can be done efficiently in less than a millisecond [GSSV12].

5.1.5.1 *Evaluating Average Core Degree Limit*

The first experiment evaluates preprocessing and query performance with varying average core degree. We abort contraction as soon as the average vertex degree in the core exceeds a limit α. In our implementation we compute the average vertex degree by dividing the number of edges by the number of vertices in our graph data structure. Note that we use *edge compression* [Delo9]: Whenever there are edges $e = (u, v)$ and $e' = (v, u)$ where $\text{len}(e) = \text{len}(e')$, we combine both edges in a single entry at u and v. As a result, the number we report may be smaller than the true average degree (at most by a factor of 2) which is, however, irrelevant for the result of this experiment.

Table 5.2 shows preprocessing and query figures on `de-road-rail`. For queries we use the `foot` automaton, which does not use public transit edges. With higher values of α more vertices are contracted,

resulting in higher preprocessing time and more shortcuts (we report them as a fraction of the input's size). At the same time, less vertices (but with higher degree) remain in the core. Setting $\alpha = \infty$ is infeasible. The amount of shortcuts is too large, and preprocessing does not finish within reasonable time. Interestingly, the query time decreases (with smaller core size) up to $\alpha \approx 25$ and then increases again. Though we settle less vertices, the increase in shortcuts results in more touched edges during query, that is, edges the algorithm has to iterate when settling a vertex. We conclude that for `de-road-rail` the trade-off between number of core vertices and added shortcut edges is optimal for $\alpha = 25$. Hence, we use this value in subsequent experiments. Accordingly, we determine α for all instances.

COMPARISON TO UNIMODAL CH. In Table 5.2 we also compare UCCH to CH when run on the unimodal road network. Computing a full hierarchy results in queries that are faster by a factor of 11.2. Since UCCH does not compute a full hierarchy by design, we evaluate two partial CH hierarchies: The first stops when the core reaches a size of 10,635—equivalent to the optimal core size of UCCH. We observe a query performance almost comparable to UCCH (slightly faster by 45 %). The second partial hierarchy stops with a core size of 6,750 which is equal to the number of transfer vertices in the network (i. e., the smallest possible core size on this instance for UCCH). Here, CH is a factor of 2 faster than UCCH. Recall that UCCH must not contract transfer vertices. In road networks these are usually unimportant: Long-range queries do not pass many railway stations or bus stops in general, which explains that UCCH's hierarchy is less pronounced. However, for *multimodal* queries transfer vertices are indeed very important, as they constitute the interchange points between different networks. To enable arbitrary automata during query, we overestimate their importance by not contracting them at all, which is reflected by the (relatively small) difference in performance compared to CH.

5.1.5.2 *Preprocessing*

Table 5.3 shows preprocessing figures for UCCH on all our instances. Besides the average degree we evaluate the core in terms of total and fractional number of core vertices, and the amount of added shortcuts. Added shortcuts are reported as percentage of all road edges and in total MiB. We observe that the preprocessing effort correlates with the graph size. On the small `ny-road-rail` instance it takes less than a minute and produces 8 MiB of data. On our largest instance, `wo-road-rail-flight`, we need 1.5 hours and produce 558 MiB of data. Because the size of the core depends on the size of the public transportation network, we obtain a much higher ratio of core vertices on `ny-road-rail` (1 : 52) than we do, for example, on `wo-road-rail-flight` (1 : 1,298).

Table 5.3: Preprocessing figures for UCCH and Access-Node Routing on the road subnetwork. Figures for the latter are taken from [DPW09a], which were obtained on a different machine. We thus scale the preprocessing time with respect to running time figures compared to Dijkstra.

	UCCH						ANR	
Network	Avg. Core-Degree	Core Vertices Total	Core Vertices Ratio	Shortcuts Percent	Shortcuts [MiB]	Time [min]	Space [MiB]	Time [min]
`ny-road-rail`	8	11 057	1:52	48.3 %	8	< 1	—	—
`de-road-rail`	25	10 635	1:475	44.2 %	63	10	—	—
`europe-road-rail`	25	39 665	1:761	39.0 %	324	38	—	—
`wo-road-rail-flight`	30	38 610	1:1 298	39.1 %	558	87	—	—
`de-road-rail(long)`	35	996	1:5 075	42.3 %	60	10	504	26
`wo-road-flight`	35	727	1:68 967	38.0 %	542	78	14 050	184

Comparing the preprocessing effort of UCCH to scaled figures of Access-Node Routing (ANR), we observe that UCCH is more than twice as fast and produces significantly less amount of data: on `de-road-rail(long)` by a factor of 8.4, on `wo-road-flight` by a factor of 26. Here, ANR requires 14 GiB of space, whereas UCCH only uses 542 MiB. Concluding, UCCH outperforms ANR in terms of preprocessing space and time.

5.1.5.3 *Query Performance*

In this experiment we evaluate the query performance of UCCH and compare it to Dijkstra and ANR (where figures are available). Results are presented in Table 5.4. We observe that we achieve speedups of several orders of magnitude over Dijkstra, depending on the instance. Generally, UCCH's speedup over Dijkstra correlates with the ratio of core vertices after preprocessing (thus, indirectly with the density of transfer vertices): the sparser our networks are interconnected, the higher the speedups we achieve. On our densest network, `ny-road-rail`, we have a speedup of 17, while on `wo-road-flight` we achieve query times of less than a millisecond—a speedup of over 50,540. To further highlight how the density of the network affects the speedup, Figure 5.3 plots the speedup of UCCH on each instance subject to its density. Note that most of the time is spent inside the core (particularly, in the public transit network), which we do not accelerate. Section 5.1.5.6 contains a detailed query time distribution analysis. Comparing UCCH to ANR, we observe that query times are in the same order of magnitude, UCCH being slightly faster. Note that we achieve this with significantly less preprocessing effort.

Table 5.4: Query performance of UCCH compared to plain multimodal Dijkstra and Access-Node Routing. Figures for the latter are taken from [DPW09a], which were obtained on a different machine. We thus scale the running time with respect to Dijkstra.

		Dijkstra		ANR			UCCH		
Network	Automaton	Settled Vertices	Time [ms]	Settled Vertices	Time [ms]	Speed-Up	Settled Vertices	Time [ms]	Speed-Up
ny-road-rail	foot-and-rail	404 816	226	—	—	—	25 525	13.61	17
de-road-rail	foot-and-rail	2 611 054	2 005	—	—	—	18 275	12.78	157
europe-road-rail	car-and-rail	30 021 567	23 993	—	—	—	90 579	53.78	446
wo-road-rail-flight	car-and-flight	36 053 717	33 692	—	—	—	42 056	26.72	1 260
wo-road-rail-flight	hierarchical	36 124 105	35 261	—	—	—	126 072	70.52	500
wo-road-rail-flight	everything	25 267 202	23 972	—	—	—	71 389	50.77	472
de-road-rail(long)	foot-and-rail	2 735 426	2 075	13 524	3.45	602	12 509	3.13	663
wo-road-flight	car-and-flight	36 582 904	33 862	4 200	1.07	31 551	1 647	0.67	50 540

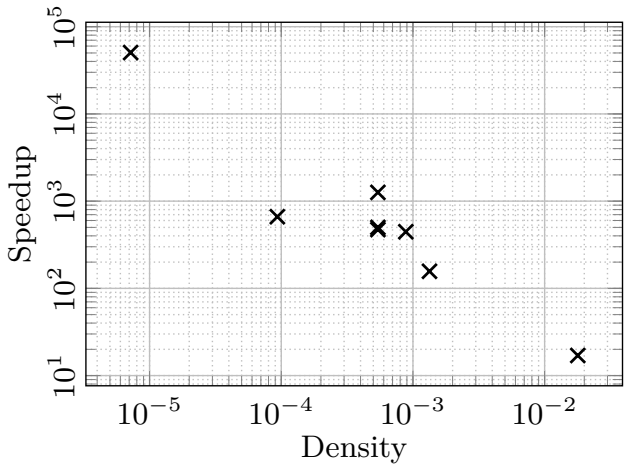


Figure 5.3: Evaluating the speedup of UCCH from Table 5.4 subject to the density of the input from Table 5.1.

5.1.5.4 *Detailed path properties*

Table 5.5 reports the impact of integrating modal sequence constraints on the paths output by the algorithm. It does so by evaluating three main figures: The percentage of the total number of paths that utilize a certain transportation mode (foot, car, rail with transfers, and flight with transfers), the average and maximum number of interchanges between transportation modes along the journeys, and the average and maximum factor by which the travel time increases when mode sequence constraints are enabled. Note that for the latter, we only count paths that actually differ from the unconstrained one, additionally reporting the amount of paths where mode sequence constraints have no impact (Ident.). Each instance in Table 5.5 is evaluated on both an appropriate constrained automaton as well as the `everything` automaton, which corresponds to running unrestricted queries.

We observe that on `ny-road-rail` 57 % of the paths utilize the rail network, regardless whether we constrain paths by the `foot-and-rail` automaton. However, 36 % of the paths are indeed different, and enabling constraints reduces the average number of modal interchanges by a factor of almost four with only a 7 % increase in travel time. Figures for `de-road-rail` are similar: All paths use the rail network, and enabling constraints reduces the number of modal interchanges by a factor of almost 3.5 with only little increase in travel time. On our sparser long-distance networks the effects are less pronounced. For example, on `wo-road-rail-flight`, we see that 89 % of the paths already follow a hierarchical use of transportation modes, and the difference in the number of modal interchanges decreases only by 0.4. However, while this difference may seem small, we argue that model constraints are nevertheless important, since our experiment shows

Table 5.5: Evaluating the impact of integrating modal sequence constraints on the paths.

		Mode Used in % Paths				# Modal Changes		Stretch		
Network	Automaton	Foot	Car	Rail	Flight	Avg.	Max.	Avg.	Max.	Ident. [%]
`ny-road-rail`	`everything`	100	—	57	—	4.1	22	—	—	—
`ny-road-rail`	`foot-and-rail`	100	—	57	—	1.1	2	1.07	2.83	64
`de-road-rail`	`everything`	100	—	100	—	6.8	24	—	—	—
`de-road-rail`	`foot-and-rail`	100	—	100	—	2.0	2	1.08	2.94	87
`europe-road-rail`	`everything`	—	100	41	—	1.2	10	—	—	—
`europe-road-rail`	`car-and-rail`	—	100	41	—	0.8	2	1.03	1.46	92
`wo-road-rail-flight`	`everything`	—	100	13	85	2.2	12	—	—	—
`wo-road-rail-flight`	`hierarchical`	—	100	9	85	1.8	4	1.08	2.25	89
`wo-road-rail-flight`	`car-and-flight`	—	100	—	85	1.7	2	1.06	2.34	84

Table 5.6: Detailed analysis of the impact on query performance by our improvements (cf. Section 5.1.4). We show figures for reordering vertices (rn), reordering edges (re), improved bidirectional search (bi), only forward search on the core (fo), state-independent search on component (si), and state-pruning (sp)

Network	Automaton	Improv.	Settled vert.	Time [ms]	Spd. up
europe-road-rail	car	none	48 488	69.93	—
		rn	48 488	35.11	2.00
		rn,re	48 488	29.38	2.38
		rn,re,bi	31 628	20.02	3.49
		rn,re,fo	24 297	14.57	4.80
wo-road-rail-flight	car	none	35 539	54.42	—
		rn	35 539	27.93	1.95
		rn,re	35 539	23.18	2.35
		rn,re,bi	29 695	19.84	2.74
		rn,re,fo	17 862	11.50	4.73
europe-road-rail	car-and-rail	rn,re,fo	95 095	57.23	—
		rn,re,fo,si	95 024	60.12	0.95
		rn,re,fo,sp	89 770	51.72	1.11
		rn,re,fo,si,sp	89 699	54.45	1.05
wo-road-rail-flight	car-and-rail	rn,re,fo	72 997	46.73	—
		rn,re,fo,si	72 895	49.09	0.95
		rn,re,fo,sp	69 627	42.35	1.10
		rn,re,fo,si,sp	69 525	44.51	1.05

that in 11 % of the cases the (unconstrained) path violates the modal constraints, which may render it completely infeasible to the user.

5.1.5.5 *Improvements*

In Table 5.6 we report figures for the improvements to UCCH described in Section 5.1.4. The table is divided into two parts. The upper part addresses unimodal improvements that are also applicable to (partial) CH. Therefore, we evaluate them using the `car` automaton. For our two biggest networks, we provide the number of settled vertices and the query time for several combinations of improvements. The first row (none) reports results for the basic version of UCCH. The other rows use: Reordered vertices (rn), reordered edges (re), improved bi-directional search on the core (bi), and uni-directional search on the core (fo), that is, no backward search is performed on the core. Combining these techniques, we obtain a speedup of up to factor 4.8.

Table 5.7: In-depth analysis of UCCH's query time. We report the distribution of query time among the particular subnetworks and compare it to Dijkstra.

			Dijkstra		UCCH		
Network	Automaton	Subgraph	Settled vert.	Time [ms]	Settled vert.	Time [ms]	Spd. up
ny-road-rail	foot-and-rail	road-comp.	—	—	203	≈ 0.0	—
		road-core	389 578	215.5	9 944	4.8	45
		rail	15 238	10.5	15 238	8.8	1.2
de-road-rail	foot-and-rail	road-comp.	—	—	188	≈ 0.0	—
		road-core	2 599 251	1 988.4	6 314	5.0	397
		rail	11 803	16.6	11 803	7.8	2.1
europe-road-rail	car-and-rail	road-comp.	—	—	213	≈ 0.0	—
		road-core	29 973 817	23 933.3	43 017	24.4	982
		rail	47 750	59.7	47 750	29.4	2.0
wo-road-rail-flight	hierarchical	road-comp.	—	—	301	≈ 0.0	—
		road-core	36 047 522	35 169.3	49 944	30.6	1 149
		rail	75 682	89.9	75 682	39.2	2.3
		flight	902	1.8	902	0.7	2.6

The lower part of Table 5.6 is dedicated to improvements for UCCH which we evaluate using the `car-and-rail` automaton. We provide numbers for state-independent search on the component (si) and state-pruning (sp). Note that these figures already include the previous improvements. Interestingly, using state-independent search results in slightly worse query times of about 5 %. However, we reduce the memory footprint of the algorithm by a significant amount since we store distance values only once per component vertex. Maintaining distance labels on the implicit product graph requires between 6.9 MiB and 1341.2 MiB on our instances. When (si) is enabled, these numbers are reduced to 2.4 MiB and 192.1 MiB, respectively. This is an improvement of up to factor 7.

Note that from the number of settled vertices we can deduce which of the improvements impact cache efficiency and which impact the search space.

5.1.5.6 *In-Depth Analysis of Query Performance*

Table 5.7 reports in-depth figures for the UCCH query including all (reasonable) improvements from the previous section. We see that a large fraction of the query is spent on the public transportation part of the multimodal network: Up to 65 % of the settled vertices and

also up to 65 % of query time. Recall that we do not further accelerate the search on the core. Interestingly, UCCH is slightly faster (up to a factor of 2.6) on the timetable subnetworks when compared to Dijkstra. UCCH settles fewer vertices in total, which helps cache performance on the public transit part. When we compare the time spent on the road network (component and core) of `de-road-rail` with the figures of Table 5.2 (where we use the same instance but with the smaller `foot` automaton), we observe that the `foot-and-rail` automaton yields a factor 1.8 slowdown. The reason is that the `foot-and-rail` automaton actually has two "foot-states" (cf. Figure 5.1) and, thus, has to do twice the work on the road subnetwork. Note that the number 1.8 (instead of exactly 2) stems from the fact that we apply state pruning.

Conclusions

We have introduced UCCH: The first, fast multimodal speedup technique that handles arbitrary modal sequence constraints at *query time*—a problem considered challenging before. Besides not determining the modal constraints during preprocessing, its advantages are small space overhead, fast preprocessing time and the ability to implicitly handle local queries without the need for a separate algorithm. Its preprocessing can handle huge networks of intercontinental size with many more stations and airports than those of previous multimodal techniques.

For future work, we are interested in augmenting our approach to more general scenarios. For example, the computation of multimodal profile queries would produce journeys whose departure time follows the timetable more closely. We would also like to further accelerate search on the uncontracted core—especially on the rail networks. Finally, we are interested to improve the contraction order. In particular, we would like to use ideas from Access Node Routing [DPW09a] to achieve better results on more densely interlinked networks: It should be possible to contract a transfer node as soon as all road nodes, for which it is an access node, have been contracted. Furthermore, UCCH can be interpreted as a point-of-interest (POI) technique [ADF+12; DW14; EPV15; FWL12; Gei11]: Instead of computing buckets with distances to relevant POIs, we simply keep the POIs (i. e., transit stops) in the uncontracted core. This could be interesting to evaluate for general POIs.

5.2 MULTICRITERIA MULTIMODAL JOURNEY PLANNING

In the previous section, we required paths to obey a user-defined pattern (often given as regular expressions) for enforcing a hierarchy of modes [BBM06; YL12] (such as "no car travel between trains"). The main advantage of this strategy is that preprocessing techniques developed for road networks carry over [BBH+09; DPW09a; DPW15; KLC12; KLPC11]. This approach, however, can hide interesting journeys (for example, taking a taxi between train stations in Paris may be an option). In fact, this exposes a fundamental conceptual problem with label-constrained optimization: It essentially relies on the user to know her options before planning the journey.

