Engineering Algorithms for Dynamic and Time-Dependent Route Planning

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Tim Zeitz

Tag der mündlichen Prüfung:	9.12.2022
Erste Referentin:	Prof. Dr. Dorothea Wagner
Zweiter Referent:	Prof. Dr. Matthias Müller-Hannemann

Acknowledgements

I want to start this thesis by thanking a number of people. First of all, I want to thank Dorothea Wagner for offering me a position at her group, for her supervision, her pragmatic management, and for establishing an environment where we could focus on research without worrying about much else. It was a privilege to do research under these conditions.

I want to thank my colleagues for making this such a fun place to work. A huge thank you to Jonas Sauer for countless scientifically fruitful DOS and counter-DOS attacks, for his proofreading, for his work on whiteboard animal species protection, for always answering the call of duty and for generally being an awesome office mate. I want to thank Torsten Ueckerdt for math consulting, Paul Jungeblut and Laura Merker for keeping the Maracuja in check, Marcel Radermacher for his input on approximation algorithm, Lukas Barth for bringing hacker culture to the group, Thomas Bläsius for hiring foosball-playing PhD students, Christopher Weyand and Marcus Wilhelm for pointing me at subpath optimality at a critical moment, Michael Zündorf for last-minute LaTeX and typography support, and last but definitely not least Michael Haman for getting me into research. I also want to thank the route planners Moritz Baum, Tobias Zündorf and Valentin Buchhold for many fruitful discussions and finding the fastest way to the cafeteria.[1] Further, I want to thank Ben Strasser for the research directions he gave me at the beginning of my PhD. These ideas proved immensely fruitful throughout this work. There are also a few colleagues whose work often goes vastly underappreciated. A huge thank you to Ralf Kölmel for managing our server infrastructure and making everything appear to just work™. Also, thank you to Lilian Beckert, Laurette Laufer, Tanja Wehrmann, and Isabelle Junge for doing all the office management and handling all the bureaucracy.

This work also greatly benefited from several proofreaders. As already mentioned, Jonas Sauer did much of that, but I also want to thank my colleagues Paul Jungeblut and Lars Gottesbühren. Further, thank you to Matthias Grundmann and Stephan Heidel for providing helpful feedback on the introduction.

I would also like to thank Alexander Kleff and Frank Schulz from PTV for the pleasant, informal cooperation, the interesting problems they suggested and for providing easy access to production-grade routing data for our research.

Lastly, I want to thank the people outside of academia who helped me through this project. I want to thank my parents for encouraging me to start my PhD and to follow through with it. Finally and most importantly, I want to thank my wife Ann-Sophie for her love and encouragement, her practical support, for enduring me in the times when this project did not go quite so well and for keeping me sane in insane times.

[1] Not over the bridge!

Abstract

Efficiently computing shortest paths is an essential building block of many mobility applications, most prominently route planning/navigation devices and applications. In this thesis, we apply the algorithm engineering methodology to design algorithms for route planning in dynamic (for example, considering real-time traffic) and time-dependent (for example, considering traffic predictions) problem settings. We build on and extend the popular Contraction Hierarchies (CH) speedup technique. With a few minutes of preprocessing, CH can optimally answer shortest path queries on continental-sized road networks with tens of millions of vertices and edges in less than a millisecond, i.e. around four orders of magnitude faster than Dijkstra's algorithm. CH already has been extended to dynamic and time-dependent problem settings. However, these adaptations suffer from limitations. For example, the time-dependent variant of CH exhibits prohibitive memory consumption on large road networks with detailed traffic predictions.

This thesis contains the following key contributions: First, we introduce CH-Potentials, an A*-based routing framework. CH-Potentials computes optimal distance estimates for A* using CH with a lower bound weight function derived at preprocessing time. The framework can be applied to any routing problem where appropriate lower bounds can be obtained. The achieved speedups range between one and three orders of magnitude over Dijkstra's algorithm, depending on how tight the lower bounds are. Second, we propose several improvements to Customizable Contraction Hierarchies (CCH), the CH adaptation for dynamic route planning. Our improvements yield speedups of up to an order of magnitude. Further, we augment CCH to efficiently support essential extensions such as turn costs, alternative route computation and point-of-interest queries. Third, we present the first space-efficient, fast and exact speedup technique for time-dependent routing. Compared to the previous time-dependent variant of CH, our technique requires up to 40 times less memory, needs at most a third of the preprocessing time, and achieves only marginally slower query running times. Fourth, we generalize A* and introduce time-dependent A* potentials. This allows us to design the first approach for routing with combined live and predicted traffic, which achieves interactive running times for exact queries while allowing live traffic updates in a fraction of a minute. Fifth, we study extended problem models for routing with imperfect data and routing for truck drivers and present efficient algorithms for these variants. Sixth and finally, we present various complexity results for non-FIFO time-dependent routing and the extended problem models.

Contents

1 Introduction

Recent years have seen a drastic increase in the usage of mobile navigation applications. Traditionally, route planning had to be done manually with a road atlas while listening to the traffic radio. Nowadays, mobile navigation applications find suitable routes within the blink of an eye. Due to the proper integration of live traffic data, one does not even need to worry about whether a detour around a traffic jam is worth it. Combined with advanced turn-by-turn guidance, this makes navigating road networks much more comfortable. It may even help to distribute the traffic load better over the network, to avoid further congestion, and thus to reduce emissions.

Integrating traffic data into the routing is crucial to obtaining "good" routes [DGPW17]. Even users knowing their way often use navigation applications because of the traffic information. Traffic data comes in two variants: On the one hand, there is data on the current traffic situation. This data is highly *dynamic* and depends on the current time of day. On the other hand, there are predictions on recurring traffic flows caused, for example, by commuters. To integrate traffic predictions into the routing, the travel time of a road segment can be considered dependent on the time of day when it is traversed. Such *time-dependent* predictions change less frequently, i.e. are usually not dynamic. In contrast, dynamic live traffic data does have a time-dependent component. Traffic jams will resolve at some point in the future. Planning a detour around congestion 400 kilometres away is not necessarily helpful. The subject of this thesis is the development of practical routing algorithms that integrate dynamic and time-dependent data.

For the practicality of routing algorithms, three central requirements have emerged: Interactive running times to compute routes, exactness with regards to the chosen problem model and a simple implementation. The history of navigation devices and routing applications demonstrates why these requirements are critical [DSSW09]. Classical navigation devices for cars have been commercially available since the 1980s. These devices applied heuristic methods to obtain

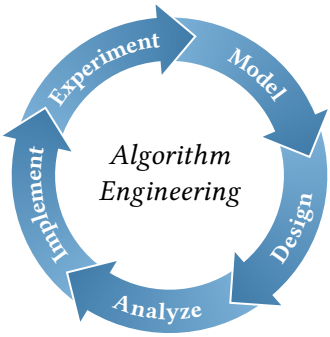


Figure 1.1: A visualization of the algorithm engineering methodology as introduced by [San09] and [MS10]. Algorithm engineering is a cyclic process consisting of modelling the problem, followed by the design and analysis of an algorithm, which is subsequently implemented and experimentally evaluated, before the next cycle begins. Figure kindly provided by Tobias Zündorf [Zün22].

routes in reasonable computation times [IOAI91]. This sometimes resulted in questionable route suggestions. In the early 2000s, the first results on *speedup techniques* for efficient and provably exact shortest path computation in road networks were published [SWW00, Lau04, GH05, SS05]. The core ingredient of these techniques is an offline preprocessing phase where some auxiliary data is computed from the road network. Utilizing this auxiliary data, shortest path computations can be accelerated significantly. Building on these still relatively complicated techniques, the first web-based navigation applications went online, most prominently Google Maps, and quickly became successful. Around 2010, several results were published which made speedup techniques significantly simpler [GSSD08, GSSV12] and more flexible [DSW16, DGPW17]. This enabled the widespread adoption of these techniques. Today, many commercial and non-commercial navigation applications and websites are built on these results.

While algorithms for route planning are already widely used, there is still room for improvement. For example, algorithms for time-dependent route planning are often very complicated and prohibitively resource-hungry. For dynamic route planning, the situation is better and efficient algorithms exist. However, even handling minor extensions to the problem can be surprisingly challenging. Designing extensible and flexible routing algorithms is an open challenge. The situation is especially unsatisfactory for routing considering real-time *and* predicted traffic. To the best of our knowledge, no efficient and exact algorithm for this problem has been presented in the literature. With this thesis, we aim to approach these problems and advance the state of the art in dynamic and time-dependent route planning algorithms. We do so using the algorithm engineering methodology.

Methodology. The development of algorithms aimed at practical requirements is the goal of the *algorithm engineering* research methodology [San09, MS10]. Algorithm engineering complements classic algorithm theory through experimental studies. The methodology is a specialized application of Popper's scientific method with its cycle of hypothesis building and experimental validation or falsification. It can be depicted as a cyclic process with five stages, as visualized in Figure 1.1. In the model stage, the practical problem is formalized. Routing problems in road networks can be formalized as the shortest path problem on directed, weighted graphs. This problem can be solved with the classical textbook algorithm of Dijkstra [Dij59],

and from a purely theoretical algorithms standpoint, there appears to be little room for improvement [Tho04]. However, the focus on practical applications allows for a crucial refinement of this simple model. Many shortest path queries will have to be answered on the same unchanging network for typical routing applications. Therefore, it may pay of to invest some time preprocessing the network and precomputing auxiliary data, which can help to accelerate shortest path computations. Thus, a refined problem model has two phases: An offline preprocessing phase and an online query phase. Designing and analyzing an efficient algorithm for this problem make up the next two steps of the methodology. These steps coincide with classical algorithm theory. However, the focus lies on practical performance rather than asymptotic worst-case bounds. Therefore, the algorithm is then implemented and optimized, taking into account the features of modern computer architectures, such as cache memory and parallelism. The final step is the experimental evaluation with realistic problem instances. Insights gained from the experiments may lead to new ideas for refinements of the model, the algorithms or the implementation, which may be realized following iterations of the methodology.

1.1 Related Work

The algorithm engineering methodology has been successfully applied in the context of navigation applications. A wide range of speedup techniques has been developed [Bas+16]. In the following, we highlight key results for the classical shortest path problem on road networks. Then, we discuss adaptions of these techniques, first, to dynamic and second, to time-dependent problem models. Third, we look at settings with combined dynamic and time-dependent data. Finally, we briefly touch on other route planning problems and applications.

1.1.1 Route Planning in Static Road Networks

A* [HNR68] is a classical approach to accelerate shortest path queries by directing the search towards the target through heuristic distance estimates. It is one of the algorithms fundamental to our work (a technical introduction will be given in Section 5.3). A* has been successfully employed in many problems beyond route planning [SHB14, Coh+18, BGHS19]. The performance of A* depends on the quality of the heuristic estimates. For example, while geographic distances may seem like a natural choice for distance estimates in road networks, the resulting heuristic is relatively ineffective: it performs worse than Dijkstra's algorithm [GH05]. A more accurate heuristic can be obtained with ALT [GH05, GW05], one of the early speedup techniques for routing in road networks. It uses precomputed distances to specific landmark vertices combined with the triangle inequality to compute distance estimates. On road networks, this achieves speedups of around two orders of magnitude over Dijkstra's algorithm.

Arc-Flags [Lau04, Lau06] is another speedup technique employing goal-direction. During preprocessing, the vertices are partitioned into several cells. Each edge is associated with a bitvector where the ith bit indicates whether the edge lies on any shortest path toward cell i. At query times, edges without the flag for the cell of the target vertex can be ignored. For long-

range queries, Arc-Flags obtains speedups of over three orders of magnitude over Dijkstra's algorithm. However, the preprocessing takes hours which is comparatively expensive and produces large amounts of auxiliary data.

Hierarchical speedup techniques utilize the inherent hierarchy in road networks. The critical observation is that large parts of the road network are only relevant to few shortest paths. Rural roads and residential areas are examples for such rarely relevant parts. In contrast, highways are relevant to almost all long-range routes. To a lesser extent, this also holds for country roads. Thus, there are many vertices which lie on few shortest paths and few vertices which lie on many shortest paths. The earliest approach explicitly utilizing this insight was Highway Hierarchies (HH) [SS05]. Highway Node Routing [SS07] improves on HH with a simpler query procedure and is significantly more space-efficient. *Contraction Hierarchies* (CH) [GSSV12] refines and simplifies the hierarchical approach further. Arguably, CH currently is the most popular speedup technique. It has been used successfully in many real-world applications, and several open-source implementations exist.[1] It also is a fundamental algorithm for this work. In the preprocessing step, the "importance" of vertices is determined heuristically. Additional shortcut edges are inserted, which allow skipping over less important vertices at query time. The process of inserting shortcuts skipping over a vertex is the name-giving *contraction*. On continental-sized road networks, this results in queries taking less than a tenth of a millisecond. This is more than four orders of magnitude faster than Dijkstra's algorithm. The preprocessing takes only a few minutes and produces little more auxiliary data than the input graph.

There are many works augmenting CH for extended problems. For example, the auxiliary data from CH preprocessing can be used for query types other than point-to-point. PHAST, a *one-to-all* CH query variant from one source vertex to *all* other vertices, was introduced in [DGNW13]. RPHAST [DGW11] is a PHAST extension for *one-to-many* queries to a subset of vertices known in advance. In [GV11], turn information is integrated into CH. Considering that a left turn often takes significantly longer than going straight ahead is critical for high-quality routing in cities. The proposed CH extension achieves fast queries, but preprocessing becomes an order of magnitude slower than classical CH. Another extension considers alternative route computation [ADGW13]. CH-based techniques will also be discussed in the following sections.

Another important hierarchical speedup technique is Multi-Level-Dijkstra (MLD). For MLD, important vertices are determined based on a multi-level partition of the network computed during preprocessing. MLD was initially used to accelerate shortest path computations on public transit networks [SWW00, SWZ02, HSW08] but variants specifically engineered to road networks have also been proposed [Del+06]. MLD only utilizes the network topology hierarchy but not the hierarchy in the edge weights. Therefore, it is much more robust against edge weights with a less pronounced hierarchy such as geographic distances. Many variants with different performance trade-offs exist. With the most recent results [DGPW17], queries are slightly slower than CH queries but still around three orders of magnitude faster than Dijkstra's algorithm. The memory consumption is even smaller than with CH.

[1] `https://gist.github.com/PayasR/bc46af938195a827e42006c3f5544e4a`

Hierarchical and goal-directed speedup techniques have also been combined [GKW07]. For example Core-ALT [Bau+10] is an accelerated ALT variant where the goal-directed search is only performed on a *core* of important vertices. This yields speedups of about three orders of magnitude compared to Dijkstra's algorithm. SHARC [BD09] combines Arc-Flags with shortcuts. This significantly improves the expensive preprocessing of Arc-Flags, and queries also become slightly faster. CHASE [Bau+10] combines CH with Arc-Flags for query times within a few tenths of microseconds.

The speedup technique currently known for achieving the fastest queries is Hub Labels (HL) [ADGW12, DGW13]. With HL, query times in less than a microsecond are possible. That is the same time as a few uncached memory accesses. These query speeds come at the cost of an expensive preprocessing phase and a tremendous memory footprint, more than an order of magnitude larger than the input graph.

Clearly, the design space for speedup techniques for computing shortest paths in road networks has been thoroughly explored. A variety of techniques with different trade-offs has been introduced. Interestingly, it is not the fastest techniques which have become most popular. For example, despite the extremely fast query times, it appears that the trade-off offered by HL is, in practice, often not very attractive. Simpler techniques like CH already offer queries fast enough to completely disappear behind other parts of practical applications (for example, network latency). We observe that simplicity and extensibility, while quite important in practice, have rarely been addressed explicitly in existing research. Therefore, in this thesis, we aim to design algorithms which do *not* solely optimize running times but are also extensible and straightforward to implement.

1.1.2 Dynamic Route Planning

Some speedup techniques were extended to dynamic scenarios. Highway Node Routing [SS07] supports single edge weight updates within a few milliseconds, complete metric updates within a few minutes and fast queries. However, since then, little development has happened on HNR because CH superseded it.

A*/ALT-based approaches were also among the early techniques [DW07] for dynamic route planning. However, because the heuristic is ALT-based, the accuracy of the estimates is limited, and the resulting query performance is not competitive. Further, the evaluation was performed with only synthetic traffic data. As we show in Chapter 8, the problem becomes significantly more challenging when using production-grade real-world traffic data. Using Core-ALT in a dynamic context was studied in [DN12]. Even though Core-ALT is significantly faster than pure ALT, the obtained speedups are not competitive with purely hierarchical techniques.

Customizable Route Planning (CRP) [DGPW17] is an engineered variant [DGPW11] of MLD which was developed to allow updating weights without invalidating the entire preprocessing. For this, a second, faster preprocessing phase is introduced, which takes at most a few seconds [DW13]. This phase is called the *customization*. It can be run regularly to update weights. This enables the integration of dynamic live traffic. Further, it is possible to model user prefer-

ences in a custom metric per user. Queries in CRP take at most a few milliseconds. Because CRP is built on MLD and utilizes a multi-level partition to determine the hierarchy, the performance is similar for any metric. CRP is one of the few examples of a speedup technique designed for flexibility rather than maximum query performance. For example, CRP was designed to support turn costs without additional modifications. Further, computing not only the shortest path but a set of reasonable alternatives is also possible in the CRP framework. Refined CRP-based methods for alternative routes have been extensively discussed in [Kob15]. Efficiently realizing Point-of-Interest queries in CRP has been studied in [DW15]. However, as discussed in the next section, the flexibility of CRP also has limits.

The three-phase setup has proven to be instrumental in supporting live traffic in practical applications [CP12]. Therefore, a CH variant operating in this three-phase setup also has been developed: *Customizable Contraction Hierarchies* (CCH) [DSW16]. CCH also uses a multi-level separator decomposition of the road network to obtain a CH order that only depends on the network topology and not on the edge weights. CCH queries are roughly as fast as CH queries, i.e. an order of magnitude faster than MLD/CRP. Improved customization algorithms were proposed in [BSW19]. Moreover, a CCH extension for efficient Point-of-Interest queries was presented in [BW21].

Both CRP and CCH are routing frameworks that have received considerable research interest. CRP, to this day, provides the most feature-complete routing framework and many essential extensions such as alternative routes and turn costs can be supported. CCH, in contrast, although conceptually more straightforward, is less flexible and fewer extensions have been proposed. For example, neither turn costs nor computing alternative routes have been studied in a CCH context. Still, CCH query times are somewhat faster than CRP. We conclude that making CCH more extensible would be a worthwhile endeavour to advance dynamic route planning.

1.1.3 Time-Dependent Route Planning

Several time-independent speedup techniques have been generalized to the time-dependent setting. However, dealing with travel time functions instead of scalar weights makes the preprocessing much harder and leads to difficult trade-offs. ALT [GH05] has been generalized to TD-ALT [NDSL12] and successively extended with vertex contraction to TD-CALT [DN12]. Even combined with approximation, TD-CALT queries may take longer than 10 ms on continental-sized graphs. SHARC [BD09] was extended to the time-dependent scenario [Del11]. Additionally, it can be combined with ALT into L-SHARC [Del11]. There also is a heuristic SHARC variant [Del11] which can find short paths in less than a millisecond. However, the preprocessing becomes quite expensive and takes hours, even on country-sized networks.

Both CH and MLD have been extended to time-dependent routing. MLD/CRP [DGPW17] has been extended to TD-CRP [BDPW16]. Like CRP, TD-CRP follows a three-phase approach and has a relatively fast customization phase. However, achieving reasonable memory consumption and query times was only possible by giving up exactness. Further, TD-CRP can only compute approximate shortest distances rather than paths.

There are several approaches based on Contraction Hierarchies [GSSV12, Bat14]. Three were introduced by Batz et al. in [BGSV13]: Time-dependent CH (TCH), inexact TCH, and Approximated TCH (ATCH). TCH has fast queries but a costly preprocessing phase taking several hours and may produce prohibitive amounts of auxiliary data, i.e. hundreds of Gigabytes. The amount of auxiliary data can be reduced at the cost of exactness (inexact TCH) or query performance (ATCH). An open-source reimplementation of [BGSV13] named KaTCH[2] exists. A simple heuristic named Time-Dependent Sampling (TD-S) was introduced by Strasser [Str17]. It samples a fixed set of weight functions with scalar values from the time-dependent functions. For each of these weight functions, a time-independent CH is constructed. To answer queries, a shortest path in each sample is computed. These paths are then combined into a small subgraph, and a non-accelerated time-dependent query is performed on the subgraph. This simple approach requires only manageable auxiliary data and produces surprisingly good results but is, of course, not exact.

Another approach is FLAT [Kon+16] and its extension CFLAT [Kon+17]. CFLAT features sublinear query running time after subquadratic preprocessing and guarantees a maximum approximation error. It uses timestamped combinatoric structures to represent the changes in shortest paths over time and computes travel times lazily. Unfortunately, preprocessing takes long in practice and generates prohibitive amounts of auxiliary data.

We observe that all existing approaches either produce prohibitive amounts of auxiliary data during preprocessing, have slow queries or are not exact. However, all these properties are essential in practice. Therefore, there is still significant room for making time-dependent route planning more practical.

1.1.4 Dynamic and Time-Dependent Route Planning

The combination of dynamic and time-dependent traffic information has not received as much research interest as the individual problems. Dynamic updates to time-dependent information have been studied for both ALT [DW07] and Core-ALT [DN12]. However, even for Core-ALT, running times are only competitive for approximate queries.

TD-CRP [BDPW16] has a customization phase where time-dependent travel time functions may be exchanged. In the evaluation in [BDPW16], the customization takes a fraction of a minute, which would be fast enough for regularly integrating real-time traffic updates. However, the evaluation was only performed on relatively old instances with little traffic prediction information. It is unclear how well this customization algorithm scales to instances with more and more complex predictions. Also, queries are inexact without any approximation guarantee, and path retrieval is not supported.