Given the limitations of current approaches, we consider the problem of finding *multicriteria multimodal* journeys on a metropolitan scale. Instead of optimizing each mode of transportation independently [EL11], we argue in Section 5.2.1 that most users optimize three criteria: travel time, convenience, and costs. As this produces a large Pareto set, we propose using fuzzy logic [FA04; Zad88] to identify, in a principled way, a modest-sized subset of representative journeys. This postprocessing step is very quick and can incorporate personal preferences. As Section 5.2.2 shows, we can use recent algorithmic developments [DPW14; DPW15; GSSV12] to answer exact queries optimizing time and convenience in less than two seconds within a large metropolitan area, for the simpler scenario of walking, cycling, and public transit. Unfortunately, this is not enough for interactive applications, and becomes much slower when more criteria, such as costs, are incorporated. We therefore also propose (in Section 5.2.3) heuristics (still multicriteria) that are significantly faster, and closely match the top journeys in the Pareto set. Section 5.2.4 presents a thorough experimental evaluation of all algorithms in terms of both solution quality and performance, and shows that our approach can be fast enough for practicable applications.

5.2.1 *Preliminaries*

Throughout Section 5.2, we use the following notation and concepts:

We want to find journeys in a network built from several *partial networks*. The first is a *public transportation network* representing all available schedule-based means of transportation, such as trains, buses, rail, or ferries. We can specify this network in terms of its timetable, which is defined as follows. A *stop* is a location in the network (such as a train platform or a bus stop) at which a user can board or leave a particular vehicle. A *route* is a fixed sequence of stops for which there is scheduled service during the day; a typical example is a bus or subway line. A route is served by one or more distinct *trips* during the day; each trip is associated with a unique vehicle, with fixed (scheduled)

arrival and departure times for every stop in the route. Each stop may also keep a *minimum change time*, which must be obeyed when changing trips.

Besides the public transportation network, we also take as input several *unrestricted networks*, with no associated timetable. Walking, cycling, and driving are modeled as distinct unrestricted networks, each represented as a directed graph $G = (V, A)$. Each vertex $v \in V$ represents an intersection and has associated coordinates (latitude and longitude). Each arc $(v, w) \in A$ represents a (directed) road segment and has an associated *duration* $\mathrm{dur}(v, w)$, which corresponds to the (constant) time to traverse it.

The *integrated transportation network* is the union of these partial networks with appropriate *link vertices*, i. e., vertices (or stops) in different networks are identified with one another to allow for changes in modes of transportation. Note that, unlike previous work [BCE+10; DKP12; DMS08; DPW14; PSWZ08], we do not necessarily require explicit *footpaths* in the public transportation networks (to walk between nearby stops). For pure public transport optimization, adding these footpaths is often done by the operator of the network or by heuristics [DKP12].

A query takes as input a *source location* s, a *target location* t, and a *departure time* τ, and it produces *journeys* that leave s no earlier than τ and arrive at t. A *journey* is a valid path in the integrated transportation network that obeys all timetable constraints.

5.2.1.1 *Criteria.*

We still have to define *which* journeys the query should return. We argue that users optimize three natural criteria in multimodal networks: arrival time, costs, and "convenience". For our first (simplified) scenario (with public transit, cycling, and walking, but no taxi), we work with three criteria. Besides arrival time, we use number of trips and walking duration as proxies for convenience. We add cost for the scenario that includes taxi.

Given this setup, a first natural problem we need to solve is the *full multicriteria problem*, which must return a full (maximal) Pareto set of journeys. We say that a journey J_1 *dominates* J_2 if J_1 is strictly better than J_2 according to at least one criterion and no worse according to all other criteria. A *Pareto set* is a set of pairwise nondominating journeys [Han79; MW01]. If two journeys have equal values in all criteria, we only keep one.

5.2.1.2 *Fuzzy Dominance.*

Solving the full multicriteria problem, however, can lead to solution sets that are too large for most users. Moreover, many solutions provide undesirable tradeoffs, such as journeys that arrive much later to

save a few seconds of walking (or walk much longer to save a few seconds in arrival time). Intuitively, most criteria are diffuse to the user, and only large enough differences are significant. Pareto optimality fails to capture this.

To formalize the notion of significance, we propose to *score* the journeys in the Pareto set in a post-processing step using concepts from fuzzy logic [Zad88] (and fuzzy set theory [Zad65]). Loosely speaking, fuzzy logic generalizes Boolean logic to handle (continuous) degrees of truth. For example, the statement "60 and 61 seconds of walking are equal" is false in classical logic, but might be considered "almost true" in fuzzy logic. Formally, a *fuzzy set* is a tuple $\mathcal{S} = (\mathcal{U}, \mu)$, where $\mathcal{U}$ is a set and $\mu\colon \mathcal{U} \to [0,1]$ a *membership function* that defines "how much" each element in $\mathcal{U}$ is contained in $\mathcal{S}$. Mostly, we use μ to refer to $\mathcal{S}$. Our application requires fuzzy relational operators $\mu_<$, $\mu_=$, and $\mu_>$. For any $x, y \in \mathbb{R}$, they are evaluated by $\mu_<(x-y)$, $\mu_>(y-x)$, and $\mu_=(x-y)$. We use the well-known [Zad88] exponential membership functions for the operators: $\mu_=(x) := \exp(\frac{\ln(\chi)}{\varepsilon^2}x^2)$, where $0 < \chi < 1$ and $\varepsilon > 0$ control the degree of fuzziness. The other two operators are derived by $\mu_<(x) := 1 - \mu_=(x)$ if $x < 0$ (0 otherwise) and $\mu_> := 1 - \mu_=(x)$ if $x > 0$ (0 otherwise). A *triangular norm* (short: *t-norm*) $T\colon [0,1]^2 \to [0,1]$ is a commutative, associative, and monotone (i.e., $a \leqslant b, x \leqslant y \Rightarrow T(a,x) \leqslant T(b,y)$) binary operator to which 1 is the neutral element. If $x, y \in [0,1]$ are truth values, $T(x,y)$ is interpreted as a fuzzy conjunction (*and*) of x and y. Given a t-norm T, the *complementary conorm* (or *s-norm*) of T is defined as $S(x,y) := 1 - T(1-x, 1-y)$, which we interpret as a fuzzy disjunction (*or*). Note that the neutral element of S is 0. Two well-known pairs of t- and s-norms are $(\min(x,y), \max(x,y))$, called *minimum/maximum norms*, and $(xy, x+y-xy)$, called *product norm/probabilistic sum*.

We now recap the concept of fuzzy dominance in multicriteria optimization, which is introduced by Farina and Amato [FA04]. Given journeys J_1 and J_2 with M optimization criteria, we denote by $n_b(J_1, J_2)$ the (fuzzy) number of criteria in which J_1 is better than J_2. More formally, $n_b(J_1, J_2) := \sum_{i=1}^{M} \mu_<^i(\kappa^i(J_1), \kappa^i(J_2))$, where $\kappa^i(J)$ evaluates the i-th criterion of J and $\mu_<^i$ is the i-th fuzzy less-than operator. (Note that each criterion may use different fuzzy operators.) Analogously, we define $n_e(J_1, J_2)$ for equality and $n_w(J_1, J_2)$ for greater-than. By definition, $n_b + n_e + n_w = M$. Hence the Pareto dominance can be generalized to obtain a *degree of domination* $d(J_1, J_2) \in [0,1]$, defined as $(2n_b + n_e - M)/n_b$ if $n_b > (M - n_e)/2$ (and 0 otherwise). Here, $d(J_1, J_2) = 0$ means that J_1 does not dominate J_2, while a value of 1 indicates that J_1 Pareto-dominates J_2. Otherwise, we say J_1 *fuzzy-dominates* J_2 by degree $d(J_1, J_2)$. Figure 5.4 shows contour lines for values of d between 0 and 1 when using the maximum norm and two exemplary criteria: arrival time and walking duration (with fuzziness parameters set as in Section 5.2.4). In the figure we fix the criteria

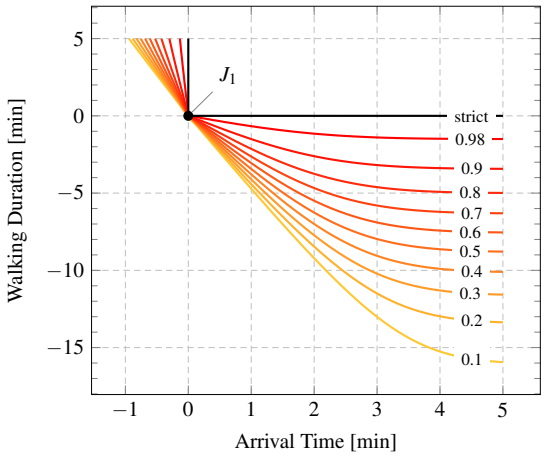


Figure 5.4: Contour lines of the fuzzy dominance function $d(J_1, J_2) = t$ for different values t and a fixed journey $J_1 = (0, 0)$ when considering two exemplary criteria: arrival time and walking duration. The thick black line marks the classical Pareto-dominance ($t = 1$).

of J_1 to $(0, 0)$. The area right-above each contour line t then contains all journeys J_2 (with respective values for their criteria) which are dominated by J_1 with degree at least t. For example, a journey is still dominated by J_1 with degree 0.4 if it has 10 minutes less walking while arriving 5 minutes later.

Now, given a (Pareto) set $\mathcal{J}$ of n journeys $J_1, \ldots, J_n$, we define a *score function* $sc\colon \mathcal{J} \to [0, 1]$ that computes the degree of domination by the whole set for each J_i. More precisely, $sc(J) := 1 - S(J_1, \ldots, J_n)$. Note that if we set S to be the maximum norm, the score is based on the (one) journey that dominates J most. On the other hand, with the probabilistic sum the score may be based on several fuzzily dominating journeys.

We finally use the score to order the journeys by significance. One may then decide to only show the top k journeys with highest score to the user. Figure 5.5 shows an example of such a score-based filtering. a (quite representative) location-to-location query from William Road (near Warren Street Station) to Caxton Street (near Westminster Abbey) on our London instance using public transit, walking, and taxi with optimization criteria arrival time, number of transfers, walking duration, and cost (in pounds). The departure time is 4:27 pm. The left figure shows all nondominating journeys of the full Pareto set (there are 65 in total), while the right figure shows the three journeys with highest score from the (same) Pareto set, when our fuzzy dominance approach is usedThis example clearly demonstrates that we obtain too many nondominating solutions (left figure), a known problem for multicriteria search. But not only is the number of solutions too high for presentation to a user, in fact, most of the journeys are not meaningful. Some of them take considerable detours (for example north of the source location), just to save some (insignificant) amount of walking.

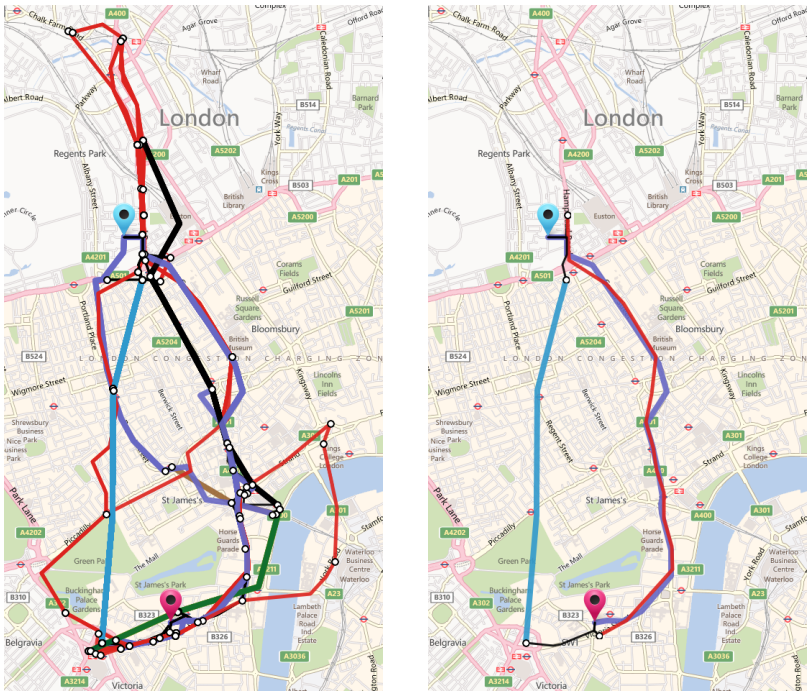


Figure 5.5: Exemplary multicriteria multimodal query on London with criteria arrival time, number of transfers, walking duration, and cost. The left figure shows the full Pareto set (65 journeys), while the right figure shows the three journeys with highest score (cf. Section 5.2.1). Each dot represents a transfer and included transportation modes are walking (thin black), taxi (thick purple), buses (thin red), and tube (other thick colors).

In contrast, our scoring approach by fuzzy domination (right figure) is able to identify the significant solutions in the Pareto set, resulting in three meaningful journeys: One taking taxi the full way (purple), one taking the subway (blue) which is faster at the cost of more walking (black), and one taking the bus (red) which takes longer but with significantly less total walking (4 min instead of 14 min).

5.2.2 *Exact Algorithms*

This section considers exact algorithms for the multicriteria multimodal problem. Sections 5.2.2.1 and 5.2.2.2 propose two solutions, each building on a different algorithm for multicriteria optimization on public transportation networks (MLC [PSWZ08] and RAPTOR [DPW14]). Section 5.2.2.3 then describes an acceleration technique that applies to both. To simplify the discussion (and notation), we first describe the algorithms in terms of our simplest scenario, considering only the (timetable-based) public transit network and the (unrestricted) walking network. Section 5.2.2.4 explains how to handle cycling and taxis, which are unrestricted but have special properties.

5.2.2.1 *Multi-label-correcting Algorithm*

Traditional solutions to the multicriteria problem on public transportation networks typically model the timetable as a graph [BDGM09; DKP12; Gei10; MSWZ07]. A particularly effective approach is to use the *time-dependent route model* [MSWZ07]. For each stop p, we create a single *stop vertex* linked by time-independent *transfer edges* to multiple *route vertices*, one for each route serving p. We also add *route edges* between route vertices associated to consecutive stops within the same route. To model the trips along a route, the cost of a route edge is given by a piecewise linear function reflecting the traversal time (including waiting for the next departure).

A journey in the public transportation network corresponds to a path in this graph. The *multi-label-correcting* (MLC) [MSWZ07] algorithm uses this to find full Pareto sets for arbitrary criteria that can be modeled as edge costs. MLC extends Dijkstra's algorithm [Dij59] by operating on labels that have multiple values, one per criterion. Each vertex v maintains a *bag* $B(v)$ of nondominated labels. In each iteration, MLC extracts from a priority queue the minimum (in lexicographic order) unprocessed label $L(u)$. For each arc (u, v) out of the associated vertex u, MLC creates a new label $L(v)$ (by extending $L(u)$ in the natural way) and inserts it into $B(v)$; newly-dominated labels (possibly including $L(v)$ itself) are discarded, and the priority queue is updated if needed. MLC can be sped up with target pruning and by avoiding unnecessary domination checks [DMS08].

To solve the multimodal problem, we extend MLC: It suffices to augment its input graph to include the walking network. We combine the original graphs by merging (public transportation) stops and (walking) intersections that share the same location (and keeping all edges). These *link vertices* are then used to switch between modes of transportation. The MLC query remains essentially unchanged, and still processes labels in lexicographic order. Although labels can now be associated to vertices in different networks, they can all share the same priority queue.

5.2.2.2 *Round-based Algorithm*

A drawback of MLC (even restricted to public transportation networks) is that it can be quite slow: Unlike Dijkstra's algorithm, MLC may scan the same vertex multiple times (the exact number depends on the criteria being optimized), and domination checks make each such scan quite costly. Delling et al. [DPW14] have recently introduced RAPTOR (*Round bAsed Public Transit Optimized Router*) as a faster alternative. The simplest version of the algorithm optimizes two criteria: arrival time and number of transfers. Unlike MLC, which searches a graph, RAPTOR uses dynamic programming to operate directly on the timetable. It works in rounds, with round i processing

all relevant journeys with exactly $i-1$ transfers. It maintains one label per round i and stop p representing the best known arrival time at p for up to i trips. During round i, the algorithm processes each *route* once. It reads arrival times from round $i-1$ to determine relevant trips (on the route) and updates the labels of round i at every stop along the way. Once all routes are processed, the algorithm considers potential transfers to nearby (predefined) stops in a second phase. Simpler data structures and better locality make RAPTOR an order of magnitude faster than MLC. Delling et al. [DPW14] have also proposed McRAPTOR, which extends RAPTOR to handle more criteria (besides arrival times and number of transfers). It maintains a *bag* (set) of labels with each stop and round.

Even with multiple modes of transport available, one trip always consists of a single mode. This motivates adapting the round-based paradigm to our scenario. We propose MCR (*multimodal multicriteria RAPTOR*), which extends McRAPTOR to handle multimodal queries. As in McRAPTOR, each round has two phases: the first processes trips in the public transportation network, while the second considers arbitrary paths in the unrestricted networks. We use a standard McRAPTOR round for the first phase (on the timetable network) and MLC for the second (on the walking network). Labels generated by one phase are naturally used as input to the other. During the second phase, MLC extends bags instead of individual labels. To ensure that each label is processed at most once, we keep track of which labels (in a bag) have already been extended. The initialization routine (before the first round) runs Dijkstra's algorithm on the walking network from the source s to determine the fastest walking path to each stop in the public transportation network (and to t), thus creating the initial labels used by MCR. During round i, the McRAPTOR subroutine reads labels from round $i-1$ and writes to round i. In contrast, the MLC subroutine may read and write labels of the same round if walking is not regarded as a trip.

5.2.2.3 *Contracting the Unrestricted Networks*

As our experiments will show, the bottleneck of the multimodal algorithms is processing the walking network $G = (V, A)$. We improve performance using ideas for quick preprocessing from Section 5.1. For any journey involving public transportation, walking between trips always begins and ends at the restricted set $K \subset V$ of link vertices. During queries, we must only be able to compute the pairwise distances between these vertices. We therefore use preprocessing to compute a smaller *core graph* [SWW00] that preserves these distances. More precisely, we start from the original graph and iteratively *contract* [GSSV12] each vertex in $V \setminus K$ in the order given by a rank function rank. Each contraction step (temporarily) removes a vertex and adds shortcuts between its uncontracted neighbors to maintain

shortest path distances (if necessary). It is usually advantageous to first contract vertices with relatively small degrees that are evenly distributed across the network [GSSV12]. Recall from Section 5.1, that we stop contraction when the average degree in the core graph reaches some threshold (we use 12 in our experiments).

To run a faster multimodal s–t query, we use essentially the same algorithm as before (based on either MLC or RAPTOR), but replacing the full walking network with the (smaller) core graph. Since the source s and the target t may not be in the core, we handle them during initialization. It works on the graph $G^+ = (V, A \cup A^+)$ containing all original arcs A as well as all shortcuts A^+ added during the contraction process. We run upward searches (only following arcs (u, v) such that $\text{rank}(u) > \text{rank}(w)$) in G^+ from s (scanning forward arcs) and t (scanning reverse arcs); they reach all potential entry and exit points of the core, but arcs within the core are not processed (cf. Section 5.1). These core vertices (and their respective distances) are used as input to MCR's (or MLC's) standard initialization, which can operate on the core from this point on.

The main loop works as before, with one minor adjustment. Whenever MLC extracts a label $L(v)$ for a scanned core vertex v, we check if it has been reached by the reverse search during initialization. If so, we create a temporary label $L'(t)$ by extending $L(v)$ with the (already computed) walking path to t and add it to $B(t)$ if needed. MCR is adjusted similarly, with bags instead of labels.

5.2.2.4 *Beyond Walking*

We now consider other unrestricted networks (besides walking). In particular, our experiments include a bicycle rental scheme, which can be seen as a hybrid network: It does not have a fixed schedule (and is thus unrestricted), but bicycles can only be picked up and dropped off at designated *cycling stations*. Picking a bike from its station counts as a trip. To handle cycling within MCR, we consider it during the first stage of each round (together with RAPTOR and before walking). Because bicycles have no schedule, we process them independently (from RAPTOR) by running MLC on the bicycle network. To do so, we initialize MLC with labels from round $i-1$ for all relevant bicycle stations and, during the algorithm, we update labels of (the current) round i.

We consider a taxi ride to be a trip as well, since we board a vehicle. Moreover, we also optimize a separate criterion reflecting the (monetary) *cost* of taxi rides. If taxis were not penalized in any way, an all-taxi journey would almost always dominate all other alternatives (even sensible ones), since it is fast and has no walking. Our round-based algorithms handle taxis as they do walking, except that in the taxi stage labels are read from round $i-1$ and written into round i. Note that we link the taxi network to public transit stops as well as bicycle stations and that—unlike with rental bicycles—we also allow taking

a taxi as the first and/or last leg of any location-to-location query. Dealing with personal cars or bicycles is simpler. Assuming that they are only available for the first or last legs of the journey, we must only consider them during initialization. Initialization can also handle other special cases, such as allowing rented bicycles to be ridden to the destination (to be returned later).

Note that contraction can be used for cycling and driving. For every unrestricted network (walking, cycling, driving), we keep the link vertices (stops and bicycle stations) in one common core and contract (up to) all other vertices. As before, queries start with upward searches in each relevant unrestricted network.

5.2.3 *Heuristics*

Even with all accelerations, the exact algorithms proposed in Section 5.2.2 are not fast enough for interactive applications. This section proposes quick heuristics aimed at finding a set of journeys that is similar to the exact solution, which we take as ground truth. We consider three approaches: weakening the dominance rules, restricting the amount of walking, and reducing the number of criteria. We also discuss how to measure the quality of the heuristic solutions we find.

5.2.3.1 *Weak Dominance.*

The first strategy we consider is to weaken the domination rules during the algorithm, reducing the number of labels pushed through the network. We test four implementations of this strategy. The first, MCR-hf, uses fuzzy dominance (instead of strict dominance) when comparing labels during the algorithm: For labels L_1 and L_2, we compute the fuzzy dominance value $d(L_1, L_2)$ (cf. Section 5.2.1) and dominate L_2 if d exceeds a given threshold (we use 0.9). The second, MCR-hb(κ), uses strict dominance, but discretizes criterion κ: before comparing labels L_1 and L_2, we first round $\kappa(L_1)$ and $\kappa(L_2)$ to predefined discrete values (*buckets*); this can be extended to use buckets for several criteria. The third heuristic, MCR-hs(κ), uses strict dominance but adds a slack of x units to κ. More precisely, L_1 already dominates L_2 if $\kappa(L_1) \leqslant \kappa(L_2) + x$ and L_1 is at least as good L_2 in all other criteria. The last heuristic, MCR-ht, weakens the domination rule by trading off two or more criteria. More concretely, consider the case in which walking (walk) and arrival time (arr) are criteria. Then, L_1 already dominates L_2 if $\text{arr}(L_1) \leqslant \text{arr}(L_2) + \alpha \cdot (\text{walk}(L_1) - \text{walk}(L_2))$, $\text{walk}(L_1) \leqslant \text{walk}(L_2) + \alpha \cdot (\text{arr}(L_1) - \text{arr}(L_2))$, and L_1 is at least as good as L_2 in all other criteria, for a tradeoff parameter α (we use $\alpha = 0.3$).

5.2.3.2 *Restricting Walking.*

Consider our simple scenario of walking and public transit. Intuitively, most journeys start with a walk to a nearby stop, followed by one or more trips (with short transfers) within the public transit system, and finally a short walk from the final stop to the actual destination. This motivates a second class of heuristics, MCR-tx. It still runs three-criterion search (walking, arrival, and trips), but limits walking transfers between stops to x minutes; in this case we precompute these transfers. MCR-tx-ry also limits walking in the beginning and end to y minutes. Note that existing solutions often use such restrictions [BCE+10].

5.2.3.3 *Fewer Criteria.*

The last strategy we study is reducing the number of criteria considered during the algorithm. As already mentioned, this is a common approach in practice. We propose MR-x, which still works in rounds, but optimizes only the number of trips and arrival times explicitly (as criteria). To account for walking duration, we count every x minutes of a walking segment (transfer) as a trip; the first x minutes are free. With this approach, we can run plain Dijkstra to compute transfers, since link vertices no longer need to keep bags. The round index to which labels are written then depends on the walking duration (of the current segment) of the considered label. A special case is $x = \infty$, where a transfer is never a trip. Another variant is to always count a transfer as a single trip, regardless of duration; we abuse notation and call this variant MR-0. We also consider MR-∞-tx: Walking duration is not an explicit criterion and transfers do not count as trips, but are limited to x minutes.

For scenarios that include cost as a criterion (for taxis), we consider variants of the MCR-hb and MCR-hf heuristics. In both cases, we drop walking as an independent criterion, leaving only arrival time, number of trips, and costs to optimize. We account for walking by making it a (cheap) component of the costs.

5.2.3.4 *Quality Evaluation*

To measure the quality of a heuristic, we compare the set of journeys it produces to the *ground truth*, which we define as the solution found by MCR. To do so, we first compute the score of each journey with respect to the Pareto set that contains it (cf. Section 5.2.1). Then, for a given parameter k, we measure the similarity between the top k scored journeys returned by the heuristics and the top k scored journeys in the ground truth. Note that the score depends only on the algorithm itself and does not assume knowledge of the ground truth, which is consistent with a real-world deployment.

To compare two sets of k journeys, we run a greedy maximum matching algorithm. First, we compute a $k \times k$ matrix where entry (i, j) represents the similarity between the i-th journey in the first set and the j-th in the second. To measure the similarity, we make use of the same fuzzy relational operators we use for scoring. More precisely, given two journeys J_1 and J_2, the similarity with respect to the i-th criterion is given by $c^i := \mu^i_=(\kappa^i(J_1) - \kappa^i(J_2))$, where κ^i is the value of this criterion and $\mu^i_=$ is the corresponding fuzzy equality relation. Then, we define the similarity between J_1 and J_2 as $T(c^1, c^2, \ldots, c^M)$, where T is an arbitrary t-norm. We always select T to be consistent with the s-norm that we use to compute the score values.

After computing the pairwise similarities, we greedily select the unmatched pairs with highest similarity (by picking the highest entry in the matrix that does not share a row or column with a previously picked entry). The similarity of the whole matching is the average similarity of its pairs, weighted by the fuzzy score of the reference journey. This means that matching the highest-scored reference journey is more important than matching the k-th one.

5.2.4 *Experiments*

This section presents an extensive evaluation of the methods introduced before. All algorithms from Sections 5.2.2 and 5.2.3 were implemented in C++ and compiled with g++ 4.6.2 (64 bits, flag `-O3`). We ran our experiments on one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM.

5.2.4.1 *Input and Methodology.*

We focus on the transportation network of London (England); results for other instances are similar, as Section 5.2.4.7 will show. We use the timetable information made available by Transport for London (TfL) [Lds; TfL], from which we extracted a Tuesday in the periodic summer schedule of 2011. The data includes subway (tube), buses, tram, ferries, and light rail (DLR), as well as bicycle station locations. To model the underlying road network, we use data provided by PTV AG [PTV79] from 2006, which explicitly indicates whether each road segment is open for driving, cycling and/or walking. We set the walking speed to 5 km/h and the cycling speed to 12 km/h, and we assume driving at free-flow speeds. We do not consider turn costs, which are not defined in the data. The resulting combined network has 564 cycle stations and about 20 k stops, 5 M departure events, and 259 k vertices in the walking network. Exact numbers are given in Table 5.8.

Recall that we specify the fuzziness of each criterion by a pair (χ, ε), roughly meaning that the corresponding Gaussian (centered at $x = 0$) has value χ for $x = \varepsilon$. We set these pairs to $(0.8, 5)$ for walking,

Table 5.8: Size figures for our input instances. We link every stop and cycle station with the walking/taxi network.

Figure	London	New York	Los Angeles	Chicago
Public Transit				
Stops	20 843	17 894	15 003	12 137
Routes	2 184	1 393	1 099	710
Trips	133 011	45 299	16 376	20 303
Daily Departure Events	4 991 125	1 825 129	931 846	1 194 571
Vertices (Route Model)	99 230	66 124	81 657	47 561
Edges (Route Model)	260 583	193 159	214 369	118 452
Walking				
Vertices	258 840	255 808	224 053	70 440
Vertices in Core	27 840	25 808	21 053	16 440
Edges	1 433 814	1 586 782	1 395 185	586 979
Footpaths $\leqslant$ 5 min	150 948	219 040	83 844	122 450
Footpaths $\leqslant$ 10 min	518 174	670 702	271 444	426 818
Cycling				
Cycle Stations	564	—	—	—
Vertices	23 311	—	—	—
Vertices in Core	1 311	—	—	—
Edges	130 971	—	—	—
Taxi				
Vertices	259 122	—	—	—
Vertices in Core	27 122	—	—	—
Edges	1 339 487	—	—	—

to $(0.8, 1)$ for arrival time, $(0.1, 1)$ for trips, and $(0.8, 5)$ for costs (given in pounds; times are in minutes). Note that the number of trips is sharper than the other criteria. Later in this section we show that our approach is robust to small variations in these parameters, but they can be tuned to account for user-dependent preferences. If not indicated otherwise, our experiments consider the minimum/maximum norms by default. We run *location-to-location* queries, with sources, targets, and departure times picked uniformly at random (from the walking network and during the day, respectively).

5.2.4.2 *Algorithms Evaluation.*

For our first experiment, we use walking, cycling, and the public transportation network and consider three criteria: arrival time, number of trips, and walking duration. We ran 1 000 queries for each algorithm. Table 5.9 summarizes the results. For each algorithm, the table first

Table 5.9: Performance and solution quality on journeys considering walking, cycling, and public transit. Bullets (•) indicate the criteria taken into account by the algorithm.

Algorithm	Arr.	Trp.	Wlk.	Rnd.	Scans / Ent.	Comp. / Ent.	Jn.	Time [ms]	Quality-3 Avg.	Quality-3 Sd.	Quality-6 Avg.	Quality-6 Sd.
MCR-full	•	•	•	13.8	13.8	168.2	29.1	4 634.0	100 %	0 %	100 %	0 %
MCR	•	•	•	13.8	3.4	158.7	29.1	1 438.7	100 %	0 %	100 %	0 %
MLC	•	•	•	—	10.6	1 246.7	29.1	4 543.0	100 %	0 %	100 %	0 %
MCR-hf	•	•	•	15.6	2.9	14.3	10.9	699.4	89 %	15 %	89 %	11 %
MCR-hb	•	•	•	10.2	2.1	12.7	9.0	456.7	91 %	12 %	91 %	10 %
MCR-hs	•	•	•	14.7	2.6	11.1	8.6	466.1	67 %	28 %	69 %	23 %
MCR-ht	•	•	•	10.5	2.0	6.4	8.6	373.6	84 %	22 %	82 %	20 %
MCR-t10	•	•	•	13.8	2.7	132.7	29.0	1 467.6	97 %	10 %	95 %	10 %
MCR-t10-r15	•	•	•	10.7	1.7	73.3	13.2	885.0	38 %	40 %	30 %	31 %
MCR-t5	•	•	•	13.8	2.7	126.6	28.9	891.9	93 %	16 %	92 %	15 %
MR-∞	•	•	○	7.6	1.4	4.8	4.5	44.4	63 %	28 %	63 %	24 %
MR-0	•	•	○	13.7	2.1	6.9	5.4	61.5	63 %	28 %	63 %	24 %
MR-10	•	•	○	20.0	1.1	4.8	4.3	39.4	51 %	33 %	45 %	29 %
MR-∞-t10	•	•	○	7.6	1.1	4.8	4.5	22.2	63 %	28 %	62 %	24 %

shows which criteria are explicitly taken into account. The next five columns show the average values observed for the number of rounds, scans per entity (stop/vertex), label comparisons per entity, journeys found, and running time (in milliseconds). The last four columns evaluate the quality of the top 3 and 6 journeys found by our heuristics, as explained in Section 5.2.3. Note that we show both averages and standard deviations.

The methods in Table 5.9 are grouped in blocks. Those in the first block compute the full Pareto set considering all three criteria (arrival time, number of trips, and walking). MCR, our reference algorithm, is round-based and uses contraction in the unrestricted networks. As anticipated, it is faster (by a factor of about three) than MCR-full (which does not use the core) and MLC (which uses the core but is not round-based). Accordingly, all heuristics we test are round-based and use the core.

The second block contains heuristics that accelerate MCR by weakening the domination rules, causing more labels to be pruned (and losing optimality guarantees). As explained in Section 5.2.3, MCR-hf uses fuzzy dominance during the algorithm, MCR-hb uses walking *buckets* (discretizing walking by steps of 5 minutes for domination), MCR-hs uses a slack of 5 minutes on the walking criterion when evaluating domination, and MCR-ht considers a tradeoff parameter

Table 5.10: Detailed performance analysis of our algorithms. The total running time includes additional overhead, such as for initialization.

				Public Transit		Walking		Cycling		Total	
Algorithm	Arr.	Trp.	Wlk.	Scans / Stop	Time [ms]	Scans / Vert.	Time [ms]	Scans / Vert.	Time [ms]	Scans / Ent.	Time [ms]
MCR-full	•	•	•	32.1	350.6	9.6	3 030.9	43.6	1 203.1	13.8	4 634.0
MCR	•	•	•	32.1	341.4	1.2	889.3	1.7	159.2	3.4	1 438.7
MLC	•	•	•	119.3	—	2.6	—	2.1	—	10.6	4 543.0
MCR-hf	•	•	•	28.1	157.7	1.0	483.9	0.7	25.6	2.9	699.4
MCR-hb	•	•	•	21.1	115.2	0.7	297.4	0.5	19.7	2.1	456.7
MCR-hs	•	•	•	25.1	97.3	0.9	322.2	0.6	16.8	2.6	466.1
MCR-ht	•	•	•	20.2	86.8	0.7	246.4	0.5	17.4	2.0	373.6
MCR-t5	•	•	•	31.5	318.4	0.5	348.6	1.7	157.2	2.7	891.9
MCR-t10	•	•	•	31.6	326.2	0.5	913.7	1.7	158.5	2.7	1 467.6
MCR-t10-r15	•	•	•	20.0	207.5	0.3	554.0	1.2	103.6	1.7	885.0
MR-∞	•	•	○	14.2	10.0	0.5	31.0	0.3	1.8	1.4	44.4
MR-0	•	•	○	21.4	13.9	0.7	42.5	0.4	2.4	2.1	61.5
MR-10	•	•	○	9.7	6.3	0.5	30.5	0.2	1.3	1.1	39.4
MR-∞-t10	•	•	○	14.4	9.4	0.2	9.5	0.3	1.6	1.2	22.2

of $\alpha = 0.3$ between walking and arrival time. All heuristics are faster than pure MCR, and MCR-hb gives the best quality at a reasonable running time.