The TD-S heuristic [Str17] employs a different approach to model real-time traffic. The predicted time-dependent travel times remain unchanged. However, queries do not search for shortest paths for the predictions but with respect to a new travel time function which combines

[2] `https://github.com/GVeitBatz/KaTCH`

real-time traffic and traffic predictions. In this combined function, it depends on the evaluation time, whether the predicted or the real-time travel time is used. The real-time information is used only when the evaluation time is sufficiently close to the time the real-time data was observed. Still, TD-S is only a heuristic and will not necessarily find shortest paths.

Another approach with a similar problem model is described in a Google Patent [Gei15]. The algorithm proposed in the patent starts with an unaccelerated Dijkstra search on the network with real-time traffic information. This search runs until the distances exceed the assumed validity of the real-time traffic (for example, 30 minutes). Then, a (possibly time-dependent) CH search is performed from many sources to the target. The CH search sources are the remaining active vertices in the queue of the Dijkstra search with their respective distances. This algorithm finds exact shortest paths. However, its performance depends on the assumed validity time for the real-time traffic information. Further, it inherits all performance problems of TCH mentioned in the previous section. Moreover, to the best of our knowledge, no research paper introduces it in greater detail and evaluates it compared to other approaches.

We conclude that an efficient and exact technique for routing in a setting with combined predicted and real-time traffic is still an open problem.

1.1.5 Other Extended Route Planning Problems

Many problem extensions other than dynamic and time-dependent routing have been considered in the literature. For example, electric vehicle routing has been studied extensively [EFS11, Bau18, Bau+19b, Bau+20]. Multi-criteria optimization is another problem variant which appears in many applications [GKS10, FNS16]. Routing with incomplete and noisy traffic data has been studied in [DSS18]. Finding reliable routes with stochastic routing has also received considerable research interest [Kob15, PYJ20]. Another topic which introduces interesting challenges is truck driver routing. Traversing certain roads with a truck may only be allowed at certain times. Further, there are regulatory driving time constraints. Finding appropriate parking locations for mandatory breaks may also be challenging. These problems have been studied in [TWB18].

So far, we have only discussed routing in road networks. While beyond the scope of this work, we still want to point out that journey planning in public transit networks is also a relevant routing problem and an area of active research [Bau+19a, Zün22]. Sometimes the problem is even modelled as a time-dependent shortest path problem [PSWZ07, KMPZ22]. However, public transit networks have a fundamentally different structure than road networks. Thus, timetable-based approaches are often more effective [DPSW13, DPW15, Wit15]. Therefore, in this work, we focus only on road networks.

Another related problem outside the scope of this work is the Vehicle Route Planning (VRP) problem. However, the problem is more related to the traveling salesman problem than to the point-to-point shortest path problems we study in this thesis. Time-dependent shortest path algorithms are sometimes used as a subroutine in time-dependent vehicle routing problems [HZVG17]. For an overview over time-dependent VRP and other time-dependent routing problem variants we refer to [GGG15].

1.2 Contribution and Outline

This thesis aims to advance routing algorithms in dynamic and time-dependent problem settings by applying the algorithm engineering methodology. It is organized into three parts. The first part deals with translating practical navigation application problems into well-defined theoretical computer science problems. In Chapter 3, we discuss our problem models for dynamic and time-dependent routing problems. The chapter also includes some refined complexity results on time-dependent shortest path problems. Chapter 4 contains a detailed discussion of our routing data sources and how we extract benchmark instances to evaluate our algorithms.

The second part focuses on speedup techniques for shortest path computations. Chapter 5 gives a formal introduction to the existing algorithms and techniques on which we build. In Chapter 6, we introduce CH-Potentials, a novel CH-based A* heuristic. In contrast to most other speedup techniques, we design this approach for extensibility rather than maximum query performance. The result is a surprisingly flexible building block. On the one hand, CH-Potentials is an effective speedup technique on its own and can be applied directly to all problem models discussed in this work. On the other hand, it is an extremely useful subroutine, and we utilize it in all but one of the following chapters. Chapter 7 discusses the Customization Contraction Hierarchies framework. Its contribution is threefold: We review recent advances on CCH, present several incremental improvements, and provide an extensive evaluation of the current state of the art of CCH. Our improvements yield speedups of up to an order of magnitude for the different phases. As a result, CCH running times are now competitive with both CH and CRP. Further, with our extensions, CCH now supports all necessary features for a practical route planning framework. The refined CCH algorithms are also an essential ingredient in the following three chapters. In Chapter 8, we present Customizable Approximated Time-Dependent Contraction Hierarchies through Unpacking (CATCHUp), a speedup technique for time-dependent routing. To the best of our knowledge, it is the first technique that is space-efficient, fast and exact at the same time. In Chapter 9, we introduce a time-dependent generalization of A* potentials. We then combine the results from the previous three chapters and present two efficient realizations of such time-dependent potentials. We use these potentials to design the first exact technique for route planning with combined predicted and real-time traffic information with interactive query performance.

In the third part, we prove the extensibility of the techniques from the second part and apply them to extended problem formulations. In Chapter 10, we study shortest *smooth* paths. This problem model aims to avoid undesired detours due to incomplete and noisy real-time traffic data. We settle its theoretical complexity and present exact and heuristic algorithms outperforming the state of the art by around two orders of magnitude. Previously, this problem has only been studied for dynamic route planning. Using our time-dependent A* potentials, we can also approach this problem with predicted traffic and combined traffic. Finally, in Chapter 11, we present a problem model for truck routing which integrates time-dependent temporary driving bans and necessary parking breaks. We also give a CH-Potentials-based exact algorithm that can answer realistic queries in less than a second, enabling practical applications.

2 Preliminaries and Notation

In this chapter, we introduce fundamental concepts and notation used throughout this work. Our notation follows a few basic conventions such that the formatting of symbols indicates the type of the represented object. For example, we use calligraphic letters $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ to represent *sets*. Capital letters $A, B, C, \ldots$ indicate tuples, sequences or intervals. Lower case letters $a, b, c, \ldots$ are used primarily for elementary variables. We use Greek letters $\alpha, \beta, \gamma, \ldots$ for parameters of instances, problems or algorithms. However, there are a few exceptions where some Greek letters have a well-established meaning, for example, ε as an infinitesimal small number. When symbols represent concrete objects in an implementation context instead of abstract mathematical objects, we indicate this by using a monospace font $\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots$. For example `D[i]` would represent the value of an array `D` at index `i`.

We consider directed graphs $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ with $n = |\mathcal{V}|$ vertices and $m = |\mathcal{E}|$ edges. We use uv as a short notation for an edge from a *tail* vertex u to a *head* vertex v. The reversed graph $\overleftarrow{G} = (\mathcal{V}, \overleftarrow{\mathcal{E}})$ contains a reversed edge vu for every edge $uv \in \mathcal{E}$. The *neighborhood* $\mathcal{N}(v)$ of a vertex v is the set of adjacent vertices u such that there is an edge from v to u. Further, we denote the number of neighbors $\deg(v) = |\mathcal{N}(v)|$ as the *degree* of v. The *undirected neighborhood* $\overleftrightarrow{\mathcal{N}}(v)$ of a vertex v is the set of adjacent vertices u in terms of edges in either direction, i.e. $\overleftrightarrow{\mathcal{N}}(v) = \{u \mid uv \in \mathcal{E} \vee vu \in \mathcal{E}\}$ and $\overleftrightarrow{\deg}(v)$ the corresponding *undirected degree*. Note that we mainly work with *simple* graphs throughout this work, i.e. there are no multiedges and no loops. However, there are a few exceptions which we will address explicitly. Nevertheless, to ease notation, we will provide basic definitions only in terms of simple graphs.

A sequence of vertices $P = (v_1, \ldots, v_k)$ where $v_i v_{i+1} \in \mathcal{E}$ is called a *path*. Paths may be non-simple, i.e. contain the same vertex multiple times. We denote by $P_{i,j} = (v_i, \ldots, v_j), 1 \leq i < j \leq k$ a *subpath* of P. If it is clear that a path is simple and more convenient to notate we will

also use $P_{v_i,v_j} = P_{i,j}$ as a subpath notation. The *concatenation* of two paths is written as $(u,\dots,v)\cdot(v,\dots,w) = (u,\dots,v,\dots,w)$.

Edge length functions $\ell : \mathcal{E} \to (\mathbb{R} \to \mathbb{R}^{\geq 0})$ (sometimes also called *weights*) map edges to time-dependent functions, which in turn map a departure time τ at the tail u to a travel time $\ell(uv)(\tau)$. To simplify notation, we will often write $\ell(uv,\tau)$. When it is clear from the context that we are writing about *constant* functions, we omit the time argument and write $\ell(uv)$. For time-*in*dependent length functions, we also define the corresponding reversed weight functions: $\overleftarrow{\ell}(vu) = \ell(uv)$. From a time-dependent length functions, we can obtain constant lower and upper bound functions $\underline{\ell}(e) = \min_{\tau\in\mathbb{R}} \ell(e,\tau)$ and $\overline{\ell}(e) = \max_{\tau\in\mathbb{R}} \ell(e,\tau)$. Note that even though we defined length functions for the most general case on $\mathbb{R}$, we restrict ourselves to integer-defined functions whenever possible. This primarily simplifies the implementation but also has complexity implications as we show in the next chapter.

We extend the definition of length functions to paths. The length/travel time of a path $P = (v_1,\dots,v_k)$ is defined recursively $\ell(P,\tau) = \ell(v_1v_2,\tau) + \ell((v_2,\dots,v_k),\tau + \ell(v_1v_2,\tau))$. The travel time of a path with only one vertex (or less) is zero. A path's travel time can be obtained by successively evaluating travel times of the edges of the path.

We are interested in computing *shortest* paths. A *shortest path* from s to t for the departure time τ^{dep}, is a path such that no other st path departing at the same time has a shorter length. We denote the length of such a shortest path as the *distance* $\text{dist}(s,t,\tau^{\text{dep}})$. When there are multiple relevant length functions, for example ℓ_{pred} and ℓ_{live}, we write $\text{dist}_{\text{pred}}$ and $\text{dist}_{\text{live}}$ to denote the respective shortest distances. When referring to distances with respect to constant bounds $\underline{\ell}$ or $\overline{\ell}$ of a time-dependent length function ℓ, we will use the notation $\underline{\text{dist}}$ and $\overline{\text{dist}}$, respectively. The time-dependent *function* of shortest travel times $f(\tau) = \text{dist}(s,t,\tau)$ is denoted as the *travel time profile* of s and t.

For any time-dependent travel time function f, we define the respective *arrival time function* $\hat{f}(\tau) = f(\tau) + \tau$. This often simplifies the notation when dealing with the travel time functions of paths. For example, for two edges uv and vw, the travel time function for traversing them successively is $\ell(uv,\tau) + \ell(vw,\tau + \ell(uv,\tau))$ while in arrival time representation the result is just the composition $\hat{\ell}(vw,\hat{\ell}(uv,\tau))$. However, to simplify notation further for travel time functions, we will simply write $\ell(uv) \oplus \ell(vw)$ to denote the travel time function of first traversing uv and then vw, i.e. $(\ell(uv)\oplus\ell(vw))(\tau) = \ell(uv,\tau) + \ell(vw,\tau+\ell(uv,\tau))$. We write $f|_T$ to denote a partial function equal to f for any $\tau \in T$, and otherwise undefined. In a slight abuse of notation, we use the union operator to denote the combination of partial functions, i.e. $f|_T \cup f|_{T'} = f|_{T\cup T'}$.