The third block has algorithms with restrictions on walking duration. Limiting transfers to 10 minutes (as MCR-t10 does) has almost no effect on solution quality (which is expected in a well-designed public transportation network). Moreover, adding precomputed footpaths of 10 minutes is not faster than using the core for unlimited walking (as MCR does). Additionally limiting the walking range from s or t (MCR-t10-r15) improves speed, but the quality becomes unacceptably low: The algorithm misses good journeys (including all-walk) quite often. If instead we allow even more restricted transfers (with MCR-t5), we get a similar speedup with much better quality (comparable to MCR-hb).

The MR-x algorithms (fourth block) reduce the number of criteria considered by combining trips and walking. The fastest variant is MR-∞-t10, which drops walking duration as a criterion but limits the amount of walking at transfers to 10 minutes, making it essentially the same as RAPTOR, with a different initialization. As expected, however, quality is much lower than for MCR-tx, confirming that considering the walking duration explicitly during the algorithm is important to obtain a full range of solutions. MR-10 attempts to improve quality

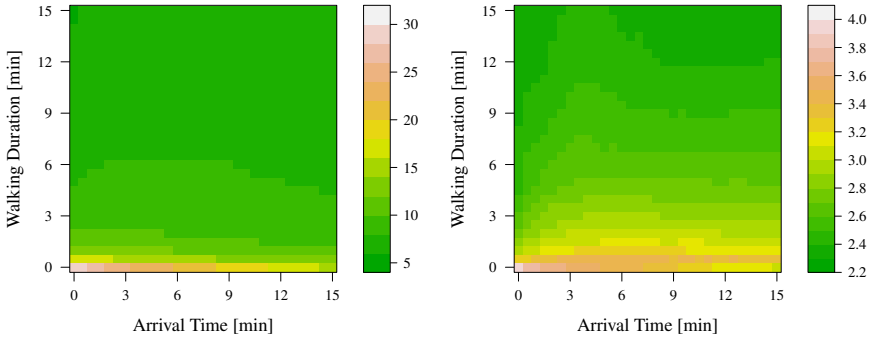


Figure 5.6: Number of Pareto optimal journeys with score higher than 0.1 for varying fuzziness. We consider both the maximum norm (left) and probabilistic sum (right). The x axis varies the fuzziness in the arrival time, while the y axis considers the walking duration. The intensity (color) of the corresponding entry indicates the average number of journeys in the filtered output.

by transforming long walks into extra trips, but is not particularly successful as the obtained results are actually worse.1

Summing up, MCR-hb should be the preferred choice for high-quality solutions, while MR-∞-t10 can support interactive queries with reasonable quality.

5.2.4.3 *Detailed Performance*

Table 5.10 presents a more detailed analysis. For each algorithm, it shows the effort (number of scans per vertex and/or stop, as well as running times in milliseconds) spent in each of the networks (public transit, walking, and cycling) and in total. The table shows that all round-based algorithms except MR-∞-t10 spend significantly more time processing the unrestricted networks (walking and cycling) than dealing with public transportation. This was to be expected: not only are the unrestricted networks bigger (they have more vertices), but also they must be processed with a (slower) Dijkstra-based algorithm (as in MLC, rather than RAPTOR). This is the reason for the good performance of the MR-∞-t10 heuristic.

5.2.4.4 *Fuzzy Parameters Evaluation.*

We also evaluated the impact of the fuzzy parameters on the number of journeys we obtain. We again use London with walking, public transit, and cycling as input. Figure 5.6 shows the number of journeys given a score higher than 0.1 (by the fuzzy ranking routine) when we vary ε (the level of fuzziness) for two criteria, walking and arrival time. We set $\chi = 0.8$, as in our main experiments. To not overload the figure, we keep the fuzziness of the third criterion (number of trips) constant.

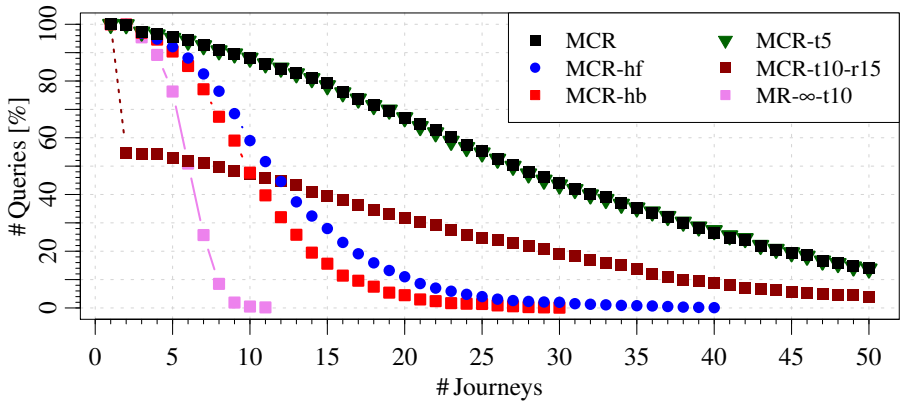


Figure 5.7: Evaluating the number of journeys returned by some of our algorithms: For a given n (on the abscissa), we report the percentage of 1 000 random queries that compute n or more journeys.

A comparison between the plots shows that, for the same set of parameters, probabilistic sum is significantly stricter than the maximum norm, and reduces the number of journeys much more drastically (for a fixed threshold). Qualitatively, however, they behave similarly. Under both norms, making the walking criterion fuzzier is more effective at identifying unwanted journeys. A couple of minutes of fuzziness in the walking criterion is enough to significantly reduce the number of journeys above the threshold. Adding fuzziness only to the arrival time has much more limited effect on the results.

5.2.4.5 *Quality of the Heuristics.*

We here further investigate the quality of our heuristics. We use London with walking, public transit, and cycling as input. Figure 5.7 reports the size of the Pareto set (the input to scoring) for various algorithms, while Figure 5.8 shows how well the the top k heuristic journeys match the ground truth, for varying k. We observe that exact MCR (even if restricted to 5-minute transfers) does indeed produce many journeys, supporting the notion of ranking them afterwards (by score). A good heuristic, such as MCR-hb, computes much fewer journeys, but they match the top MCR journeys quite well. An interesting observation is that the quality of the heuristic hardly depends on the number of journeys we try to match.

5.2.4.6 *Multimodal Problem with Taxis.*

Our final experiment considers a multimodal problem, including taxis. We add *cost* as fourth criterion (at 2.40 pounds per taxi trip plus 60 pence per minute). We do not consider the cost of public transit, since it is significantly cheaper. Table 5.11 presents the average performance of some of our algorithms over 1 000 random queries in London. The first block includes algorithms that optimize all four criteria (arrival time,

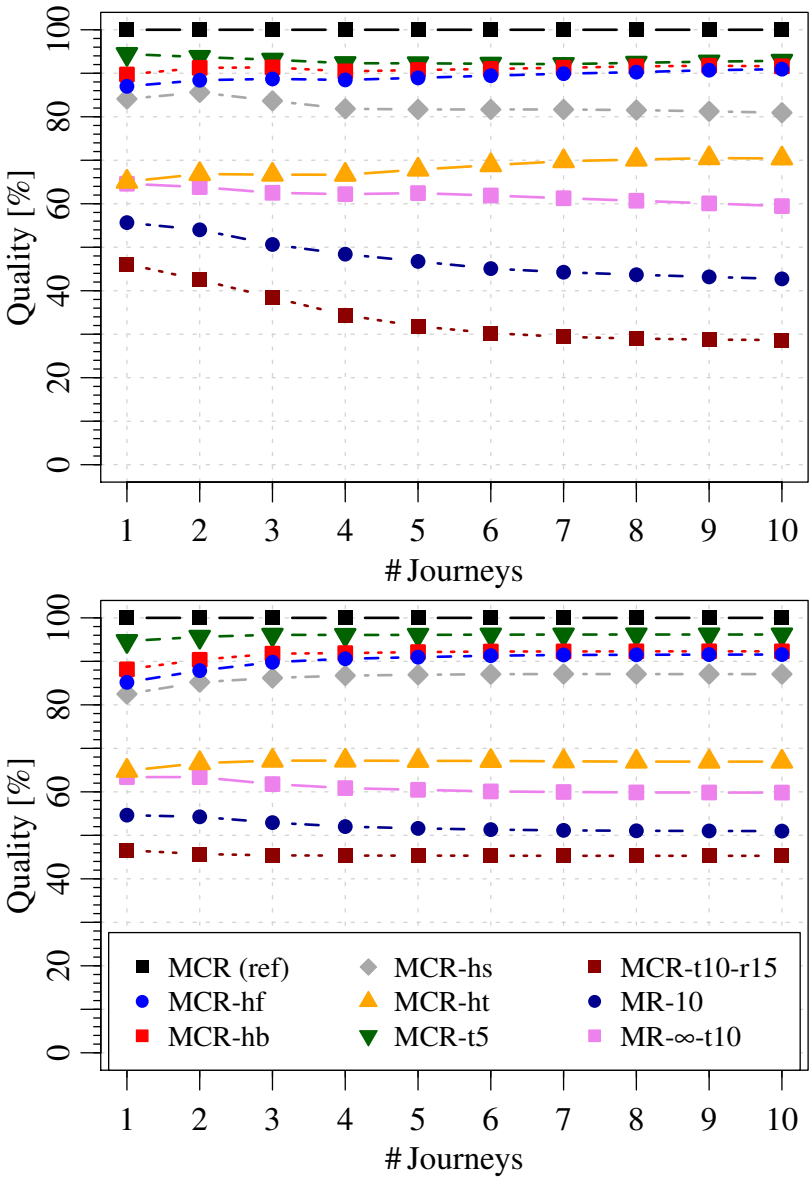


Figure 5.8: Evaluating the solution quality by matching the top k journeys in the solution with the top k of the reference algorithm (MCR). The scores and similarity values are obtained by using the minimum/maximum norms (left) and the product norm/probabilistic sum (right). The legend of the right plot also applies to the left.

Table 5.11: Performance on our London instance when taking taxi into account.

Algorithm	Arr.	Trp.	Wlk.	Cost	Rnd.	Scans / Ent.	Comp. / Ent.	Jn.	Time [ms]	Quality-3 Avg.	Quality-3 Sd.	Quality-6 Avg.	Quality-6 Sd.
MCR	•	•	•	•	16.3	3.1	369 606.0	1 666.0	1 960 234.0	100 %	0 %	100 %	0 %
MCR-hf	•	•	•	•	17.1	2.1	137.1	35.2	6 451.6	92 %	12 %	92 %	6 %
MCR-hb	•	•	•	•	9.9	1.3	86.8	27.6	2 807.7	96 %	8 %	92 %	6 %
MCR	•	•	○	•	14.6	2.4	7 901.4	250.9	25 945.8	98 %	6 %	97 %	5 %
MCR-hf	•	•	○	•	12.0	1.4	33.6	17.6	2 246.3	87 %	12 %	74 %	12 %
MCR-hb	•	•	○	•	9.0	1.0	20.0	11.6	996.4	86 %	12 %	74 %	12 %

walking duration, number of trips, and costs). While exact MCR is impractical, fuzzy domination (MCR-hf) makes the problem tractable with little loss in quality. Using 5-minute buckets for walking and 5-pound buckets for costs (MCR-hb) is even faster, though queries still take more than two seconds. The second block shows that we can reduce running times by dropping walking duration as a criterion (we incorporate it into the cost function at 3 pence per minute, instead), with almost no loss in solution quality. This is still not fast enough, though. Using 5-pound buckets (MCR-hb) reduces the average query time to about 1 second, with reasonable quality.

5.2.4.7 *Additional Inputs*

In addition to London, we tested inputs representing other large metropolitan areas (New York, Los Angeles, and Chicago). We built the public transit network from publicly available General Transit Feeds (GTFS) [Gtfs], restricting ourselves to the timetable for August 10, 2011 (a Wednesday). The walking network data is still given by PTV [PTV79], and these instances do not include bicycles. Detailed statistics for all instances were presented in Table 5.8.

Table 5.12 compares the performance of our algorithms on these inputs. For reference, we also consider a simplified version of the London network, without bicycles. For each input, we show the average values (over 1 000 queries) for number of journeys found, running time, and quality (considering the top 6 journeys). The results are consistent with those obtained for the full London network, showing that our preferred choice of heuristics also holds here. MCR-hb is always the best choice in terms of solution quality (among methods with reasonable speedups), while MR-∞-t10 is preferred if query times should be as low as possible.

Table 5.12: Evaluating the performance of MCR and MR with different heuristics on other instances. The quality is determined identically to Table 5.9 (cf. Section 5.2.4).

				New York			Los Angeles			Chicago		
Algorithm	Arr.	Trp.	Wlk.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.
MCR	•	•	•	25.5	1 703.0	100 %	16.7	644.6	100 %	22.1	532.8	100 %
MCR-hf	•	•	•	8.6	611.0	91 %	8.9	445.0	88 %	8.3	241.3	72 %
MCR-hb	•	•	•	7.2	413.8	94 %	7.6	295.8	93 %	7.1	160.8	92 %
MCR-hs	•	•	•	6.7	414.0	84 %	7.4	310.7	62 %	6.6	158.8	58 %
MCR-ht	•	•	•	6.6	300.9	80 %	6.7	228.4	69 %	6.2	113.9	79 %
MCR-t5	•	•	•	25.6	695.5	69 %	16.6	262.7	93 %	21.9	277.7	95 %
MCR-t10	•	•	•	25.3	1 401.4	85 %	16.8	424.5	96 %	22.0	578.8	98 %
MCR-t10-r15	•	•	•	5.4	677.9	10 %	3.9	202.0	13 %	9.6	372.7	28 %
MR-∞	•	•	○	3.4	26.3	65 %	3.6	21.5	51 %	3.3	12.3	63 %
MR-0	•	•	○	3.8	37.6	65 %	4.3	28.5	52 %	3.7	15.6	63 %
MR-10	•	•	○	6.0	26.1	41 %	6.1	26.6	42 %	5.1	13.9	50 %
MR-∞-t10	•	•	○	3.6	10.6	60 %	3.6	11.0	51 %	3.3	7.1	63 %

Conclusions

We have studied multicriteria journey planning in multimodal networks. We argued that users optimize three criteria: arrival time, costs, and convenience. Although the corresponding full Pareto set is large and has many unnatural journeys, fuzzy set theory can extract the relevant results and rank them. Since exact algorithms are too slow, we have introduced several heuristics that closely match the best journeys in the Pareto set. Our experiments show that our approach enables efficient realistic multimodal journey planning in large metropolitan areas. A natural avenue for future research is accelerating our approach further to enable interactive queries with an even richer set of criteria in dynamic scenarios, handling delay and traffic information. The ultimate goal is to compute multicriteria multimodal journeys on a global scale in real time.

6 FINAL REMARKS

In this thesis, we examined several new approaches towards more realistic route planning in road networks, public transit and multimodal journey planning. A central theme was enabling user preferences, which we implemented in different ways. In Sections 3.1 and 3.2, it was achieved through fast metric-dependent preprocessing that can be adapted to a new global optimization objective, quickly. To this end, we introduced Customizable Contraction Hierarchies (CCH) and extended Customizable Route Planning (CRP) [DGPW15] to time-dependent, functional metrics. In Section 5.1 we achieved user choice by introducing the first label-constrained shortest path technique that takes multimodal transfer sequence constraints as query input, enabling the user to specify (and change) modal preferences even after preprocessing. In Sections 4.1 and 4.2, and especially in Section 5.2, we enabled user choice through multicriteria optimization, providing solution sets to choose from. In addition, the approach for multicriteria multimodal journey planning proposed in Section 5.2 allows parametrization of the fuzzy logic operators (for identifying significant journeys) to a user's situational preferences and trade-offs: For example, even an enthusiastic pedestrian might start to notice the difference in five more minutes of walking, if it is raining heavily.

For public transit journey planning, we introduced two new approaches: A baseline algorithm, Connection Scan Algorithm (CSA), that works directly on the timetable and requires no priority queue, as well as a preprocessing technique, Public Transit Labeling (PTL), adapted from road networks. Both approaches greatly improve performance over the state-of-the-art.

6.1 FUTURE WORK

We conclude by discussing interesting directions for future work.

6.1.1 *User Preferences*

User preferences can be challenging in terms of user interface design. Especially on mobile devices, there is hardly room for parameter input. But more importantly, they require the user to set parameters before understanding their effect in terms of route options. One way around this issue, of course, is to provide multiple solutions from the parameter space at once (such as we have done in Section 5.2). Another approach would learn optimization preferences by observation: In-

stead of *customizable* route planning it would provide *customized* route planning. For a journey planner with integrated fare management, this could be achieved by monitoring the journeys that are finally selected for booking. For car navigation, it has been proposed to analyze GPS traces [DGG+15]. It would be interesting to extend this approach to multimodal journey planning (where GPS tracking itself can already be a challenge).