In this work, time-dependent travel time functions for graphs are typically *periodic piecewise linear functions* (PPLF) represented by a sequence of breakpoints. They are defined on a limited time interval which we call *horizon* H. Typically, the horizon covers one day. We denote the number of breakpoints in piecewise linear function f as the complexity $|f|$.

Part I

Modelling

3 Formalizing Routing Problems

A properly formalized problem is essential not only to the algorithm engineering methodology but to any algorithmic study. Therefore, we begin this thesis by discussing problem models for routing applications. This chapter focuses on the theoretical side. The complexity proofs in Section 3.4 are based on our article [Zei22b] published in Information Processing Letters. In the second chapter of this part, we discuss available routing data sets and how we derive realistic problem instances from them, which enable practically conclusive experimental evaluations.

The core task of navigation applications is to find "good" routes between two locations. This task admits a very natural problem model. The road network is modelled as a directed graph. Vertices represent intersections. Road segments are modelled as edges. Edges are weighted by their traversal time. Matching the source and target locations onto respective vertices (or positions on edges), we can obtain a route by solving the classical *shortest path problem* (SPP).

Definition 3.1 (ShortestPathProblem). *Given a weighted, directed graph* $G = (\mathcal{V}, \mathcal{E})$ *with a non-negative scalar edge length function* $\ell : \mathcal{E} \to \mathbb{Z}^{\geq 0}$ *and vertices* s *and* t*, obtain a path from* s *to* t *in* G *of length* $\operatorname{dist}(s, t)$.

This is a classical problem of theoretical computer science, and it has been studied since at least the 1950s. It can be solved efficiently with well-known algorithms such as the one by Bellman and Ford [Bel58] or, most importantly for this thesis, the algorithm of Dijkstra [Dij59]. Even though it is already more than half a century old, it is still considered the practically fastest algorithm for this general problem [Bas+16]. Nevertheless, Dijkstra's algorithm is too slow on realistic problem instances such as the ones we will consider throughout this thesis. It takes seconds to answer point-to-point shortest path queries on continental road networks. This is infeasible for practical applications. Fortunately, improvements are possible when considering

a more realistic model based on a simple observation: Practical applications do not answer singular, isolated shortest path queries. Instead, many shortest path queries will be answered on the same rarely changing graph. This allows a problem formalization with two *phases*.

During an offline *preprocessing* phase, the graph $G = (\mathcal{V}, \mathcal{E})$ with its weights ℓ is given. A preprocessing algorithm may compute auxiliary data, possibly taking a long (but still reasonable) time. The second phase is the online *query* phase. Here, many source-target pairs $s, t \in \mathcal{V}$ are provided, and the shortest distances on G must be computed quickly. Crucially, the query algorithm may utilize the auxiliary data from the preprocessing to accelerate shortest path computations. We denote this as the *two-phase shortest path problem*.

The Case for Exactness. In this thesis, we go to great lengths to solve problems exactly. While ad-hoc algorithms might be practically tempting, we believe that an adequately formalized problem solved to exactness has great practical value. First, proper formalization is a prerequisite to any theoretical study. Without a problem model, there is nothing to examine. Second, a problem model allows a separation of concerns between routing data and algorithms. Each can be refined independently. Third, implementations of exact algorithms are debuggable. If an implementation produces unexpected results, the results can be verified. There might be a bug in the implementation. Alternatively, there might be artefacts in the data causing the results. It might even turn out that the problem formalization needs refinement. Nevertheless, in any case, there is a clear path forward. Fourth and finally, obtaining exact results helps to present users with a consistent worldview where travel times do not suddenly become better when the user takes an unexpected alternative.

Of course, a chosen model sometimes appears infeasible to solve in a reasonable timeframe. In that case, heuristic algorithms might be an option, ideally with approximation guarantees. However, we only employ this strategy as a last resort. Often, a better alternative is to try to find a refined, feasible problem model, like, for example, the two-phase model.

3.1 Dynamic Route Planning

One focus of this thesis is *dynamic* route planning. More specifically, our goal is to integrate dynamic real-time traffic information into our route planning algorithms. The underlying problem is still the SPP. However, the simple two-phase model becomes problematic. The assumption that the graph $G = (\mathcal{V}, \mathcal{E})$ is known at preprocessing time and will not change for the queries is still valid. However, the edge lengths ℓ representing travel times will change due to the current traffic situation. We consider two different formalizations for this problem.

In the first model, we want *instantaneous* traffic updates. Here, the edge weights are fully dynamic and may change for any query. From an application perspective, this approach is most flexible and yields accurate and up-to-date results at the moment of the request. However, from an algorithm design perspective, the options to design speedup techniques are somewhat limited. In a preprocessing phase, only the graph topology $G = (\mathcal{V}, \mathcal{E})$ could be utilized. However, we

can also derive a model refinement based on the observation that traffic will only increase travel times. For preprocessing, we can consider travel times without traffic, i.e. the preprocessing also has access to a *free-flow* edge length function ℓ_{free}. On the query side, the *query weights* ℓ_q may change arbitrarily between queries but must adhere to one constraint: query travel times must not be faster than the free-flow travel times, i.e. $\ell_q(e) \geq \ell_{\text{free}}(e)$ for every edge $e \in \mathcal{E}$. We denote this model as the *two-phase shortest path problem with dynamic traffic weights*.

We also consider a second model for live traffic built around *batched traffic updates*. Here, we accept a slight delay before traffic updates must be reflected in the shortest paths. This delay can be used for a second, faster preprocessing phase which we call the *update* phase. This phase is often called *customization* in the literature. In this work, we distinguish between the *update phase* and the *customization algorithms* as we sometimes apply customization algorithms in other phases. We call this the *three-phase shortest path problem*. The advantage of this model is that we do not need any additional constraints on the weights. In the preprocessing phase, only the graph topology $G = (\mathcal{V}, \mathcal{E})$ is given. The edge length function ℓ is provided in the update phase. Finally, queries must be answered quickly for vertex pairs provided online on G for the current edge length function ℓ.

Dynamic route planning models provide excellent results for short-range queries. However, for long-range queries where the user may drive for several hours, the traffic information used to determine the route may be outdated when the user reaches later parts of the route. A pragmatic mitigation for this problem is repeatedly recomputing the best path for the remaining route while underway. However, sometimes a routing decision in the current moment has far-reaching implications, for example deciding between a northern and a southern route from Karlsruhe to Berlin. In such cases, a purely dynamic route planning model is not ideal.

3.2 Time-Dependent Route Planning

The other focus of this thesis is *time-dependent* route planning. Time-dependent routing is primarily aimed at integrating traffic predictions. With traffic predictions, the travel time of a road segment depends on the time at which it is traversed. Therefore, the classical shortest path problem is insufficient to model this problem. Thus, we consider an extended problem formulation, the *time-dependent shortest path problem* (TD-SPP):

Definition 3.2 (TimeDependentShortestPathProblem$_T$)**.** *Given is a weighted, directed graph $G = (\mathcal{V}, \mathcal{E})$ with time-dependent edge length functions $\ell : \mathcal{E} \to (T \to T^{\geq 0})$ defined over a domain T, vertices s and t, and a departure time $\tau^{\text{dep}} \in T$. Obtain a path in G departing at s at instant τ^{dep} and arriving at the earliest possible time* $\text{dist}(s, t, \tau^{\text{dep}})$ *at t.*

This problem statement deliberately leaves out two crucial details: the representation of the travel time functions and the time domain T. Both have significant consequences as they determine which operations can be implemented efficiently for travel time functions. Previous works have established *periodic piecewise linear functions* (PPLF) as the standard function class used in time-dependent routing. The sorted sequence of breakpoints is a compact and flexible

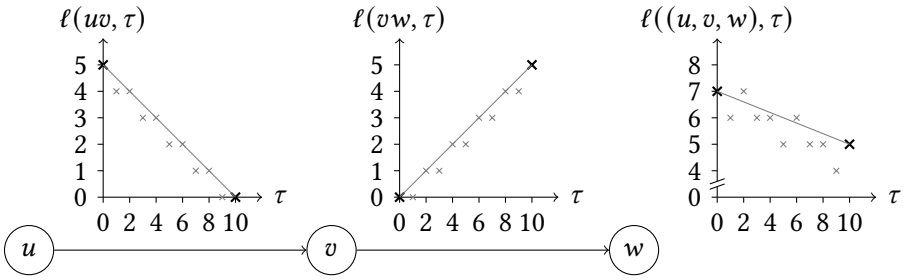


Figure 3.1: Example of the travel time of a path in either the integer or the rational/real domain. The thick black crosses are the interpolation points of the piecewise linear functions. Interpolating linearly with integer division between the breakpoints of the path travel time function leads to different results than successively evaluating the travel time functions of the path edges.

representation. Functions can be evaluated in time logarithmic in the number of breakpoints with a binary search for the adjacent breakpoints for a given evaluation time and then interpolating linearly. As for the time domain, setting $T = \mathbb{Z}$ and using integers in some subsecond resolution[1] appears to be the most desirable approach from an implementation perspective. When the linear interpolation yields results outside of $\mathbb{Z}$, integer division provides a convenient, efficient and consistent way of rounding back into the domain. Calculating with integers avoids all of the implementation complications associated with representations of fractional numbers or, even worse, lossy representations of reals as floating point numbers. Therefore, using the TD-SPP$_{\mathbb{Z}}$ to model time-dependent routing problems appears promising.

Sadly, this is not always possible. Practically, function evaluation is not the only important operation. Another crucial operation is computing the function composition. This operation is necessary whenever we need to obtain the travel time function of a path. For many algorithms (for example [BGSV13] and Chapter 8), it is critical that composed functions

- stay in the same function class,
- remain as small as possible, i.e. have breakpoints at most linear in the number of breakpoints of the composed functions,
- and can be computed efficiently, i.e. in linear running time in the output size.

PPLF on $\mathbb{Q}$ or $\mathbb{R}$ support these requirements. On $\mathbb{Z}$, however, this is not the case. The integer rounding may introduce jumps in the composed function. See Figure 3.1 for an example. We are aware of no better way to represent these jumps than to insert all the breakpoints. This approach is infeasible because the input function sizes do not bound the number of these additional breakpoints. Therefore, whenever we use algorithms that require computing *exact* travel time function compositions, we must switch to the TD-SPP$_{\mathbb{Q}}$ or the TD-SPP$_{\mathbb{R}}$ problem.

[1] A resolution in seconds is typically slightly too coarse. Many short edges (for example, for intersections modelled in detail) would have travel times rounded down to zero.