6.1.2 *Customizable Contraction Hierarchies*

While we considered point-to-point queries, other query types are also of practical importance, for example: one-to-all, many-to-many, and k-nearest-neighbor queries, which have been evaluated for traditional (greedy) Contraction Hierarchies (CH) and related hierarchical techniques [ADF+12; DGNW13; DGW11a; GLS+10; GSSV12; KSS+07]. Since Customizable CH (CCH), see Section 3.1, still yields a CH search space, algorithmic results directly carry over. Similarly, CCH could be applied to the computation of alternative routes [ADGW13; KRS13].

Preliminary experiments by us for customizable PHAST (applying nested dissection orders to the approach presented in [DGNW13]) show that performance on travel time metric degrades only by 40 % over results in [DGNW13], although we observe about a factor of 2 increase in the number of CCH arcs (compared to traditional CH). This is due to the fact that graph separation yields improved cache locality. Moreover, it should enable better parallelization of PHAST without requiring synchronization between levels (as opposed to [DGNW13]).

Likewise, customizable hub labeling (HL) (applying nested dissection orders to the approach presented in, e. g., [ADGW11; DGPW14]) enables straightforward parallelization of pruned labeling [AIY13]. It also achieves much earlier pruning, as the graph decomposes naturally. However, we have experimentally observed an increase in labeling size by an order of magnitude. Given the already high overhead of normal HL (tens of gigabytes), this seems impractical. Partial labeling approaches, however, with limited local search, might be practical.

6.1.3 *Traffic Patterns*

In Section 3.2, we essentially provided a 4-phase speedup technique: Phase 1 computes nested multi-level separators and the metric-independent overlay. Phase 2 computes historic time-dependent overlay functions.[1] Then, Phase 3 customizes overlay arcs with live traffic and user preferences. Finally, Phase 4 provides queries. A detailed experimental analysis of such a setup would be very interesting (we examined Phase 2 and 3 as one).

1 These are relatively static, typically updated every few months, according to personal communication with a large provider of navigation services.)

Furthermore, our approach is still not fully customizable: it requires arc cost functions that map time to time. This is good enough to model avoidance of highways or driving slower than the speed limit, but it cannot handle combined linear optimization of (time-dependent) travel time and, e. g., toll costs. For that, one should investigate the application of generalized time-dependent objective functions as proposed in [BS12].

Also, time-dependent shortcuts have proven problematic in Section 3.2, due to functional complexity growth on long paths. Revisiting a hierarchical preprocessing technique that does not use shortcuts, such as Reach [Gut04; MCB14], could be interesting.

6.1.4 *Public Transit*

For CSA (cf. Section 4.1), we observed fast query times, which since have further been accelerated [SW14]. For plain CSA, this is largely because of good cache locality. The approach is, however, quite verbose in its representation of the timetable: Connections are not grouped by routes or trips as in other approaches [DKP12; DPW14; PSWZ04; PSWZ08; Wit15]. It would be interesting to investigate whether frequency compression (see, e. g., [BS14; DPW14]) could be applied to CSA without too much of a performance loss.

For PTL (cf. Section 4.2), we saw excellent performance for earliest arrival and profile search problems, even on the largest networks available to us. Multicriteria labeling (minimizing travel time and number of transfers), however, suffers from a very steep increase in preprocessing time and space overhead (requiring 26 GiB on metropolitan London). Future research could revisit 3-hop labeling [YAIY13], covering shortest paths with trips [Wit15] instead of connections. Since the number of trips taken corresponds directly to the number of transfers, this could greatly improve multicriteria labeling.

Furthermore, currently for PTL, dynamic scenarios (e. g., temporary station closures and train delays or cancellations) are difficult, as they change not only the metric but also the topology of the underlying graph. Here, PTL could benefit from recent results in [CDD+14; MMPZ13] that examine fast graph topology updates.

Finally, as evidenced by comparison with the state-of-the-art in Section 4.8, there is a great need for a set of common, shared, and open benchmarks for public transit journey planning research.

6.1.5 *Multimodal*

In Chapter 5, we discussed two approaches to multimodal journey planning: label constraints on modal transfers (to suppress results suggesting, e. g., to take a private car between train rides) and multicriteria optimization. While we originally discussed the first approach in

the context of user *preferences*, examination of the second approach suggests that future work should treat modal constraints only as physical *restrictions*. This simplifies label constraints from (somewhat arbitrary) automata towards just tracking the availability of modes for subpaths: If, for example, the user left her bike at the first metro stop, the current path cannot be extended by biking. Algorithms for multicriteria set dominance, as used for fare zone optimization [DPW14], then already solve the problem of physical restrictions.

Another natural avenue for future research is accelerating the multicriteria approach further to enable interactive queries. So far, in Section 5.2, we did not employ preprocessing other than for the road network. Future work should investigate preprocessing of multimodal multicriteria solutions on an overlay. For that, our current approach of identifying significant solutions (by means of fuzzy logic) is problematic: Candidates qualify as accepted solutions based on relative comparisons with other candidates. In that regard, the approach described in [BBS13] seems promising: It accepts a solution only based on its own properties, rejecting solutions that, e. g., use only little taxi at the end of a long walking journey. It would, however, also reject the exemplary taxi ride between train stations in Paris, which some traveler might prefer. Further research on this matter is necessary.

For multimodal preprocessing, labeling is surprisingly straightforward from a conceptional point of view (extending our results on PTL): If one can formalize the set of multimodal shortest paths to be covered, a greedy approach [DGPW14] could always pick the hub that covers the most multimodal paths. Of course, the challenge lies in designing efficient algorithms for preprocessing this in practice, without exhaustive enumeration of all wanted paths.

BIBLIOGRAPHY

[ADF+11] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. "VC-Dimension and Shortest Path Algorithms." In: *Proceedings of the 38th International Colloquium on Automata, Languages, and Programming (ICALP'11)*. Vol. 6755. Lecture Notes in Computer Science. Springer, 2011, pp. 690–699 (cit. on p. 10).

[ADF+12] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. "HLDB: Location-Based Services in Databases." In: *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*. ACM Press, 2012, pp. 339–348 (cit. on pp. 9, 122, 152, 174).

[ADF+13] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. "Highway dimension and provably efficient shortest path algorithms." In: (2013) (cit. on pp. 2, 10).

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks." In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, pp. 230–241 (cit. on pp. 2, 9, 18, 174).

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Hierarchical Hub Labelings for Shortest Paths." In: *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Vol. 7501. Lecture Notes in Computer Science. Springer, 2012, pp. 24–35 (cit. on pp. 6, 9, 15, 17, 18, 40, 52, 114, 115, 116, 118).

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Alternative Routes in Road Networks." In: *ACM Journal of Experimental Algorithmics* 18.1 (2013), pp. 1–17 (cit. on p. 174).

[ADN+15] Simeon Danailov Andreev, Julian Dibbelt, Martin Nöllenburg, Thomas Pajor, and Dorothea Wagner. "Towards Realistic Pedestrian Route Planning." In: *Proceedings of the 15th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'15)*.

Vol. 48. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 09/2015, pp. 1–15 (cit. on p. 7).

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. "Highway Dimension, Shortest Paths, and Provably Efficient Algorithms." In: *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*. SIAM, 2010, pp. 782–793 (cit. on pp. 2, 10, 135).

[AIY13] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. "Fast exact shortest-path distance queries on large networks by pruned landmark labeling." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM Press, 2013, pp. 349–360 (cit. on pp. 9, 18, 115, 116, 174).

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. "Transit Node Routing Reconsidered." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 55–66 (cit. on pp. 9, 17).

[And12] Simeon Danailov Andreev. "Realistic Pedestrian Routing." Bachelor Thesis. Karlsruhe Institute of Technology, 11/2012 (cit. on p. 94).

[AW88] H. Alt and E. Welzl. "Visibility Graphs and Obstacle-avoiding Shortest Paths." English. In: *Zeitschrift für Operations Research* 32.3-4 (1988), pp. 145–164 (cit. on pp. 14, 92).

[Bas09] Hannah Bast. "Car or Public Transport – Two Worlds." In: *Efficient Algorithms*. Vol. 5760. Lecture Notes in Computer Science. Springer, 2009, pp. 355–367 (cit. on p. 14).

[BBH+09] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. "Engineering Label-Constrained Shortest-Path Algorithms." In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. DIMACS Book. American Mathematical Society, 2009, pp. 309–319 (cit. on pp. 15, 153).

[BBHK06] Raymond C. Browning, Emily A. Baker, Jessica A. Herron, and Rodger Kram. "Effects of Obesity and Sex on the Energetic Cost and Preferred Speed of Walking." In: *Journal of Applied Physiology* 100.2 (2006), pp. 390–398 (cit. on p. 96).

[BBM06] Maurizio Bielli, Azedine Boulmakoul, and Hicham Mouncif. "Object modeling and path computation for multimodal travel systems." In: *European Journal of Operational Research* 175.3 (2006), pp. 1705–1730 (cit. on pp. 15, 153).

[BBRW13] Reinhard Bauer, Moritz Baum, Ignaz Rutter, and Dorothea Wagner. "On the Complexity of Partitioning Graphs for Arc-Flags." In: *Journal of Graph Algorithms and Applications* 17.3 (2013), pp. 265–299 (cit. on p. 10).

[BBS13] Hannah Bast, Mirko Brodesser, and Sabine Storandt. "Result Diversity for Multi-Modal Route Planning." In: *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. OpenAccess Series in Informatics (OASIcs). 2013, pp. 123–136 (cit. on p. 176).

[BCE+10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. "Fast Routing in Very Large Public Transportation Networks using Transfer Patterns." In: *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Vol. 6346. Lecture Notes in Computer Science. Springer, 2010, pp. 290–301 (cit. on pp. 6, 14, 126, 154, 162).

[BCK+10] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. "Preprocessing Speed-Up Techniques is Hard." In: *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*. Vol. 6078. Lecture Notes in Computer Science. Springer, 2010, pp. 359–370 (cit. on p. 10).

[BCKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. 3rd. Springer, 2008 (cit. on pp. 88, 90, 91, 92, 93).

[BCRW13] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. "Search-Space Size in Contraction Hierarchies." In: *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13)*. Vol. 7965. Lecture Notes in Computer Science. Springer, 2013, pp. 93–104 (cit. on pp. 4, 10, 18, 21, 22).

[BD09] Reinhard Bauer and Daniel Delling. "SHARC: Fast and Robust Unidirectional Routing." In: *ACM Journal of Experimental Algorithmics* 14.2.4 (08/2009), pp. 1–29 (cit. on p. 9).

[BDD+12] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner. "The Shortcut Problem – Complexity and Algorithms." In: *Journal of Graph Algorithms and Applications* 16.2 (2012), pp. 447–481 (cit. on p. 10).

[BDG+15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller–Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*. Tech. rep. abs/1504.05140. ArXiv e-prints, 2015 (cit. on pp. 1, 6, 9, 84, 125, 128).

[BDGM09] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. "Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected." In: *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. OpenAccess Series in Informatics (OASIcs). 2009 (cit. on pp. 14, 138, 158).

[BDPW13] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. "Energy-Optimal Routes for Electric Vehicles." In: *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2013, pp. 54–63 (cit. on pp. 4, 13, 70).

[BDPW15] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. *Dynamic Time-Dependent Route Planning in Road Networks with User Preferences*. Tech. rep. 1512.09132. ArXiv e-prints, 2015 (cit. on p. 7).

[BDS+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. "Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm." In: *ACM Journal of Experimental Algorithmics* 15.2.3 (01/2010). Special Section devoted to WEA'08., pp. 1–31 (cit. on pp. 9, 14, 135).

[BDW11] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. "Experimental Study on Speed-Up Techniques for Timetable Information Systems." In: *Networks* 57.1 (01/2011), pp. 38–52 (cit. on p. 14).

[Ben75] Jon Louis Bentley. "Multidimensional Binary Search Trees Used for Associative Searching." In: *Commun. ACM* 18.9 (09/1975), pp. 509–517 (cit. on p. 93).

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. "Fast Routing in Road Networks with Transit Nodes." In: *Science* 316.5824 (2007), p. 566 (cit. on pp. 9, 10).

[BGM10] Annabell Berger, Martin Grimmer, and Matthias Müller–Hannemann. "Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks." In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Vol. 6049. Lecture Notes in Computer Science. Springer, 05/2010, pp. 35–46 (cit. on p. 14).

[BGNS10] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. "Time-Dependent Contraction Hierarchies and Approximation." In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Vol. 6049. Lecture Notes in Computer Science. Springer, 05/2010, pp. 166–177 (cit. on p. 139).

[BGSV13] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. "Minimum Time-Dependent Travel Times with Contraction Hierarchies." In: *ACM Journal of Experimental Algorithmics* 18.1.4 (04/2013), pp. 1–43 (cit. on pp. 4, 12, 18, 70, 75, 81, 82, 83, 84, 85).

[BHH+14] Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. "Efficient Computation of Visibility Polygons." In: *CoRR* abs/1403.3905 (2014) (cit. on p. 93).

[BJM00] Chris Barrett, Riko Jacob, and Madhav V. Marathe. "Formal-Language-Constrained Path Problems." In: *SIAM Journal on Computing* 30.3 (2000), pp. 809–837 (cit. on pp. 3, 15, 134, 135).

[BK10] Hans L. Bodlaender and Arie M. C. A. Koster. "Treewidth computations I. Upper bounds." In: *Information and Computation* 208.3 (2010), pp. 259–275 (cit. on p. 46).

[BKMW10] Reinhard Bauer, Marcus Krug, Sascha Meinert, and Dorothea Wagner. "Synthetic Road Networks." In: *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM'10)*. Vol. 6124. Lecture Notes in Computer Science. Springer, 2010, pp. 46–57 (cit. on p. 10).

[Bod07] Hans L. Bodlaender. "Treewidth: Structure and Algorithms." In: *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity*. Vol. 4474. Lecture Notes in Computer Science. Springer, 2007, pp. 11–25 (cit. on p. 21).

[Bod93] Hans L. Bodlaender. "A Tourist Guide through Treewidth." In: *j-acta-cybernet* 11 (1993), pp. 1–21 (cit. on p. 21).

[BPS11] Miquel Ginard Ballester, Maurici Ruiz Pérez, and John Stuiver. "Automatic Pedestrian Network Generation." In: *Proceedings 14th AGILE International Conference on GIS*. 2011, pp. 1–13 (cit. on pp. 13, 102).

[BS12] Gernot Veit Batz and Peter Sanders. "Time-Dependent Route Planning with Generalized Objective Functions." In: *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Vol. 7501. Lecture Notes in Computer Science. Springer, 2012 (cit. on pp. 18, 175).

[BS14] Hannah Bast and Sabine Storandt. "Frequency-Based Search for Public Transit." In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 11/2014, pp. 13–22 (cit. on pp. 6, 14, 125, 126, 129, 175).

[CDD+14] Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. "Engineering Graph-Based Models for Dynamic Timetable Information Systems." In: *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*. Ed. by Stefan Funke and Matúš Mihalák. Vol. 42. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 09/2014, pp. 46–61 (cit. on pp. 14, 110, 111, 129, 175).

[Cgal15] The CGAL Project. *CGAL User and Reference Manual*. 4.6. CGAL Editorial Board, 2015 (cit. on pp. 88, 92, 94, 97).

[CH66] K. Cooke and E. Halsey. "The Shortest Route Through a Network with Time-Dependent Internodal Transit Times." In: *Journal of Mathematical Analysis and Applications* 14.3 (1966), pp. 493–498 (cit. on p. 11).

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. "Reachability and Distance Queries via 2-Hop Labels." In: *SIAM Journal on Computing* 32.5 (2003), pp. 1338–1355 (cit. on pp. 6, 9, 15, 18, 114, 115).

[CHWF13] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. "TF-Label: A Topological-folding Labeling Scheme for Reachability Querying in a Large Graph." In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM Press, 2013, pp. 193–204 (cit. on pp. 15, 114, 115, 127).

[CZ00] Soma Chaudhuri and Christos Zaroliagis. "Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms." In: *Algorithmica* (2000) (cit. on pp. 10, 18, 69).

[Dan62] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962 (cit. on p. 9).

[DBS10] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. "A case for time-dependent shortest path computation in spatial networks." In: *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'10)*. 2010, pp. 474–477 (cit. on p. 11).

[DDFV12] Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, and Camillo Vitale. "Fully Dynamic Maintenance of Arc-Flags in Road Networks." In: *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*. Vol. 7276. Lecture Notes in Computer Science. Springer, 2012, pp. 135–147 (cit. on p. 11).

[DDP+12] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. *Computing and Evaluating Multimodal Journeys*. Tech. rep. 2012-20. Faculty of Informatics, Karlsruhe Institute of Technology, 2012 (cit. on p. 8).

[DDP+13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. "Computing Multimodal Journeys in Practice." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 260–271 (cit. on p. 8).

[DDPW15] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. "Public Transit Labeling." In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Lecture Notes in Computer Science. Springer, 2015, pp. 273–285 (cit. on pp. 8, 114).

[Dea04] Brian C. Dean. "Algorithms for Minimum-Cost Paths in Time-Dependent Networks with Waiting Policies." In: *Networks* 44.1 (08/2004), pp. 41–46 (cit. on p. 71).

[Del09] Daniel Delling. "Engineering and Augmenting Route Planning Algorithms." PhD thesis. Universität Karlsruhe (TH), Fakultät für Informatik, 2009 (cit. on p. 144).

[Del11] Daniel Delling. "Time-Dependent SHARC-Routing." In: *Algorithmica* 60.1 (05/2011), pp. 60–94 (cit. on pp. 12, 14, 82, 83, 84, 85).

[DGG+15] Daniel Delling, Andrew V. Goldberg, Moises Goldszmidt, John Krumm, Kunal Talwar, and Renato F. Werneck. "Navigation Made Personal: Inferring Driving Preferences from GPS Traces." In: *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2015 (cit. on p. 174).

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, eds. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. DIMACS Book. American Mathematical Society, 2009 (cit. on pp. 41, 42, 67).

[DGNW11] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. "PHAST: Hardware-Accelerated Shortest Path Trees." In: *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011, pp. 921–931 (cit. on p. 140).

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. "PHAST: Hardware-accelerated shortest path trees." In: *Journal of Parallel and Distributed Computing* 73.7 (2013), pp. 940–952 (cit. on pp. 9, 18, 174).

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Customizable Route Planning." In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, pp. 376–387 (cit. on pp. 9, 11, 17, 60, 61).

[DGPW14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Robust Distance Queries on Massive Networks." In: *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Vol. 8737. Lecture Notes in Computer Science. Springer, 09/2014, pp. 321–333 (cit. on pp. 9, 18, 62, 116, 117, 118, 119, 126, 127, 128, 129, 174, 176).

[DGPW15] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Customizable Route Planning in Road Networks." In: *Transportation Science* (2015) (cit. on pp. 4, 9, 11, 17, 18, 53, 60, 61, 62, 63, 72, 73, 74, 76, 77, 81, 87, 94, 95, 98, 173).

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. "Graph Partitioning with Natural Cuts." In: *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011, pp. 1135–1146 (cit. on pp. 10, 17, 22, 73, 78, 98).

[DGRW12] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. "Exact Combinatorial Branch-and-Bound for Graph Bisection." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 30–44 (cit. on p. 22).

[DGSW14] Daniel Delling, Andrew V. Goldberg, Ruslan Savchenko, and Renato F. Werneck. "Hub Labels: Theory and Practice." In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 259–270 (cit. on p. 9).

[DGW11a] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Faster Batched Shortest Paths in Road Networks." In: *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*. Vol. 20. OpenAccess Series in Informatics (OASIcs). 2011, pp. 52–63 (cit. on pp. 18, 174).

[DGW11b] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Shortest Paths in Road Networks: From Practice to Theory and Back." In: *it—Information Technology* 53 (12/2011), pp. 294–301 (cit. on p. 2).

[DGWZ08] Daniel Delling, Kalliopi Giannakopoulou, Dorothea Wagner, and Christos Zaroliagis. *Contracting Timetable Information Networks*. Tech. rep. 144. Arrival Technical Report, 2008 (cit. on p. 14).

[DHM+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. "High-Performance Multi-Level Routing." In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. DIMACS Book. American Mathematical Society, 2009, pp. 73–92 (cit. on pp. 9, 17).

[Dij59] Edsger W. Dijkstra. "A Note on Two Problems in Connexion with Graphs." In: *Numerische Mathematik* 1 (1959), pp. 269–271 (cit. on pp. 1, 9, 11, 14, 88, 158).

[DKKT13] Themistoklis Diamantopoulos, Dionysios Kehagias, Felix König, and Dimitrios Tzovaras. "Investigating the Effect of Global Metrics in Travel Time Forecasting." In: *Proceedings of the 16th International IEEE Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2013, pp. 412–417 (cit. on p. 70).

[DKP12] Daniel Delling, Bastian Katz, and Thomas Pajor. "Parallel Computation of Best Connections in Public Transportation Networks." In: *ACM Journal of Experimental*

Algorithmics 17.4 (07/2012), pp. 4.1–4.26 (cit. on pp. 14, 103, 107, 108, 109, 110, 111, 122, 123, 133, 143, 154, 158, 175).

[DKW14] Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. "Customizing Driving Directions with GPUs." In: *Proceedings of the 20th International Conference on Parallel Processing (Euro-Par 2014)*. Vol. 8632. Lecture Notes in Computer Science. Springer, 2014, pp. 728–739 (cit. on p. 11).

[DMS08] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. "Multi-Criteria Shortest Paths in Time-Dependent Train Networks." In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Vol. 5038. Lecture Notes in Computer Science. Springer, 06/2008, pp. 347–361 (cit. on pp. 14, 154, 158).

[DN08] Daniel Delling and Giacomo Nannicini. "Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks." In: *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*. Vol. 5369. Lecture Notes in Computer Science. Springer, 12/2008, pp. 813–824 (cit. on p. 139).

[DN12] Daniel Delling and Giacomo Nannicini. "Core Routing on Dynamic Time-Dependent Road Networks." In: *Informs Journal on Computing* 24.2 (2012), pp. 187–201 (cit. on pp. 12, 82, 83, 84, 85).

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. "Intriguingly Simple and Fast Transit Routing." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 43–54 (cit. on pp. 7, 115, 125, 126).

[DPW09a] Daniel Delling, Thomas Pajor, and Dorothea Wagner. "Accelerating Multi-Modal Route Planning by Access-Nodes." In: *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*. Vol. 5757. Lecture Notes in Computer Science. Springer, 09/2009, pp. 587–598 (cit. on pp. 6, 16, 133, 139, 142, 143, 146, 147, 152, 153).

[DPW09b] Daniel Delling, Thomas Pajor, and Dorothea Wagner. "Engineering Time-Expanded Graphs for Faster Timetable Information." In: *Robust and Online Large-Scale Optimization*. Vol. 5868. Lecture Notes in Computer Science. Springer, 2009, pp. 182–206 (cit. on pp. 14, 103, 111).

[DPW12a] Daniel Delling, Thomas Pajor, and Renato F. Werneck. "Round-Based Public Transit Routing." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 130–140 (cit. on pp. 14, 103).

[DPW12b] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. "User-Constrained Multi-Modal Route Planning." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 118–129 (cit. on p. 8).

[DPW14] Daniel Delling, Thomas Pajor, and Renato F. Werneck. "Round-Based Public Transit Routing." In: *Transportation Science* 49.3 (2014), pp. 591–604 (cit. on pp. 5, 6, 14, 103, 109, 111, 115, 125, 126, 127, 128, 153, 154, 157, 158, 159, 175, 176).

[DPW15] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. "User-Constrained Multi-Modal Route Planning." In: *ACM Journal of Experimental Algorithmics* 19 (04/2015), 3.2:1.1–3.2:1.19 (cit. on pp. 6, 8, 153).

[DPWZ09] Daniel Delling, Thomas Pajor, Dorothea Wagner, and Christos Zaroliagis. "Efficient Route Planning in Flight Networks." In: *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. OpenAccess Series in Informatics (OASIcs). 2009 (cit. on p. 133).

[Dre69] Stuart E. Dreyfus. "An Appraisal of Some Shortest-Path Algorithms." In: *Operations Research* 17.3 (1969), pp. 395–412 (cit. on pp. 9, 11, 72).

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. "Engineering Route Planning Algorithms." In: *Algorithmics of Large and Complex Networks*. Vol. 5515. Lecture Notes in Computer Science. Springer, 2009, pp. 117–139 (cit. on p. 139).

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies." In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 271–282 (cit. on pp. 7, 86).

[DSW15] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs." In: *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2015 (cit. on p. 13).

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies." In: *ACM Journal of Experimental Algorithmics* (2016). To appear. (cit. on pp. 7, 86).

[DW07] Daniel Delling and Dorothea Wagner. "Landmark-Based Routing in Dynamic Graphs." In: *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*. Vol. 4525. Lecture Notes in Computer Science. Springer, 06/2007, pp. 52–65 (cit. on pp. 11, 12).

[DW09a] Daniel Delling and Dorothea Wagner. "Pareto Paths with SHARC." In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*. Vol. 5526. Lecture Notes in Computer Science. Springer, 06/2009, pp. 125–136 (cit. on p. 13).

[DW09b] Daniel Delling and Dorothea Wagner. "Time-Dependent Route Planning." In: *Robust and Online Large-Scale Optimization*. Vol. 5868. Lecture Notes in Computer Science. Springer, 2009, pp. 207–230 (cit. on pp. 11, 12, 72).

[DW13] Daniel Delling and Renato F. Werneck. "Faster Customization of Road Networks." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 30–42 (cit. on pp. 27, 52).

[DW14] Daniel Delling and Renato F. Werneck. "Customizable Point-of-Interest Queries in Road Networks." In: *IEEE Transactions on Knowledge and Data Engineering* (2014). to appear (cit. on p. 152).

[EG08] David Eppstein and Michael T. Goodrich. "Studying (non-planar) road networks through an algorithmic lens." In: *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08)*. ACM Press, 2008, pp. 1–10 (cit. on pp. 10, 17).

[Ehr05] Matthias Ehrgott. *Multicriteria Optimization*. Springer, 2005 (cit. on p. 13).

[EKS14] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. "Parallel Bi-objective Shortest Paths Using Weight-Balanced B-trees with Bulk Updates." In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 111–122 (cit. on p. 13).

[EL11] Andrew Ensor and Felipe Lillo. *Partial order approach to compute shortest paths in multimodal networks*. Tech. rep. http://arxiv.org/abs/1112.3366v1, 2011 (cit. on p. 153).

[EP13] Alexandros Efentakis and Dieter Pfoser. "Optimizing Landmark-Based Routing and Preprocessing." In: *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. ACM Press, 11/2013, 25:25–25:30 (cit. on pp. 9, 11, 85).

[EPV15] Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. "SALT. A Unified Framework for All Shortest-Path Query Variants on Road Networks." In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Lecture Notes in Computer Science. Springer, 2015, pp. 298–311 (cit. on pp. 9, 152).

[FA04] Marco Farina and Paolo Amato. "A Fuzzy Definition of "Optimality" for Many-Criteria Optimization Problems." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part A* 34.3 (2004), pp. 315–326 (cit. on pp. 6, 131, 153, 155).

[FFKP15] Andreas Emil Feldmann, Wai Shing Fung, Jochen Könemann, and Ian Post. "A (1+ϵ) (1 + ϵ) -Embedding of Low Highway Dimension Graphs into Bounded Treewidth Graphs." In: *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*. Ed. by Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann. Vol. 9134. Lecture Notes in Computer Science. Springer, 2015, pp. 469–480 (cit. on p. 10).

[FG65] Delbert R. Fulkerson and O. A. Gross. "Incidence Matrices and Interval Graphs." In: *Pacific Journal of Mathematics* 15.3 (1965), pp. 835–855 (cit. on p. 22).

[FHS14] Luca Foschini, John Hershberger, and Subhash Suri. "On the Complexity of Time-Dependent Shortest Paths." In: *Algorithmica* 68.4 (04/2014), pp. 1075–1097 (cit. on pp. 4, 11, 12, 70, 72, 85).

[FNS14] Stefan Funke, André Nusser, and Sabine Storandt. "On k-Path Covers and their Applications." In: *Proceedings of the 40th International Conference on Very Large Databases (VLDB 2014)*. 2014, pp. 893–902 (cit. on p. 13).

[FS13] Stefan Funke and Sabine Storandt. "Polynomial-time Construction of Contraction Hierarchies for Multicriteria Objectives." In: *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX'13)*. SIAM, 2013, pp. 31–54 (cit. on p. 13).

[FWL12] Fletcher Foti, Paul Waddell, and Dennis Luxen. "A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale." In: *Proceedings of the 4th TRB Conference on Innovations in Travel Modeling*. Transportation Research Board, 2012 (cit. on pp. 18, 152).

[Gei10] Robert Geisberger. "Contraction of Timetable Networks with Realistic Transfers." In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Vol. 6049. Lecture Notes in Computer Science. Springer, 05/2010, pp. 71–82 (cit. on pp. 14, 109, 125, 126, 137, 138, 158).

[Gei11] Robert Geisberger. "Advanced Route Planning in Transportation Networks." PhD thesis. Karlsruhe Institute of Technology, 02/2011 (cit. on pp. 18, 152).

[Geo73] Alan George. "Nested Dissection of a Regular Finite Element Mesh." In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363 (cit. on pp. 4, 10, 18, 21).

[GH05] Andrew V. Goldberg and Chris Harrelson. "Computing the Shortest Path: A* Search Meets Graph Theory." In: *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*. SIAM, 2005, pp. 156–165 (cit. on pp. 9, 111).

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979 (cit. on p. 13).

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. "Route Planning with Flexible Objective Functions." In: *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*. SIAM, 2010, pp. 124–137 (cit. on pp. 13, 137).

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Better Landmarks Within Reach." In: *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*. Vol. 4525. Lecture Notes in Computer Science. Springer, 06/2007, pp. 38–51 (cit. on p. 9).

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. "Reach for A*: Shortest Path Algorithms with Preprocessing." In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. DIMACS Book. American Mathematical Society, 2009, pp. 93–139 (cit. on p. 9).

[GL78] Alan George and Joseph W. Liu. "A Quotient Graph Model for Symmetric Factorization." In: *Sparse Matrix Proceedings*. SIAM, 1978 (cit. on p. 23).

[GL89] Alan George and Joseph W. Liu. "The Evolution of the Minimum Degree Ordering Algorithm." In: *SIAM Review* 31.1 (1989), pp. 1–19 (cit. on pp. 62, 63).

[GLS+10] Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker. "Fast Detour Computation for Ride Sharing." In: *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*. Vol. 14. OpenAccess Series in Informatics (OASIcs). 2010, pp. 88–99 (cit. on pp. 18, 174).

[GPPR04] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. "Distance Labeling in Graphs." In: *Journal of Algorithms* 53 (2004), pp. 85–112 (cit. on pp. 9, 18).

[GRST12] Robert Geisberger, Michael Rice, Peter Sanders, and Vassilis Tsotras. "Route Planning with Flexible Edge Restrictions." In: *ACM Journal of Experimental Algorithmics* 17.1 (2012), pp. 1–20 (cit. on p. 13).

[GS10] Robert Geisberger and Peter Sanders. "Engineering Time-Dependent Many-to-Many Shortest Paths Computation." In: *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*. Vol. 14. OpenAccess Series in Informatics (OASIcs). 2010 (cit. on p. 18).

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks." In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Vol. 5038. Lecture Notes in Computer Science. Springer, 06/2008, pp. 319–333 (cit. on pp. 9, 17, 56, 135).

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. "Exact Routing in Large Road Networks Using Contraction Hierarchies." In: *Transportation Science* 46.3 (08/2012), pp. 388–404 (cit. on pp. 4, 6, 9, 11, 12, 17, 18, 20, 21, 36, 37, 40, 41, 43, 49, 52, 54, 56, 59, 60, 61, 81, 132, 135, 138, 143, 144, 153, 159, 160, 174).

[GT86] John R. Gilbert and Robert Tarjan. "The analysis of a nested dissection algorithm." In: *Numerische Mathematik* (1986) (cit. on p. 21).

[Gtfs] General Transit Feed Specification. `https://developers.google.com/transit/gtfs/` (cit. on pp. 143, 170).

[Gut04] Ronald J. Gutman. "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks." In: *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. SIAM, 2004, pp. 100–111 (cit. on pp. 9, 85, 175).

[HaC84] HaCon - Ingenieurgesellschaft mbH. `http://www.hacon.de`. 1984 (cit. on pp. 109, 143).

[Han79] Pierre Hansen. "Bricriteria Path Problems." In: *Multiple Criteria Decision Making – Theory and Application –*. Springer, 1979, pp. 109–127 (cit. on pp. 13, 154).

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. "Fast Point-to-Point Shortest Path Computations with Arc-Flags." In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. DIMACS Book. American Mathematical Society, 2009, pp. 41–72 (cit. on p. 9).

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4 (1968), pp. 100–107 (cit. on pp. 9, 13).

[HS15] Michael Hamann and Ben Strasser. *Graph Bisection with Pareto-Optimization*. Tech. rep. arXiv, 2015 (cit. on pp. 10, 17, 69, 73).

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. "Engineering Multilevel Overlay Graphs for Shortest-Path Queries." In: *ACM Journal of Experimental Algorithmics* 13.2.5 (12/2008), pp. 1–26 (cit. on pp. 9, 11, 17).

[HSWW06] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. "Combining Speed-up Techniques for Shortest-Path Computations." In: *ACM Journal of Experimental Algorithmics* 10.2.5 (2006), pp. 1–18 (cit. on p. 9).

[II87] H. Imai and Masao Iri. "An optimal algorithm for approximating a piecewise linear function." In: *Journal of Information Processing* 9.3 (1987), pp. 159–162 (cit. on pp. 12, 75).

[JP02] Sungwon Jung and Sakti Pramanik. "An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps." In: *IEEE Transactions on Knowledge and Data Engineering* 14.5 (09/2002), pp. 1029–1046 (cit. on pp. 9, 11, 17).

[JW13] Ruoming Jin and Guan Wang. "Simple, Fast, and Scalable Reachability Oracle." In: *Proceedings of the VLDB Endowment* 6.14 (2013), pp. 1978–1989 (cit. on pp. 15, 114, 115, 127).

[KK13] Hassan A. Karimi and Piyawan Kasemsuppakorn. "Pedestrian Network Map Generation Approaches and Recommendation." In: *International Journal of Geographical Information Science* 27.5 (2013), pp. 947–962 (cit. on p. 13).

[KK99] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." In: *SIAM Journal on Scientific Computing* 20.1 (1999), pp. 359–392 (cit. on p. 22).