3.2.1 Complexity

Choosing a suitable function class and time domain is not the only complication with the time-dependent shortest path problem. Additionally, the problem is only polynomially-time solvable (assuming P ≠ NP) when certain constraints are placed on the travel time functions. Most importantly, all travel time functions must conform to the *first-in first-out* (FIFO) property:

Definition 3.3 (First-In First-Out Property)**.** *A travel time function f adheres to the* first-in first-out *property when the following holds:*

$$\forall \tau \in T, \varepsilon > 0 : f(\tau) \leq \varepsilon + f(\tau + \varepsilon)$$

Assuming continuous functions, time intervals where the FIFO property is violated have slopes less than -1 in travel time representation and slopes less than zero in arrival time representation. Informally, the FIFO property holds when it is impossible to arrive earlier by departing later. For networks where this property holds, the earliest arrival problem can be solved efficiently with a variation of Dijkstra's algorithm [Dre69].

The FIFO property might appear quite natural for travel times on road networks. However, there are exceptions. An example in road networks might be an important tunnel which can only be traversed during certain times. When arriving too early while the tunnel has not yet opened, one may have to drive a longer detour. Of course, it may be possible to avoid such a detour by waiting at the tunnel entry. Likewise, non-FIFO functions can be transformed into FIFO functions when modelling waiting at the tail vertex as part of the travel time function. However, allowing waiting at arbitrary locations and times may not always be a realistic modelling assumption. Therefore, investigating the complexity of the problem in non-FIFO settings is also relevant. The complexity of the shortest path problem in non-FIFO time-dependent networks has been studied before, most prominently by Orda and Rom [OR90, OR91]. In [OR90], they present several time-dependent shortest path algorithms and study different waiting policies. They prove that under certain assumptions allowing waiting at the source vertex is sufficient to find paths with the same arrival as the fastest path with waiting allowed everywhere. Also, they present examples where the earliest arrival path has an infinite number of edges. See Figure 3.2 for an example. Therefore, searching for a shortest *path* is no longer a well-defined problem.

Many recent works [DW09, NDSL12, FHS14, GGG15] cite [OR90] stating that the TD-SPP in non-FIFO time-dependent networks is NP-hard. However, the paper only states that the hardness can be shown (second paragraph of Section 3.2) but does not provide any evidence. Some works [Bat14] state the hardness referring to an unpublished manuscript by Orda and Rom [OR89]. We could not find this manuscript in any public source but could only obtain it through personal contact with the authors. In this manuscript, Orda and Rom proved the weak NP-hardness of the time-dependent earliest arrival problem with travel time functions defined on integers and forbidden waiting by reduction from FiniteFunctionGeneration. They also show that the problem can be solved in pseudo-polynomial time. However, since the proof is quite complex and was never published, we conducted our own study and reproduced, simplified and extended the complexity results. We present the proofs in Section 3.4.

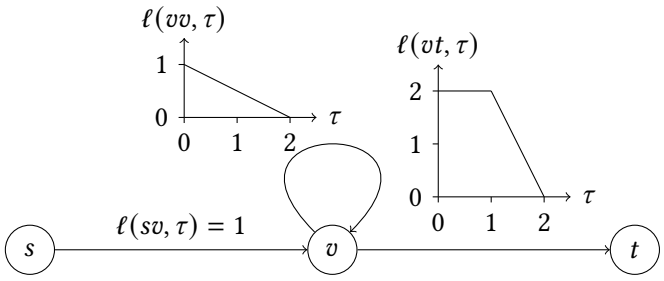


Figure 3.2: Example of a non-finite shortest path as previously observed in [OR90]. When departing at $\tau^{\text{dep}} = 0$, the travel time from s to t decreases with each round through the loop at v. Taking the loop k times leads to an arrival time of $2 + \frac{1}{2^k}$.

In the light of these complexity results, we follow previous practical works [DW09, NDSL12, BGSV13, BDPW16] and only allow FIFO travel time functions for time-dependent routing problems. At least for cars, there usually are sufficient possibilities for waiting and parking. Therefore, we believe that this is a justifiable modelling decision. However, the situation is significantly different for trucks. Therefore, in Chapter 11, we present a model specifically developed to enable efficient routing while handling non-FIFO travel times and waiting only at designated locations.

3.2.2 Shortest Travel Time Profiles

In time-dependent networks, asking for the shortest path when departing at a specific time is not the only important problem. User may also want to know *all* shortest travel times (and the respective paths) during a specific time window or the departure time for which the travel time will be minimal. We formalize this as the *travel time profile* problem:

Definition 3.4 (TravelTimeProfileProblem$_T$)**.** *Given is a weighted, directed graph $G = (\mathcal{V}, \mathcal{E})$ with a time-dependent edge length functions $\ell : \mathcal{E} \to (T \to T^{\geq 0})$, vertices s and t, and a time window $T' \subseteq T$. Obtain the time dependent function $f(\tau) = \text{dist}(s, t, \tau)$, $f : T' \to T^{\geq 0}$ of the shortest travel times of paths in G departing at s at any time $\tau \in T'$ and arriving at the earliest possible time at t.*

As shown by Foschini et al. [FHS14], with FIFO piecewise linear functions, the number of breakpoints in such a profile may be super-polynomial in the input size. Therefore, we will not be able to design guaranteed efficient algorithms for this problem. Nevertheless, we will investigate algorithms achieving practically good performance.

3.2.3 Accelerating Time-Dependent Route Planning

To improve running times in practice, we again consider two-phase variants of these time-dependent routing problems, denoted as the *two-phase time-dependent shortest path problem*. The graph G and the time-dependent travel time functions ℓ are given during preprocessing. In the online query phase, many source, target, departure (or time window) triples are queried. The respective time-dependent shortest path problems must be solved quickly using the auxiliary data from the preprocessing for acceleration. As the time-dependent functions are part of the preprocessing, this problem model is only reasonable for rarely changing traffic predictions such as commuter flows. However, considering *only* traffic predictions is also not satisfactory in practice. Therefore, we propose a problem model for combined traffic information in the next section.

3.3 Dynamic and Time-Dependent Route Planning

We consider a three-phase model to combine predicted and dynamic traffic information. As in the two-phase time-dependent shortest path problem, the graph $G = (\mathcal{V}, \mathcal{E})$ and a weight function ℓ_{pred} of time-dependent traffic predictions are given in the preprocessing phase. Predicted travel time functions are periodic piecewise linear functions represented by a sequence of breakpoints covering one day. A preprocessing algorithm may now precompute auxiliary data, which may take several hours. Preprocessing should still be fast enough to rerun it daily. In the *update* phase, a weight function ℓ_{live} of currently observed live travel times are given for the current moment τ^{now}. These live travel times are time-*in*dependent and can be represented by a single scalar value. Further, each edge e has a point in time $\tau^{\text{end}}(e)$ when we switch back to the predicted travel time. For edges without live traffic data, we consider $\tau^{\text{end}}(e) = \tau^{\text{now}}$. The update phase will be repeated frequently and should be as fast as possible. For the query phase, we define a combined travel time function:

$$\ell_{\text{comb}}(e,\tau) = \begin{cases} \ell_{\text{pred}}(e,\tau) & \text{if } \tau \geq \tau^{\text{end}}(e) \\ \min(\ell_{\text{live}}(e), \ell_{\text{pred}}(e,\tau^{\text{end}}(e)) + \tau^{\text{end}}(e) - \tau) & \text{else if } \ell_{\text{live}}(e) \geq \ell_{\text{pred}}(e,\tau^{\text{end}}(e)) \\ \max(\ell_{\text{live}}(e), \ell_{\text{pred}}(e,\tau^{\text{end}}(e)) - \tau^{\text{end}}(e) + \tau) & \text{else} \end{cases}$$

Now, many shortest path queries (s,t,τ^{dep}) where $\tau^{\text{dep}} \geq \tau^{\text{now}}$ should be answered as quickly as possible by obtaining a path $P = (s,\ldots,t)$ that minimizes $\ell_{\text{comb}}(P,\tau^{\text{dep}})$. We denote this as the *three-phase combined traffic shortest path problem*.

Note that with this model, we do not try to support fast updates to the travel time predictions. The dynamic real-time traffic information ℓ_{live} is handled separately from the traffic predictions ℓ_{pred}. Fast prediction updates would, of course, also be desirable. However, the traffic predictions are *periodic*, and traffic incidents are not expected to repeat after 24 hours. Therefore, separately handling real-time traffic makes more sense than frequently adjusting predictions. Further, this approach opens up possibilities for the design of efficient algorithms.

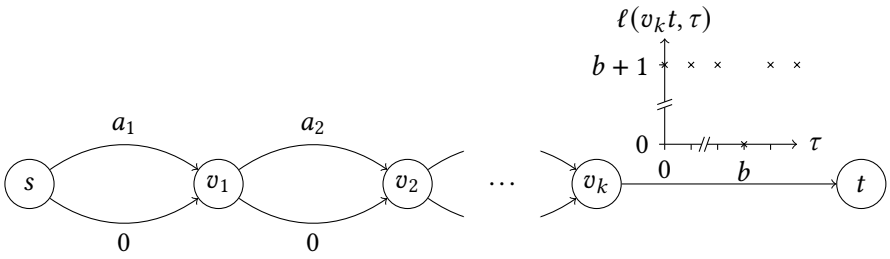


Figure 3.3: Transformed TDEA instance for SubsetSum instance $(\mathcal{A} = \{a_1, \ldots, a_k\}, b)$.

3.4 NP-Hardness of Shortest Path Problems in Networks with Non-FIFO Time-Dependent Travel Times

In this section, we present a simple proof of the weak NP-hardness of the earliest arrival problem on networks with time-dependent travel times defined on integers when waiting is forbidden. Additionally, this proof implies that the problem becomes strongly NP-hard when the functions are defined on rational numbers. Further, we show that the problem remains strongly NP-hard when we limit ourselves to a more practical travel time function model where functions are piecewise linear and given as a sequence of breakpoints with integer coordinates, but the computation is performed in the rational number domain. As an intermediate result we show that SubsetProduct on rational numbers is strongly NP-complete. Note that we published these complexity results in [Zei22b].

To argue about the complexity of the TD-SPP problem, we first introduce an appropriate decision problem, the time-dependent earliest arrival problem:

Definition 3.5 (TDEA_T)**.** *Given a graph $G = (\mathcal{V}, \mathcal{E})$ with non-negative travel times $\ell : \mathcal{E} \to (T \to T^{\geq 0})$, vertices s and t, a departure time $\tau^{\text{dep}} \in T$ and a maximum arrival time $\tau^{\text{max}} \in T$, is there a path $P = (s, \ldots, t)$ such that $\tau^{\text{dep}} + \ell(P, \tau^{\text{dep}}) \leq \tau^{\text{max}}$?*

On integers, this problem is weakly NP-hard.