[KLC12] Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo. "A Label Correcting Algorithm for the Shortest Path Problem on a Multi-Modal Route Network." In: *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*. Vol. 7276. Lecture Notes in Computer Science. Springer, 2012, pp. 236–247 (cit. on pp. 16, 153).

[KLPC11] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto Wolfler Calvo. "UniALT for Regular Language Constraint Shortest Paths on a Multi-Modal Transportation Network." In: *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*. Vol. 20. OpenAccess Series in Informatics (OASIcs). 2011, pp. 64–75 (cit. on pp. 16, 153).

[KLSV10] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. "Distributed Time-Dependent Contraction Hierarchies." In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Vol. 6049. Lecture Notes in Computer Science. Springer, 05/2010, pp. 83–93 (cit. on p. 18).

[KMP+15] Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. "Analysis and Experimental Evaluation of Time-Dependent Distance Oracles." In: *Proceedings of the 17th Meeting on Algorithm Engineering and Experiments (ALENEX'15)*. SIAM, 2015, pp. 147–158 (cit. on p. 12).

[KMP+16] Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. "Engineering Oracles for Time-Dependent Road Networks." In: *Proceedings of the*

18th Meeting on Algorithm Engineering and Experiments (ALENEX'16). SIAM, 2016 (cit. on pp. 12, 82, 83, 84, 85).

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. "Evolution and Evaluation of the Penalty Method for Alternative Graphs." In: *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. OpenAccess Series in Informatics (OASIcs). 2013, pp. 94–107 (cit. on p. 174).

[KSS+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. "Computing Many-to-Many Shortest Paths Using Highway Hierarchies." In: *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007, pp. 36–45 (cit. on p. 174).

[KST99] Haim Kaplan, Ron Shamir, and Robert Tarjan. "Tractability of Parameterized Completion Problems on Chordal, Strongly Chordal, and Proper Interval Graphs." In: *SIAM Journal on Computing* (1999) (cit. on p. 21).

[KWZ15] Spyros Kontogiannis, Dorothea Wagner, and Christos Zaroliagis. *Hierarchical Oracles for Time-Dependent Networks*. Tech. rep. Technical Report on arXiv. 2015 (cit. on p. 12).

[KZ14] Spyros Kontogiannis and Christos Zaroliagis. "Distance Oracles for Time-Dependent Networks." In: *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP'14)*. Vol. 8572. Lecture Notes in Computer Science. Springer, 07/2014, pp. 713–725 (cit. on p. 12).

[KZ15] Spyros Kontogiannis and Christos Zaroliagis. "Distance Oracles for Time-Dependent Networks." In: *Algorithmica* (2015), pp. 1–31 (cit. on p. 12).

[Lat91] Jean-Claude Latombe. *Robot Motion Planning*. Vol. 124. Springer International Series in Engineering and Computer Science. Springer, 1991 (cit. on p. 14).

[Lau04] Ulrich Lauther. "An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background." In: *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*. Vol. 22. IfGI prints, 2004, pp. 219–230 (cit. on p. 9).

[Lds] London Data Store. `http://data.london.gov.uk/` (cit. on pp. 109, 163).

[LR89] Michael Luby and Prabhakar Ragde. "A Bidirectional Shortest-Path Algorithm with Good Average-Case Behavior." In: *Algorithmica* 4.4 (1989), pp. 551–567 (cit. on p. 9).

[LRT79] Richard J. Lipton, Donald J. Rose, and Robert Tarjan. "Generalized Nested Dissection." In: *SIAM Journal on Numerical Analysis* 16.2 (04/1979), pp. 346–358 (cit. on pp. 10, 21).

[LS12] Dennis Luxen and Dennis Schieferdecker. *Doing More for Less – Cache-Aware Parallel Contraction Hierarchies Preprocessing*. Tech. rep. Karlsruhe Institute of Technology, 2012 (cit. on p. 18).

[MA04] Ellips Masehian and M. R. Amin-Naseri. "A Voronoi Diagram-visibility Graph-potential Field Compound Algorithm for Robot Path Planning." In: *J. Robotic Systems* 21.6 (2004), pp. 275–300 (cit. on p. 14).

[Mar84] Ernesto Queiros Martins. "On a Multicriteria Shortest Path Problem." In: *European Journal of Operational Research* 26.3 (1984), pp. 236–245 (cit. on p. 13).

[MCB14] Joris Maervoet, Patrick De Causmaecker, and Greet Vanden Berghe. "Fast Approximation of Reach Hierarchies in Networks." In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 11/2014 (cit. on pp. 9, 85, 175).

[Mei11] Sascha Meinert. "Engineering Data Generators for Robust Experimental Evaluations. Planar Graphs, Artificial Road Networks, and Traffic Information." PhD thesis. Fakultät für Informatik, Karlsruher Institut für Technologie (KIT), 12/2011 (cit. on p. 10).

[Met66] Metropolitan Transportation Authority of the State of New York. `http://www.mta.info/`. 1966 (cit. on p. 142).

[Mil12] Nikola Milosavljević. "On optimal preprocessing for contraction hierarchies." In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. ACM Press, 2012, pp. 33–38 (cit. on p. 10).

[MM12] Enrique Machuca and Lawrence Mandow. "Multiobjective Heuristic Search in Road Maps." In: *Expert Systems with Applications* 39.7 (06/2012), pp. 6435–6445 (cit. on p. 13).

[MMPZ13] Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, and Christos Zaroliagis. "A New Dynamic Graph Structure for Large-Scale Transportation Networks." In: *Proceedings of the 8th Conference on Algorithms and Complexity*. Vol. 7878. Lecture Notes in Computer Science. Springer, 05/2013, pp. 312–323 (cit. on pp. 14, 129, 175).

[MP10] Lawrence Mandow and José-Luis Pérez-de-la-Cruz. "Multiobjective A* Search with Consistent Heuristics." In: *Journal of the ACM* 57.5 (06/2010), 27:1–27:24 (cit. on p. 13).

[MS07] Matthias Müller–Hannemann and Mathias Schnee. "Finding All Attractive Train Connections by Multi-Criteria Pareto Search." In: *Algorithmic Methods for Railway Optimization*. Vol. 4359. Lecture Notes in Computer Science. Springer, 2007, pp. 246–263 (cit. on p. 14).

[MS10] Matthias Müller–Hannemann and Stefan Schirra, eds. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice*. Vol. 5971. Lecture Notes in Computer Science. Springer, 2010 (cit. on p. 1).

[MS14] Florian Merz and Peter Sanders. "PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies." In: *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Vol. 8737. Lecture Notes in Computer Science. Springer, 09/2014, pp. 701–712 (cit. on pp. 15, 114, 127).

[MSWZ07] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. "Timetable Information: Models and Algorithms." In: *Algorithmic Methods for Railway Optimization*. Vol. 4359. Lecture Notes in Computer Science. Springer, 2007, pp. 67–90 (cit. on pp. 14, 105, 108, 109, 110, 158).

[MW01] Matthias Müller–Hannemann and Karsten Weihe. "Pareto Shortest Paths is Often Feasible in Practice." In: *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*. Vol. 2141. Lecture Notes in Computer Science. Springer, 2001, pp. 185–197 (cit. on pp. 13, 14, 154).

[MW06] Matthias Müller–Hannemann and Karsten Weihe. "On the cardinality of the Pareto set in bicriteria shortest path problems." In: *Annals of Operations Research* 147.1 (2006), pp. 269–286 (cit. on p. 115).

[MW95] Alberto O. Mendelzon and Peter T. Wood. "Finding Regular Simple Paths in Graph Databases." In: *SIAM Journal on Computing* 24.6 (1995), pp. 1235–1258 (cit. on p. 15).

[MZ07] M. Mokhtarzade and M.J. Valadan Zoej. "Road Detection from High-Resolution Satellite Images Using Artificial Neural Networks." In: *International Journal of Applied Earth Observation and Geoinformation* 9.1 (2007), pp. 32–40 (cit. on p. 13).

[NDLS12] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. "Bidirectional A* Search on Time-Dependent Road Networks." In: *Networks* 59 (2012), pp. 240–251 (cit. on pp. 12, 78).

[Nic66] T. A. Nicholson. "Finding the Shortest Route between Two Points in a Network." In: *The Computer Journal* 9.3 (1966), pp. 275–280 (cit. on p. 9).

[OR90] Ariel Orda and Raphael Rom. "Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length." In: *Journal of the ACM* 37.3 (1990), pp. 607–625 (cit. on p. 11).

[OW88] M. H. Overmars and Emo Welzl. "New Methods for Computing Visibility Graphs." In: *Proc. 4th Annu. ACM Sympos. Comput. Geom.* 1988, pp. 164–171 (cit. on p. 92).

[Paj09] Thomas Pajor. "Multi-Modal Route Planning." MA thesis. Fakultät für Informatik, 03/2009 (cit. on p. 16).

[PBB+08] Dieter Pfoser, Sotiris Brakatsoulas, Petra Brosch, Martina Umlauft, Nektaria Tryfona, and Giorgos Tsironis. "Dynamic Travel Time Provision for Road Networks." In: *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08)*. ACM Press, 2008 (cit. on p. 11).

[PJPZ10] Ting Peng, Ian H. Jermyn, Veronique Prinet, and Josiane Zerubia. "Extended Phase Field Higher-Order Active Contour Models for Networks." English. In: *International Journal of Computer Vision* 88.1 (2010), pp. 111–128 (cit. on p. 13).

[Poh71] Ira Pohl. "Bi-directional Search." In: *Proceedings of the Sixth Annual Machine Intelligence Workshop*. Vol. 6. Edinburgh University Press, 1971, pp. 124–140 (cit. on p. 9).

[Pot88] Alex Pothen. *The complexity of optimal elimination trees*. Tech. rep. Pennsylvania State University, 1988 (cit. on p. 21).

[PSWZ04] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. "Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach." In: *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03)*. Vol. 92. Electronic Notes in Theoretical Computer Science. 2004, pp. 85–103 (cit. on p. 175).

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. "Efficient Models for Timetable Information in Public Transportation Systems." In: *ACM Journal of Experimental Algorithmics* 12.2.4 (2008), pp. 1–39 (cit. on pp. 6, 14, 103, 105, 109, 110, 115, 116, 120, 133, 138, 154, 157, 175).

[PTV79] PTV AG – Planung Transport Verkehr. *http://www.ptv.de*. 1979 (cit. on pp. 143, 163, 170).

[PV03] Scott Parker and Ellen Vanderslice. "Pedestrian Network Analysis." In: *Walk 21 IV*. Portland, OR, 2003 (cit. on pp. 13, 90).

[PWK12] Léon Planken, Mathijs de Weerdt, and Roman van Krogt. "Computing All-pairs Shortest Paths by Leveraging Low Treewidth." In: *Journal of Artificial Intelligence Research* (2012) (cit. on pp. 10, 18, 69).

[RT10] Michael Rice and Vassilis Tsotras. "Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions." In: *Proceedings of the VLDB Endowment* 4.2 (11/2010), pp. 69–80 (cit. on pp. 13, 16, 20).

[SABW13] Stephan Seufert, Avishek Anand, Srikanta Bedathur, and Gerhard Weikum. "FERRARI: Flexible and Efficient Reachability Range Assignment for Graph Indexing." In: *Proceedings of the 29th International Conference on Data Engineering*. IEEE Computer Society, 2013, pp. 1009–1020 (cit. on pp. 15, 114, 127).

[San09] Peter Sanders. "Algorithm Engineering – An Attempt at a Definition." In: *Efficient Algorithms*. Vol. 5760. Lecture Notes in Computer Science. Springer, 2009, pp. 321–340 (cit. on p. 1).

[SM13] Peter Sanders and Lawrence Mandow. "Parallel Label-Setting Multi-Objective Shortest Path Search." In: *27th International Parallel and Distributed Processing Symposium (IPDPS'13)*. IEEE Computer Society, 2013, pp. 215–224 (cit. on p. 13).

[Som14] Christian Sommer. "Shortest-Path Queries in Static Networks." In: *ACM Computing Surveys* 46.4 (2014) (cit. on pp. 1, 9).

[SOS98] Hanif D. Sherali, Kaan Ozbay, and Shivaram Subramanian. "The time-dependent shortest pair of disjoint paths problem: Complexity, models, and algorithms." In: *Networks* 31.4 (1998), pp. 259–272 (cit. on p. 71).

[SS05] Peter Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries." In: *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*. Vol. 3669. Lecture Notes in Computer Science. Springer, 2005, pp. 568–579 (cit. on p. 98).

[SS07] Dominik Schultes and Peter Sanders. "Dynamic Highway-Node Routing." In: *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*. Vol. 4525. Lecture Notes in Computer Science. Springer, 06/2007, pp. 66–79 (cit. on p. 11).

[SS12a] Peter Sanders and Dominik Schultes. "Engineering Highway Hierarchies." In: *ACM Journal of Experimental Algorithmics* 17.1 (2012), pp. 1–40 (cit. on p. 37).

[SS12b] Peter Sanders and Christian Schulz. "Distributed Evolutionary Graph Partitioning." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 16–29 (cit. on pp. 10, 17, 73).

[SS13] Peter Sanders and Christian Schulz. "Think Locally, Act Globally: Highly Balanced Graph Partitioning." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 164–175 (cit. on pp. 22, 69).

[SS15] Aaron Schild and Christian Sommer. "On Balanced Separators in Road Networks." In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Lecture Notes in Computer Science. Springer, 2015 (cit. on pp. 10, 17, 69, 73).

[SS88] J.T. Schwartz and M. Sharir. "A Survey of Motion Planning and Related Geometric Algorithms." In: *Artificial Intelligence* 37.1—3 (1988), pp. 157–169 (cit. on p. 13).

[Sto12] Sabine Storandt. "Route Planning for Bicycles – Exact Constrained Shortest Paths Made Practical Via Contraction Hierarchy." In: *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*. 2012, pp. 234–242 (cit. on p. 102).

[Stu12] Nathan Sturtevant. "Benchmarks for Grid-Based Pathfinding." In: *Transactions on Computational Intelligence and AI in Games* (2012) (cit. on p. 42).

[SW11] Peter Sanders and Dorothea Wagner. "Algorithm Engineering." In: *it—Information Technology* 53.6 (2011), pp. 263–265 (cit. on p. 1).

[SW14] Ben Strasser and Dorothea Wagner. "Connection Scan Accelerated." In: *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX'14)*. SIAM, 2014, pp. 125–137 (cit. on pp. 6, 15, 125, 126, 175).

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. "Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport." In: *ACM Journal of Experimental Algorithmics* 5.12 (2000), pp. 1–23 (cit. on pp. 9, 11, 14, 17, 159).

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. "Using Multi-Level Graphs for Timetable Information in Railway Systems." In: *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*. Vol. 2409. Lecture Notes in Computer Science. Springer, 2002, pp. 43–59 (cit. on pp. 9, 11, 14).

[TfL] Transport for London. `http://www.tfl.gov.uk/`. 2000 (cit. on p. 163).

[TW67] William F. Tinney and J.W. Walker. "Direct solutions of sparse network equations by optimally ordered triangular factorization." In: *Proceedings of the IEEE* 55.11 (11/1967), pp. 1801–1809 (cit. on p. 62).

[Weg14] Michael Wegner. "Finding Small Node Separators." Bachelor Thesis. Karlsruhe Institute of Technology, 2014 (cit. on p. 69).

[Wei10] Fang Wei. "TEDI: efficient shortest path query answering on graphs." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM Press, 2010 (cit. on pp. 10, 18).

[Wir15] Alexander Wirth. "Algorithms for Contraction Hierarchies on Public Transit Networks." MA thesis. Karlsruhe Institute of Technology, 2015 (cit. on p. 14).

[Wit15] Sascha Witt. "Trip-Based Public Transit Routing." In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*. Lecture Notes in Computer Science. Accepted for publication. Springer, 2015, pp. 1025–1036 (cit. on pp. 14, 103, 175).

[WLY+15] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. "Efficient Route Planning on Public Transportation Networks: A Labelling Approach." In: *Proceedings of the 2015 ACM SIGMOD International Con-*

ference on Management of Data (SIGMOD'15). ACM Press, 2015, pp. 967–982 (cit. on p. 114).

[WWZ05] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. "Geometric Containers for Efficient Shortest-Path Computation." In: *ACM Journal of Experimental Algorithmics* 10.1.3 (2005), pp. 1–30 (cit. on p. 9).

[WXD+12] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. "Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation." In: *PVLDB* 5.5 (2012), pp. 406–417 (cit. on p. 9).

[YAIY13] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. "Fast and Scalable Reachability Queries on Graphs by Pruned Labeling with Landmarks and Paths." In: *Proceedings of the 22nd International Conference on Information and Knowledge Management*. ACM Press, 2013, pp. 1601–1606 (cit. on pp. 15, 114, 115, 117, 127, 129, 175).

[Yan81] Mihalis Yannakakis. "Computing the minimum fill-in is NP-complete." In: *SIAM Journal on Algebraic and Discrete Methods* (1981) (cit. on p. 21).

[YCZ10] Hilmi Yildirim, Vineet Chaoji, and Mohammad J. Zaki. "GRAIL: Scalable Reachability Index for Large Graphs." In: *Proceedings of the VLDB Endowment* 3.1 (2010), pp. 276–284 (cit. on pp. 15, 114, 127).