Theorem 3.1. *$TDEA_{\mathbb{Z}}$ is weakly NP-hard.*

Proof. We prove NP-hardness by reduction from the weakly NP-complete problem SubsetSum [GJ79]. A SubsetSum instance consists of a multiset $\mathcal{A}$ of integers a_i and a target value b.[2] The goal is to decide whether there is a subset $\mathcal{A}' \subseteq \mathcal{A}$ such that $\sum_{a \in \mathcal{A}'} a = b$. We construct our $\text{TDEA}_{\mathbb{Z}}$ instance as follows: The instance has vertices $\mathcal{V} = \{v_i \mid a_i \in \mathcal{A}\} \cup \{s, t\}$. Let $v_0 = s$ and $k = |\mathcal{A}|$. For each SubsetSum element $a_i \in \mathcal{A}$, we create two edges from v_{i-1} to v_i. The

[2] Note that usually SubsetSum is defined in terms of a set and a function mapping each element to a not necessarily unique weight. To simplify notation, we instead use a multiset containing the weights directly as elements.

first one denoted as e'_i has constant travel time zero and the second one denoted as e_i constant travel time a_i. We insert a final edge $v_k t$ with the following non-FIFO travel time function:

$$\ell(v_k t, \tau) = \begin{cases} 0 & \text{if } \tau = b \\ b+1 & \text{else} \end{cases}$$

Setting $\tau^{\text{dep}} = 0$ and $\tau^{\max} = b$ completes the instance. See Figure 3.3 for an illustration. This transformation runs in $\mathcal{O}(|\mathcal{A}|)$. The transformed graph contains zero weights and multiedges, but neither is necessary. Zero weights can be eliminated by adding 1 to the travel time of each edge, shifting the instant where $\ell(v_{|\mathcal{A}|+1}t)$ has its minimum back by k time units and increasing the arrival time $\tau^{\max}$ by $k+1$. Multiedges can be avoided by inserting an additional vertex in the middle of each multiedge.

We now show the equivalence of the transformed instance. Assume that the SubsetSum instance admits a subset $\mathcal{A}'$ with $\sum_{a\in\mathcal{A}'} a = b$. Consider the path P from s to v_k which uses e_i if $a_i \in \mathcal{A}'$ and e'_i otherwise. Obviously, it has a constant total travel time of b. Thus, the edge $v_k t$ will be traversed at $\tau = b$. Since $\ell(v_k t, b) = 0$ the total travel time is b. It follows that the $\text{TDEA}_{\mathbb{Z}}$ instance admits a feasible path.

Conversely, assume that the $\text{TDEA}_{\mathbb{Z}}$ instance admits a path P of travel time b. Since the last edge has a travel time greater than b for all times except b, the travel time to v_k must have been b, too. Let $\mathcal{A}' = \{a_i \mid e_i \in P\}$. Since the travel time to v_k was b, $\sum_{a\in\mathcal{A}'} a = b$ has to hold and the SubsetSum instance also admits a feasible solution. This proves the weak NP-hardness. □

To prove that the problem is not strongly NP-hard, we now give a pseudo-polynomial algorithm. Algorithm 3.1 shows the procedure in pseudocode. The algorithm is a time-expanded variation of the shortest path algorithm of Bellman and Ford [Bel58]. Instead of a distance array, it maintains a 2-dimensional array indicating whether it is possible to reach a vertex at a given time. Instead of scanning edges n times, edges are relaxed for each instant between τ^{dep} and $\tau^{\max}$. This leads to a running time in $\mathcal{O}(\tau^{\max} \cdot m)$ which makes the algorithm pseudo-polynomial in the input size.

For simplicity, we describe the algorithm under the assumption that all travel times are strictly positive. Travel times of zero can be handled by repeating the loop in line 4 n times.

We prove correctness inductively. We show that in step τ of the loop in line 3 for all vertices v, all possible arrival times $\tau' \leq \tau$ have been correctly marked in R. Clearly, the base case holds, because in the absence of zero travel times, s is the only vertex reachable at τ^{dep}. Assume the induction hypothesis for $\tau - 1$. For any path $(s, \ldots, u, v)$ of travel time $\tau - \tau^{\text{dep}}$ the vertex u was reachable at a time $\tau' < \tau$. Thus, $\texttt{R}[u][\tau']$ was correctly set, the outgoing edges of u were relaxed and $\texttt{R}[v][\tau]$ will also be set which completes the inductive step. Repeating the loop in line 4 n times would additionally allow intermediate paths of travel time zero with up to n vertices, which clearly suffices to solve this case, too.

Wojtczak showed in [Woj18] that many common weakly NP-hard problems, including SubsetSum, become strongly NP-hard when defined on rational numbers. As our proof generalizes to rational numbers without modification, we get the following corollary:

Algorithm 3.1: Time-Expanded Bellman-Ford Algorithm.

Data: $\mathtt{R}[u][\tau]$: 2-dimensional array of booleans indicating whether it is possible to arrive at vertex u at time τ.
Data: $G = (\mathcal{V}, \mathcal{E})$: Directed graph with travel time functions ℓ.

```
1 Function TimeExpandedBellmanFord(s, t, τ^dep, τ^max):
2     R[s][τ^dep] ← true
3     for τ ∈ [τ^dep, τ^max] do
4         for uv ∈ E do
5             if R[u][τ] then
6                 if u = t then
7                     return true
8                 R[v][τ + ℓ(uv, τ)] ← true
9     return false
```

Corollary 3.2. *$TDEA_{\mathbb{Q}}$ is strongly NP-hard.*

In practice, however, we usually use a very limited subclass of rational functions: piecewise linear functions represented as a sequence of breakpoints. Further, as input data usually comes at a limited resolution, input breakpoint coordinates usually can be represented by integers. The result of a linear interpolation may, of course, still be a rational number. This lead us to the question if $\mathrm{TDEA}_{\mathbb{Q}}$ on piecewise linear functions with integer coordinates is weakly or strongly NP-hard. In the following we consider non-periodic piecewise linear functions. We define the travel times before the first and after the last breakpoints to equal the travel time of the first and last breakpoint, respectively. However, this does not make any difference in terms of complexity. The construction for the reduction only uses a limited time horizon.

With piecewise linear functions with integer breakpoints, we cannot easily construct constant travel time functions with arbitrary rational travel times. Thus, generalizing the reduction from SubsetSum is difficult. However, we can encode rational numbers in the *slope* of arrival time function and exploit that when composing linear functions, the result function's slope is equal to the product of the slopes of the composed functions. In the following, we show the strong NP-hardness with a two-step reduction from SubsetProduct on rational numbers as an intermediate problem. On integers, SubsetProduct is weakly NP-complete [GJ79, Joh81].[3]

[3] Garey and Johnson [GJ79] state that SubsetProduct on $\mathbb{Z}^{\geq 0}$ is strongly NP-complete by reduction from ExactCoverBy3Sets with reference to private communication with A. C. Yao in 1978. This is surprising since there is no obvious reason why the pseudo-polynomial dynamic programming algorithm for SubsetSum should not be applicable to SubsetProduct. This suggests that the problem is only weakly NP-complete and that the reduction proved only weak hardness. While we were unable to obtain the original proof, other realizations of the reduction can be found online [Jon14]. This reduction uses a number exponential in the input size for the target product b. Clearly, this admits a pseudo-polynomial algorithm. As it turns out, on integers, the problem is only weakly NP-hard. Johnson's NP-completeness column later contains a correction [Joh81].

However, as we now show, when allowing rational numbers, the problem becomes strongly NP-complete. SubsetProduct is defined as follows:

Definition 3.6 (SubsetProduct)**.** *Given a finite multiset of positive numbers $\mathcal{A}$ and a positive number b, is there a subset $\mathcal{A}' \subseteq \mathcal{A}$ such that $\prod_{a \in \mathcal{A}'} a = b$?*

Theorem 3.3. *SubsetProduct on rational numbers is strongly NP-complete.*

Proof. A guessed solution for SubsetProduct can be verified in polynomial time. In the worst case, all numerators and denominators in the solution would have to be multiplied with each other. Thus, SubsetProduct is in NP even with rational numbers.

For hardness, we reduce from the ExactCoverBy3Sets problem. It is strongly NP-complete due to [GJ79] and defined as follows:

Definition 3.7 (ExactCoverBy3Sets (X3C))**.** *Given a set $\mathcal{X} = \{x_1, \ldots, x_k\}$ with k being a multiple of 3, and a collection $\mathcal{C}$ of 3-element subsets of $\mathcal{X}$, does $\mathcal{C}$ contain an exact cover $\mathcal{C}'$ for $\mathcal{X}$, where $\mathcal{C}' \subseteq \mathcal{C}$ and every element in $\mathcal{X}$ occurs in exactly one element of $\mathcal{C}'$?*

For the transformation, we first generate the first $2k+1$ primes. We refer to these primes in two groups of each k primes $\{p_1, \ldots, p_k\}$ and $\{p'_1, \ldots, p'_k\}$ and one final prime number p^*. Generating these primes takes polynomial running time [Woj18]. We construct the set $\mathcal{A} = \mathcal{A}_\mathcal{X} \cup \mathcal{A}_\mathcal{C}$ from one number for each element in $\mathcal{X}$ and one number for each 3-element set in $\mathcal{C}$:

$$\mathcal{A}_\mathcal{X} = \left\{ \frac{p_i p'_{i+1}}{p'_i} \,\middle|\, x_i \in \mathcal{X} \setminus x_k \right\} \cup \left\{ p^* \cdot \frac{p_k p'_1}{p'_k} \right\}$$

$$\mathcal{A}_\mathcal{C} = \left\{ \frac{1}{p_i p_j p_k} \,\middle|\, \{c_i, c_j, c_k\} \in \mathcal{C} \right\}$$

Setting $b = p^*$ completes the instance.

Assume that the X3C instance admits a set $\mathcal{C}' \subseteq \mathcal{C}$ which covers $\mathcal{X}$. Consider the set $\mathcal{A}' = \mathcal{A}_\mathcal{X} \cup \mathcal{A}_{\mathcal{C}'}$ containing all elements from $\mathcal{A}_\mathcal{X}$ and all elements from $\mathcal{A}_\mathcal{C}$ corresponding to elements form $\mathcal{C}'$. By construction, the product over all numbers in $\mathcal{A}'$ equals p^*:

$$\prod_{a \in \mathcal{A}'} a = \prod_{a \in \mathcal{A}_\mathcal{X}} a \cdot \prod_{a \in \mathcal{A}_{\mathcal{C}'}} a = p^* \prod_{x_i \in \mathcal{X}} \frac{p_i p'_i}{p'_i} \cdot \prod_{x_i \in \mathcal{X}} \frac{1}{p_i} = p^*$$

It follows that the SubsetProduct admits a feasible subset, too.

Conversely, assume $\mathcal{S}'$ is a fulfilling subset for the SubsetProduct instance. First, since a_k is the only element which contains p^* as a factor, it must be contained in $\mathcal{S}'$. Secondly, note that if $\mathcal{S}'$ contains any element from $\mathcal{A}_\mathcal{X}$, it has to contain *all* elements in $\mathcal{A}_\mathcal{X}$ to cancel out the p'_i prime factors. The product of all elements in $\mathcal{A}_\mathcal{X}$ is equal to $p^* \cdot \prod_{x_i \in \mathcal{X}} p_i$. For the total product of all elements in $\mathcal{S}'$ to be equal to p^*, the product of the remaining elements in $\mathcal{S}'$ has to equal exactly $\prod_{x_i \in \mathcal{X}} p_i^{-1}$. This can only be the case when the elements of $\mathcal{C}$ corresponding to these remaining elements in $\mathcal{S}'$ form an exact cover of $\mathcal{X}$. Thus, the X3C instance admits a feasible cover. □

With the strong hardness of SubsetProduct on rational numbers, we can now prove strong hardness for the probably most practical model.

Theorem 3.4. *$TDEA_{\mathbb{Q}}$ with piecewise linear functions represented by a sequence of breakpoints is strongly NP-hard even when all input numbers are integers.*

Proof. Let $\mathcal{A} = \{\frac{p_1}{q_1}, \dots, \frac{p_k}{q_k}\}, b = \frac{p^*}{q^*}$ be our SubsetProduct instance. First, we sort $\mathcal{A}$ in ascending order, i.e. $\frac{p_i}{q_i} \leq \frac{p_{i+1}}{q_{i+1}}$. We then construct the $\mathrm{TDEA}_{\mathbb{Q}}$ instance as follows: Similarly to the hardness proof for $\mathrm{TDEA}_{\mathbb{Z}}$, we build a graph with $k+2$ vertices and two parallel edges e_i and e_i' between v_{i-1} and v_i for $1 \leq i \leq k$ and a final non-FIFO edge between v_k and t. All edges e_i' get constant travel time zero. The edges e_i get a time-dependent travel time function which has slope $\frac{p_i}{q_i}$ in the *arrival time representation*. The final edge has a travel time function with breakpoints $(0, p^* + 1)$ and $(p^*, 0)$. The topology is the same as in the previous proof as illustrated in Figure 3.3. We set $\tau^{\text{dep}} = q^*$ and $\tau^{\max} = p^*$. We pick the breakpoints for the e_i functions in such a way that the arrival time function has the correct slope at least for all times in $[0, \max(p^*, q^*)]$. Thus, with $s_i = \lceil \frac{\max(p^*, q^*)}{q_i} \rceil$ the edge e_i will have the arrival time function breakpoints $(0, 0)$ and $(s_i q_i, s_i p_i)$. This transformation has polynomial running time and also all numbers are polynomially bounded in the size of the SubsetProduct instance.

Some of these arrival time functions have slope < 1 and thus for $\tau \in [0, \tau^{\max}]$ we get $\hat{\ell}(\tau) < \tau$. This means the corresponding travel time functions have negative travel times. We can obtain an equivalent instance without negative weights as follows: Let e_i be an edge with a breakpoint (x, y) in the travel time function where $y < 0$. We increase the travel times of both e_i and e_i' by the constant $c = -y$. Additionally, the departure times coordinates of breakpoints in every following travel time function must be increased by c, i.e. for (x, y) a breakpoint of a travel time function of an edge with head v_j where $j > i$, this breakpoint will be set to $(x + c, y)$. Also, $\tau^{\max}$ must be increased by c. Even though negative travel times are not necessary, we still use them during the rest of the proof because it simplifies the calculations. This also applies to travel times of zero which can be avoided similarly.

Suppose the SubsetProduct instance admits a subset $\mathcal{A}'$ such that the product of all elements in $\mathcal{A}'$ is b. We consider the st-path which uses e_i when $\frac{p_i}{q_i} \in \mathcal{A}'$ and e_i' otherwise. When visiting vertex v_{i-1} at instant τ_{i-1} and traversing the edge e_i with arrival time breakpoints $(0, 0)$ and $(s_i q_i, s_i p_i)$ with $\tau_{i-1} \leq s_i q_i$, the arrival time at v_i can be computed by linear interpolation: $\tau_i = \tau_{i-1} \cdot \frac{p_i}{q_i}$. Assuming $\tau_{i-1} < s_i q_i$ for $1 \leq i \leq k$, the arrival time τ_k at the final vertex v_k before t can be computed as $\tau^{\text{dep}} \cdot \prod_{a_i \in \mathcal{A}'} \frac{p_i}{q_i} = q^* \cdot \frac{p^*}{q^*} = p^*$. The final edge has a travel time of 0 at p^* which leads to an arrival of p^* at t which shows, that the $\mathrm{TDEA}_{\mathbb{Q}}$ instance admits a feasible path given that our assumption on the τ_i holds. This assumption is valid because the elements of $\mathcal{A}$ and thus the slopes of the edges are ordered ascendingly. For all $\frac{p_i}{q_i} \leq 1$, $\tau_i \leq \tau_{i-1}$, i.e. the travel time is negative, and we arrive earlier than we started, so $\tau_i < q^*$ holds. Once edges with $\frac{p_i}{q_i} > 1$ are reached, the τ_i grow monotonically. But since the final result p^* is within the desired range, all intermediate τ_i have to be, too. Thus, the assumption holds for all τ_i.

Conversely, assume the $\mathrm{TDEA}_{\mathbb{Q}}$ instance admits a path P with the desired arrival time. This

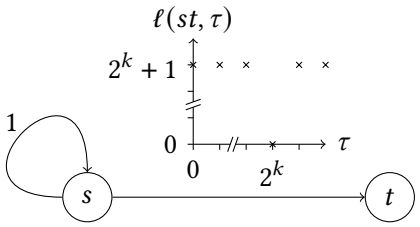


Figure 3.4: Example graph with a shortest path with an exponential number of hops.

is only possible when $\tau_k = p^*$ because before p^*, the arrival time function has a strictly negative slope and after p^* a strictly positive slope. With the same argument as in the previous paragraph, we get that all $\tau_i \leq \max(p^*, q^*)$. Hence, the following holds: $\tau^{\text{dep}} \cdot \prod_{e_i \in P} \frac{s_i p_i}{s_i q_i} = \tau^{\max} = p^*$ which is equivalent to $\prod_{e_i \in P} \frac{p_i}{q_i} = \frac{p^*}{q^*}$. Thus, the set $\mathcal{A}' = \left\{ \frac{p_i}{q_i} \mid e_i \in P \right\}$ is a fulfilling subset for the SubsetProduct instance. □

So far, we only discussed NP-hardness but not inclusion in NP. On first glance, it appears likely that the TDEA problem lies in NP because verifying path lengths takes running time linear time in the path length. However, this is not sufficient because solutions might have superpolynomial length in the input size. Consider a graph with two vertices s and t, one loop edge at s with constant travel time 1 and one st edge with travel time 0 at instant 2^k and travel time 2^{k+1} during the rest of the time as depicted in Figure 3.4. For an instance with $\tau^{\text{dep}} = 0$ and $\tau^{\max} = 2^k$, the solution is a path which uses the loop at s 2^k times. A naive encoding of this solution would already be exponentially bigger than the (binary encoded) input. Even if it would be possible to show that a compact encoding of solutions is always possible, one would still need to find a way to evaluate the travel time of this path in polynomial running time in the input size. Whether this is possible likely depends on the involved function classes. Therefore, the question remains open. Nevertheless, we only used acyclic graphs and simple paths in our reductions. Thus, the problem remains hard even when considering problem variants restricted to acyclic graphs or when only allowing simple paths. Since paths with an exponential number of hops are not possible in these variants, guessed solutions could be verified in polynomial time assuming polynomial time function evaluation. Therefore, we can at least conclude that such restricted TDEA variants would be NP-complete.

4 Route Planning Data

A crucial ingredient to the algorithm engineering methodology is to evaluate the designed algorithms with realistic benchmark instances. In this chapter, we present the primary data sets used throughout this work. In Section 4.1, we discuss our data sources and methodology to turn the raw map and traffic data into problem instances for the models presented in the previous chapter. Our benchmark instances and a discussion of their relevant properties are presented in Section 4.2.

4.1 Data Sources and Instance Extraction

4.1.1 9th DIMACS Implementation Challenge

We use the Europe instance from the 9th DIMACS Implementation Challenge [DGJ09] from 2006. This instance is probably the most used benchmark instance for routing algorithms. Most speedup techniques have been evaluated on it [Bas+16]. As it was the first production-grade continental-sized routing graph available to a broad range of researchers, it was instrumental in the success of route planning algorithm research. The instance was provided by PTV and is based on TomTom data. It is not publicly available but can be obtained free of charge for research purposes.[1]

The instance is provided as a weighted directed graph. Both travel time and distance weights are provided. Further, geographical coordinates of the vertices are provided. Therefore, we can use this instance without adjustments for our routing problems.

[1] `https://i11www.iti.kit.edu/resources/roadgraphs.php`, `https://i11www.iti.kit.edu/information/roadgraphs`

Sadly, we are not aware of any real-world traffic data for this instance. Synthetic traffic data has been generated [NDSL12] and used in previous works to evaluate time-dependent speedup techniques [NDSL12, BGSV13]. These time-dependent travel times are still available on the technical infrastructure of our research group. However, the synthetic functions deviate significantly from the properties we observe in travel time functions obtained from real-world traffic predictions from other data sources. They resemble traffic jams caused by accidents rather than recurring traffic patterns. Nevertheless, we include the instances in our evaluation for reproducibility and comparability.

4.1.2 PTV

PTV[2] kindly provided us with additional newer road networks and real-world traffic data sets:

- The first one is from 2006 and includes real-world traffic predictions for the subgraph of Germany of the DIMACS Europe instance. There are different predictions for Monday, Friday, Saturday, and Sunday, and one prediction set for Tuesday till Thursday. The traffic predictions are given as speed predictions for 15 minute intervals. From these speed predictions, travel time functions were derived for previous works. The results are still available on the infrastructure of our group, and we reuse those derived instances to ensure comparability. From internal documentation, it appears that the methodology used to derive those travel time functions was the same as we describe in Section 4.1.5.
- The second one is from 2017 and includes a graph of the road network of Europe and traffic predictions for a typical Tuesday. Further, there are subinstances of Germany and Luxembourg. This data set contains significantly more traffic prediction information than the data from 2006. The traffic predictions are given as travel time functions and are directly usable for our problem instances.
- The third set is from 2020 and covers everything that the 2017 data set did and more. These traffic predictions cover an even larger share of the network. Further, a real-time traffic incident snapshot is included. The traffic incidents are provided as updated speeds for some edges of the graph. Further, each traffic incident is associated with an expected remaining duration of the incident. The traffic predictions are given as travel time functions and directly usable as problem instances.

4.1.3 TomTom

We also have map data provided directly by TomTom[3] in 2012. The data set covers the north-eastern part of Germany, but in the literature, it is referred to by the largest city, Berlin. It includes speed predictions in five-minute intervals for every weekday. Like the PTV data,

[2] https://ptvgroup.com
[3] https://www.tomtom.com

time-dependent graphs were already generated for previous works with the methodology from Section 4.1.5. We reuse these to ensure the comparability of our results.