[YL12] Haicong Yu and Feng Lu. "Advanced multi-modal routing approach for pedestrians." In: *2nd International Conference on Consumer Electronics, Communications and Networks*. 2012, pp. 2349–2352 (cit. on pp. 15, 153).

[Zad65] Lotfi A. Zadeh. "Fuzzy Sets." In: *Information and Control* 8.3 (1965), pp. 338–353 (cit. on p. 155).

[Zad88] Lotfi A. Zadeh. "Fuzzy Logic." In: *IEEE Computer* 21.4 (1988), pp. 83–93 (cit. on pp. 6, 131, 153, 155).

[Zei13] Tim Zeitz. "Weak Contraction Hierarchies Work!" Bachelor Thesis. Karlsruhe Institute of Technology, 2013 (cit. on pp. 4, 10, 18, 23, 49).

[ZLWX14] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. "Reachability Queries on Large Dynamic Graphs: A Total Order Approach." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM Press, 2014, pp. 1323–1334 (cit. on pp. 15, 114, 115, 127).

A

CURRICULUM VITÆ

Julian Dibbelt

born in Munich, Germany

EDUCATION

07/2011 – 02/2016 **PhD student in Informatics**
Karlsruhe Institute of Technology, Germany
Advisors: Prof. Dr. Dorothea Wagner
Prof. Dr. Christos Zaroliagis

02/2010 **Diploma in Informatics (M.Sc. equiv.)**
University of Karlsruhe, Germany
Specialization: Algorithmics and Algorithm Engineering, Cognitive Systems, Operations Research
Thesis: Alternative Routes in Road Networks

EXPERIENCE ABROAD

06/2014 – 09/2014 **Internship at Microsoft Research Silicon Valley,**
Mountain View, CA, United States
Supervisors: Daniel Delling, Thomas Pajor, and Renato F. Werneck;
Research on fast public transit route planning

TEACHING EXPERIENCE

04/2015 – 07/2015 Lecture "Algorithms for Route Planning"

10/2014 – 03/2015 Practical course "Algorithm Engineering"

04/2014 – 07/2014 Lecture "Algorithms for Route Planning"

10/2013 – 03/2014 Practical course "Algorithm Engineering"

04/2013 – 07/2013 Lecture "Algorithms for Route Planning"

10/2012 – 03/2013 Practical course "Algorithm Engineering"

10/2011 – 03/2012 Practical course "Algorithm Engineering"

SUPERVISED STUDENTS

STUDENT RESEARCH PROJECTS AND BACHELOR THESES

08/2013 – 11/2013 Janis Hamme, Customizable Route Planning in External Memory

06/2013 – 09/2013 Tim Zeitz, Weak Contraction Hierarchies Work!

03/2013 – 06/2013	L. Hübschle-Schneider, Speed–Consumption Trade-off for Electric Vehicle Routing
08/2012 – 11/2012	Simeon Andreev, Realistic Pedestrian Routing
01/2012 – 05/2012	Andreas Bauer, Multimodal Profile Queries
01/2012 – 05/2012	Jörg Weißbarth, Shortest-Path Cover on Restricted Graph Classes
01/2012 – 03/2012	Michael Nagel, Bounded Verification of an Optimized Shortest Path Implementation

DIPLOMA AND MASTER THESES

01/2015 – 06/2015	Valentin Buchhold, Fast Computation of Isochrones in Road Networks
12/2014 – 05/2015	Alexander Wirth, Algorithms for Contraction Hierarchies on Public Transit Networks
11/2014 – 04/2015	Simeon Andreev, Consumption and Travel Time Profiles in Electric Vehicle Routing
06/2014 – 11/2014	Tobias Zündorf, Route Planning for Electric Vehicles with Realistic Charging Models
03/2013 – 08/2013	Qi-Bai Zhu, Consideration of Toll Costs for Route Planning in Road Networks
04/2012 – 09/2012	Joan Reixach, Constraint Programming based Local Search for the Vehicle Routing Problem with Time Windows
02/2012 – 07/2012	Ben Strasser, Delay-Robust Stochastic Routing in Timetable Networks

B

LIST OF PUBLICATIONS

All conference and journal publications have been peer-reviewed.

JOURNAL ARTICLES

1 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies." In: *ACM Journal of Experimental Algorithmics* (2016). To appear.

2 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. "User-Constrained Multi-Modal Route Planning." In: *ACM Journal of Experimental Algorithmics* 19 (04/2015), 3.2:1.1–3.2:1.19.

3 Moritz Baum, Julian Dibbelt, Andreas Gemsa, and Dorothea Wagner. "Towards route planning algorithms for electric vehicles with realistic constraints." In: *Computer Science - Research and Development* (2014), pp. 1–5.

IN CONFERENCE PROCEEDINGS

1 Simeon Danailov Andreev, Julian Dibbelt, Martin Nöllenburg, Thomas Pajor, and Dorothea Wagner. "Towards Realistic Pedestrian Route Planning." In: *Proceedings of the 15th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'15)*. Vol. 48. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 09/2015, pp. 1–15.

2 Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. "Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles." In: *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2015.

3 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. "Public Transit Labeling." In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Lecture Notes in Computer Science. Springer, 2015, pp. 273–285.

4 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs." In: *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2015.

5 Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. "Speed-Consumption Tradeoff for Electric Vehicle Route Planning." In: *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*. Vol. 42. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 09/2014, pp. 138–151.

6 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies." In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 271–282.

7 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Delay-Robust Journeys in Timetable Networks with Minimum Expected Arrival Time." In: *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*. Vol. 42. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 09/2014, pp. 1–14.

8 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. "Energy-Optimal Routes for Electric Vehicles." In: *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2013, pp. 54–63.

9 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. "Computing Multimodal Journeys in Practice." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 260–271.

10 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. "Intriguingly Simple and Fast Transit Routing." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 43–54.

11 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. "User-Constrained Multi-Modal Route Planning." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 118–129.

TECHNICAL REPORTS

1 Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. *Fast Computation of Isochrones in Road Networks*. Tech. rep. 1512.09090. ArXiv e-prints, 2015.

2 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. *Dynamic Time-Dependent Route Planning in Road Networks with User Preferences*. Tech. rep. 1512.09132. ArXiv e-prints, 2015.

3 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs*. Tech. rep. abs/1509.03165. ArXiv e-prints, 2015.

4 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Customizable Contraction Hierarchies*. Tech. rep. abs/1402.0402. ArXiv e-prints, 2014.

5 Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. *Energy-Optimal Routes for Electric Vehicles*. Tech. rep. 2013-06. Faculty of Informatics, Karlsruhe Institute of Technology, 2013.

6 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. *Computing and Evaluating Multimodal Journeys*. Tech. rep. 2012-20. Faculty of Informatics, Karlsruhe Institute of Technology, 2012.

C

DEUTSCHE ZUSAMMENFASSUNG

Die schnellste Route durch den Verkehr, ein gut bewertetes Restaurant unterwegs, eine möglichst späte Zugfahrt, die einen noch rechtzeitig ans Ziel bringt, oder auch die bezahlbare Wohnung, mit sicherem Schulweg, in Fahrradreichweite zur Arbeit und guter Nahverkehrsanbindung – all diesen Szenarien liegen Kürzeste-Wege-Anfragen in verschiedenen Transportnetzwerken zugrunde.

Die weit verbreitete und tägliche Nutzung von web-basierten Kartendiensten, GPS-Navigation und anderen standortsbezogenen Dienstleistungen, die uns so zur Gewohnheit geworden ist, wurde ermöglicht durch umfangreiche Forschung auf dem Gebiet der Beschleunigung von Kürzeste-Wege-Anfragen.

Insbesondere ist die Entwicklung praxistauglicher Algorithmen zur Routenplanung in Transportnetzwerken ein Paradebeispiel für erfolgreiches Algorithm Engineering. Hierbei wird Entwurf, Analyse und Implementierung von Algorithmen durch experimentelle Evaluation auf Echtweltinstanzen vorangetrieben. Auf dem Gebiet der Routenplanung hat diese Methodik zahlreiche *Beschleunigungstechniken* hervorgebracht, die sich in Vorverarbeitungsdauer und -platz, Anfragegeschwindigkeit, aber auch Einfachheit ihrer Implementierung unterscheiden. Die meisten Ansätze folgen einem gemeinsamen Paradigma: Während der *Vorverarbeitung* werden Zusatzdaten berechnet, die anschließend den Suchraum nachfolgender Anfragen verkleinern (und somit diese Anfragen beschleunigen). Die schnellsten bekannten Techniken ermöglichen beweisbar korrekte Anfragen mit Hilfe weniger Speicherzugriffe.

Mittlerweile hat sich deswegen der Forschungsschwerpunkt in Richtung aufwendigerer und realistischerer Szenarien verschoben, die z.B. Benutzerpräferenzen, Fahrpläne, und multimodale Transportnetzwerke mit einbeziehen. Die Herausforderungen sind dabei: 1) Eine adäquate Formulierung der Problemstellung zu finden, und 2) Strukturen zu finden, die es auszunutzen gilt, um schnelle Anfragen zu ermöglichen. Beides sind Themenfelder der vorliegenden Arbeit: Sie führt neue Modellierungen und Beschleunigungstechniken ein für die Routenplanung auf Straßennetzwerken, Fahrplannetzwerken und die multimodale Optimierung.

STRASSENNETZWERKE. Die meisten webbasierten Systeme oder Standalone-Navigationsgeräte bieten nicht viel Raum für Benutzereinstellungen, von der Angabe der Start- und Zieladresse abgesehen natürlich. Aber außer der reinen Fahrzeitoptimierung, möchte mancher

Fahrer auch kompliziertes Abbiegen, enge Wohnstraßen, Autobahnen oder teure Mautbrücken, etc. vermeiden. Nicht nur, dass verschiedene Benutzer verschiedene Kostenfunktionen optimiert haben wollen, sogar der gleiche Benutzer könnte einen schnellen Weg am Morgen und einen sicheren am Abend bevorzugen. Der Hauptgrund dafür, dass feinkörnige Einstellungen oft nicht unterstützt werden, sind die prohibitiv hohen Vorverarbeitungskosten der implementierten Beschleunigungstechnik. (Neben der Schwierigkeit, trotz komplizierter Parametrisierung eine einfache Benutzerführung zu gestalten.)

Um benutzerdefinierte Einstellungen zu ermöglichen, wurde deshalb kürzlich vorgeschlagen, die teure Vorverarbeitung weiter zu unterteilen: in eine erste Phase, die ausschließlich die Topologie des Graphen nutzt, und eine zweite, schnelle Phase zur Anpassung an eine bestimmte Kostenfunktion, in der Literatur *Customization* genannt. Auf diese Idee bauen wir auf, wenn wir Contraction Hierarchies (eine gut etablierte Technik) um eine schnelle Customization erweitern. Desweiteren evaluieren wir einen neuen, schnelleren Anfrage-Algorithmus, der ohne Prioritätswarteschlange auskommt.

Während der vorgenannte Ansatz dynamische Kosten mit Leichtigkeit beachten kann, geht er (wie es typisch ist für zeitunabhängige Routenplanung) immer noch davon aus, dass die Kosten pro Straßensegment pro Anfrage konstant sind. In der Praxis wird die Fahrzeit deutlich beeinflusst durch die aktuelle Verkehrssituation im Straßennetz, die sich über den Tag ändert. Man kann unterscheiden zwischen Staus, die mit historischen Verkehrsdaten vorhersehbar sind (z. B. Berufsverkehr), und Staus aufgrund von unvorhersehbaren Ereignissen wie z. B. Unfällen. Wir untersuchen ein *dynamisch und zeitabhängiges* Routenplanungsproblem, das sowohl aktuellen Verkehr und Vorhersage berücksichtigt. Zu diesem Zweck schlagen wir einen praktischen Algorithmus vor, der in der Lage ist, netzwerkweit zugleich Benutzereinstellungen und die globalen Veränderungen der zeitabhängigen Metrik zu integrieren, schneller als bisherige Ansätze.

Andere Szenarien, wie z. B. Routen für Fußgänger, werden oft vernachlässigt oder als triviale Angelegenheit der Anwendung einer anderer Kostenfunktion abgetan. Stattdessen beobachten wir, dass Fußgänger-Routing eine spezifische Reihe von Problemen hat, die nicht von State-of-the-Art Routenplanern erfüllt werden. Zum Beispiel führt das Fehlen detaillierter Gehsteigdaten und die Unfähigkeit, Plätze und Parks auf eine natürliche Weise zu durchqueren, oft zu nicht ansprechenden und nicht optimalen Routen. Deshalb schlagen wir vor, das Netzwerk um Gehwege auf der Grundlage der Straßengeometrie und um Kanten zum Routing über Plätze und in Parks zu erweitern. Mit Hilfe dieser und weiterer Informationen, löst unser Ansatz nahtlos Anfragen, deren Ausgangs- oder Ziel eine beliebige Stelle im Straßennetzwerk, auf einem Platz oder in einem Park ist. Unsere Experimente zeigen, dass wir in der Lage sind, ansprechende

Fußgängerrouten zu berechnen, mit vernachlässigbarem Overhead gegenüber Standard-Routing-Algorithmen.

ÖFFENTLICHER FAHRPLANVERKEHR. Öffentliche Verkehrsnetze unterscheiden sich deutlich von Straßennetzen, da jedes Fahrzeug einem Fahrplan folgt und eine fest Sequenz von Haltestellen abfährt. Typische Routenplanungsprobleme im öffentlichen Verkehr betrachten entweder nur die Optimierung der frühesten Ankunftszeit oder sogar mehrere Kriterien wie Ankunftszeit und Anzahl der Umstiege, und können entweder für eine bestimmte Abfahrtszeit oder für einen ganzen Zeitbereich (Profilanfrage genannt) formuliert werden.

Dementsprechend ist, während es für die Straße oft eine einfache Charakterisierung als Kürzeste-Wege-Problem gibt, selbst die Erforschung von Basisalgorithmen (also noch vor dem Entwurf von Beschleunigungstechniken) noch nicht abgeschlossen für den öffentlichen Nahverkehr. Wir stellen einen neuartigen algorithmischen Ansatz vor, der Fahrten direkt auf dem Fahrplan berechnet (ohne Überführung in eine Graphrepräsentation). Dabei organisieren wir die Eingabe als Array von Elementarverbindungen (der Grundbaustein eines Fahrplans), die nur einmal pro Abfrage kontinuierlich gescannt werden. Trotz seiner Einfachheit ist unser Algorithmus sehr vielseitig und löst früheste Ankunfts- sowie multikriterielle Profilanfragen.

Die Entwicklung von Vorverarbeitungs-basierten Beschleunigungstechniken für öffentliche Verkehrsnetze hat sich als anspruchsvoller als für Straßennetze erwiesen: Aktuelle Ansätze erfordern entweder massive Vorverarbeitungsdauer oder bieten nur eine begrenzte Beschleunigung. Aufbauend auf jüngsten Ergebnissen zu Hub-Labeling (der derzeit schnellste Ansatz für Straßennetze) und durch Ausnutzung anwendungspezifischer Eigenschaften, entwickeln wir einfache und effiziente Algorithmen für früheste Ankunfts-, Profil- und multikriterielle Anfragen, die um Größenordnungen schneller als der Stand der Technik sind.

MULTIMODALE ROUTENPLANUNG. Schließlich untersuchen wir das Problem, multimodale Routen in integrierten Verkehrsnetzen zu finden, also eine Reise zwischen zwei beliebigen Orten, die uneingeschränktes Laufen, Autofahren, Radfahren und Fahrplan-basierte öffentliche Verkehrsmittel einbezieht.

Dabei ist es von entscheidender Bedeutung, auf die Vorlieben eines Benutzers in Hinblick auf die Wahl der Verkehrsmittel zu achten: Nicht jeder Verkehrsträgerwechsel könnte zu jedem Zeitpunkt der Reise eine Option darstellen. Im allgemeinen hat der Benutzer Einschränkungen an die Abfolge der Verkehrsmittel. Zum Beispiel könnte manch Nutzer bereit sein, ein Taxi zwischen zwei Zugfahrten zu nehmen, wenn es die Reise schneller macht. Andere bevorzugten es, öffentliche Verkehrsmittel am Stück zu verwenden. Wir bieten ein

multimodales Routenplanungssystem, das solche Einschränkungen als *Benutzereingabe* für jede *Anfrage* zulässt (und nicht bereits während der Vorverarbeitung festlegt).

Eine weitere, natürliche Lösung für das Problem ist es, multimodale multikriterielle Optimierung zu verwenden, um die Vielzahl der verfügbaren Reisemöglichkeiten zu erfassen (die dem Benutzer womöglich noch gar nicht bewusst sind). Vollständige multikriterielle Suche neigt jedoch dazu langsam zu sein und zu viele Lösungen von oft überraschend geringem Wert zu produzieren. In Bezug auf das letztgenannte schlagen wir vor, die Lösungen in einem Nachbearbeitungsschritt zu filtern. Wir verwenden dazu Techniken aus der Fuzzy-Logik, um die Signifikanz einer Lösung zu beurteilen. Wir untersuchen ferner mehrere (weiterhin multikriterielle) Heuristiken, die ähnliche Fahrten viel schneller berechnen. Unsere Experimente zeigen, dass dieser Ansatz die Berechnung von hochwertigen multimodalen Fahrten ermöglicht und dabei schnell genug für praktische Anwendungen ist, selbst in großen Ballungsgebieten.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of February, 2016 (`classicthesis` version 4.2).