4.1.4 OpenStreetMap

OpenStreetMap[4] (OSM) is a community-maintained open-source map. It is possible to derive routing graphs from OSM data. Since OSM can be used freely and is publicly available to everyone, evaluating algorithms on OSM graphs helps other researchers reproduce these results. Therefore, we also included an OSM-based instance in our evaluation. We use a snapshot of the map of Germany from the beginning of 2020 and derive a routing graph using the import from RoutingKit.[5]

Typically, OSM data contains much more information than is necessary for routing. Naively constructed OSM graphs often contain many vertices of degree two. Their purpose in OSM is to model the course of the street, but for the routing, they are irrelevant. Therefore, RoutingKit will remove these vertices and combine the adjacent edges as long as no relevant attributes (for example, a speed limit) change. We extend RoutingKit's import to allow us to specify vertices that should be kept as we need them to match traffic data.

4.1.5 Mapbox

In 2019, Mapbox[6] kindly provided us with real-world traffic data for Germany matched to OSM. The data includes two real-time traffic snapshots. These consist of currently observed speeds for specific OSM edges. Some of these edges' endpoints are vertices that the RoutingKit OSM import would usually remove. To avoid this, we specify that these vertices must be kept during the import. In contrast to the PTV traffic incident data, no estimated validity is provided. We exclude live speeds which are faster than the free-flow speed computed by RoutingKit. Even though some traffic participants might drive faster than predicted (and allowed), we believe this should not be reflected in the routing.

The data also contains traffic predictions for some edges. These come as speed predictions for every five-minute interval for an entire week. We build a time-dependent routing graph from the predictions for Tuesdays. During preliminary experiments, we noticed that the predicted speeds sometimes have heavy fluctuations. This is likely because the speeds reflect aggregated observations instead of proper traffic predictions. Therefore, we apply some smoothing to the data before extracting travel time functions. First, we ignore any speed predictions between 23:00 and 5:00. Second, we apply a rolling window of half an hour and set each bucket to the average values in the window. Finally, we disregard any predicted speeds faster than the free-flow speed computed by RoutingKit.

[4] `https://openstreetmap.org`
[5] `https://github.com/RoutingKit/RoutingKit`
[6] `https://mapbox.com`

Obtaining Travel Time Functions from Speed Predictions

To transform speed predictions to travel time functions, we choose a direct and naive interpretation of the speed predictions: Every car will drive at all times, precisely at the predicted speed. When a car is on an edge while the time of a speed prediction ends and the next interval start, the car's speed will, at that moment, jump instantaneously to the new speed. While physically impossible, this approach yields useful, piecewise linear travel time functions. The functions are guaranteed to fulfil the FIFO property as no car can overtake another.

4.1.6 BMW and Here

Finally, we also have commercial map data for western Europe from the late 2016 release of Here.[7] The data set was kindly provided by BMW[8] for research purposes. It is neither freely available nor used in any related work we know. We include it nonetheless because we have a week of GPS traces for the Munich region matched to the Here map. Each tracepoint contains a timestamp, a matched edge ID and a measured current speed of the car. From these traces, we built our own traffic predictions and constructed a time-dependent routing graph for Munich.

We use the following methodology: First, we only use the traces collected from Tuesday to Thursday. Second, we ignore any traces with speeds faster than 1.5 times the edges' free-flow speed. Third, we discard the weekday information and group the traces into 15 minute buckets. Now for each edge and bucket where we have three or more data points, we compute a speed prediction as a weighted average. For this, we sort the traces by time. Each observed speed is weighted with the time between the data points before and after. This is because slower driving cars stay longer on a road segment and thus produce more data points. For example, a car waiting at a traffic light will produce many more tracepoints at a slow speed than a car driving over the street segment at full speed. If we were to give each observation the same weight, we would underestimate speeds. For all other buckets and edges with too few data points, we assume the free-flow speed of the map data. To construct travel time functions from the speed prediction we employ the method described in Section 4.1.5.

This traffic prediction methodology is, of course, rather simplistic. Much more sophisticated approaches have been proposed [ALPR12, CZL12, JK13, PZWS13, Pan+13]. Realistic traffic predictions can be obtained by using proper traffic flow models where the model parameters are derived from the GPS traces [CZL12]. Unfortunately, calibrating these models is often non-trivial. Also, for proper predictions, more GPS trace data would be necessary. We also performed a more thorough investigation of the performance of several general-purpose statistical learning methods [JWHT13] to obtain better speed predictions. Sadly, we cannot share the results in detail due to non-disclosure agreements. We still want to point out that the difficulty in such studies is that the quality of traffic predictions is tough to quantify. It is possible to split the available data into training and evaluation sets and quantify the quality by checking the error

[7] `https://www.here.com/`
[8] `https://www.bmwgroup.com`

Table 4.1: Main static road networks used for the evaluation our algorithms.

	Source	Vertices $[\cdot 10^3]$	Edges $[\cdot 10^3]$	Size [MB]
Germany	OSM	16 169.0	35 442.2	346.2
Europe	DIMACS	18 010.2	42 188.7	409.6

of the learned predictions against the evaluation set. However, this only evaluates how the learning methodology performed and not if the resulting predictions allow good routing. From a navigation application perspective, routes must satisfy the users and using traffic predictions is only a means to that end. Nevertheless, the focus of this work is algorithms and their performance. Our crude predictions only make the instance harder for algorithms exploiting the properties of regular traffic flows. Therefore, the derived instance serves as a stress test for our algorithms. Our main conclusions will be drawn from other instances.

4.2 Benchmark Instances

4.2.1 Static Road Networks

Table 4.1 depicts our main benchmark instances for time-independent routing problems. On the one hand, since OSM is open-source, the Germany network should allow the independent reproduction of our results. On the other hand, the Europe instance allows for comparing our results to the results of previous works.

Even though the Europe instance covers a significantly larger area, it is only slightly larger than the Germany instance. This is because the OSM instance is newer and modelled in greater detail. The degree of modelling detail is also reflected in the average degrees: on Europe, every vertex has, on average, 2.34 outgoing edges, while on Germany, there are only 2.19 outgoing edges per vertex.

Beside these main instances, we use some additional benchmark graphs in Chapter 7, 10 and 11. We do not discuss these here as their relevance is limited to the scope of these chapters.

4.2.2 Networks with Traffic Predictions

In Table 4.2, we give an overview of our time-dependent benchmark instances and their relevant properties. The performance of algorithms for the classical shortest path problem with scalar weights primarily depends on the size of the network. However, for time-dependent routing problems, the amount and complexity of time-dependent information also significantly impact the performance. To measure this, we report the fraction of edges with a non-constant travel time and the average number of breakpoints. We also report the *relative total delay* as a measure

Table 4.2: Time-dependent road networks used throughout this work. The TD column indicates the share of edges for which the travel time functions are non-constant. The Avg. $|\ell(e)|$ columns show the average number of breakpoints among all/only the non-constant travel time functions, respectively. Similarly, Rel. Del. indicates the relative total delay among the respective edges. The size column shows the space requirement of a compact representation in main memory.

		Vertices	Edges	TD	Avg. $\lvert\ell(e)\rvert$		Rel. Del. [%]		Size
		$[\cdot 10^3]$	$[\cdot 10^3]$	[%]	all	TD	all	TD	[GB]
Mun	Tue.	22.5	53.2	32.4	12.1	35.4	41.0	279.0	0.0
Ber	Mon.	443.2	988.5	27.4	21.1	74.6	3.1	17.7	0.2
	Tue.	443.2	988.5	27.4	21.3	75.0	3.1	17.6	0.2
	Wed.	443.2	988.5	27.5	21.3	74.9	3.1	17.5	0.2
	Thu.	443.2	988.5	27.6	21.5	75.2	3.2	17.7	0.2
	Fri.	443.2	988.5	27.2	20.7	73.4	3.1	17.5	0.2
	Sat.	443.2	988.5	20.2	14.7	69.1	2.1	14.8	0.1
	Sun.	443.2	988.5	19.9	14.1	67.2	2.0	14.6	0.1
Ger06	Mon.	4 688.2	10 795.8	7.0	2.3	20.1	1.7	33.1	0.3
	midw.	4 688.2	10 795.8	7.2	2.3	19.5	1.7	33.1	0.3
	Fri.	4 688.2	10 795.8	6.4	2.1	18.9	1.5	32.0	0.3
	Sat.	4 688.2	10 795.8	3.9	1.6	15.8	0.8	28.5	0.2
	Sun.	4 688.2	10 795.8	2.5	1.3	15.0	0.4	26.2	0.2
SynEur	Low	18 010.2	42 188.7	0.1	1.0	13.2	0.3	125.2	0.8
	Med.	18 010.2	42 188.7	1.0	1.1	13.2	0.8	124.9	0.8
	High	18 010.2	42 188.7	6.2	1.8	13.2	4.6	124.8	1.0
Ger17	Tue.	7 247.6	15 752.1	29.2	10.0	31.6	3.5	20.8	1.3
Eur17	Tue.	25 758.0	55 503.8	27.2	8.8	29.5	2.7	19.0	4.2
Ger19	Tue.	16 169.0	35 442.2	38.0	47.4	123.1	2.3	78.8	13.7
Eur20	Tue.	28 510.0	60 898.8	76.3	17.4	22.5	21.0	34.9	8.7

for the degree of time-dependency of the predictions. It is defined as:

$$\frac{\sum_{e \in \mathcal{E}} \overline{\ell}(e) - \underline{\ell}(e)}{\sum_{e \in \mathcal{E}} \underline{\ell}(e)}$$

The smaller the relative delay, the greater the effectiveness of simple bound-based pruning schemas. Variants of this measure have been used in previous works. In [Del11], Delling reported the average relative delay of time-dependent earliest arrival queries over the result of a time-independent query with lower bound travel times. Batz [BGSV13] reported the *average*

of the relative delays $\frac{\overline{\ell}(e)-\underline{\ell}(e)}{\underline{\ell}(e)}$ over all edges $e \in \mathcal{E}$. We instead use the *total* delay because averages of ratios have hard to interpret semantics [HB15]. For example, a short edge with a large relative delay could have a much bigger influence on the average relative delay than on the shortest path structure.

The Munich instance is, in terms of graph size, our smallest graph by an order of magnitude. Still, a third of the edges have time-dependent travel times, and time-dependent functions have more than 35 breakpoints on average, more than all other instances except Berlin and OSM Germany. Further, the relative delay shows that the time-dependent travel times fluctuate heavily. Considering only time-dependent edges, a path could become up to 3.8 times slower than in the best case, depending on the departure time. This extreme fluctuation appears unrealistic and is likely an artefact of our simplistic speed prediction methodology amplified because the graph is small. Therefore, we do not expect this instance to yield perfectly representative performance results. We still include it to test the robustness of speedup techniques against extreme travel time functions.

The network of Berlin is more than an order of magnitude larger. Compared to Munich, the share of time-dependent functions is slightly smaller, and the complexity of the functions is even significantly greater. However, the relative delays show that the fluctuation of the time-dependent information is an order of magnitude smaller. Thus, even though the time-dependency is modelled in greater detail, it will likely have less impact on the shortest path structure of the network. The differences between the weekdays are comparatively small. On weekend days, fewer edges have time-dependent travel times. The other measurements decrease accordingly.

The network of Ger06, the benchmark instance used in most related work, is an order of magnitude larger than the Berlin network. However, the complexity of the time-dependent information is minimal. Only the SynEur instance has less complex functions. Further, only up to 7.2% of the edges have time-dependent travel times. While the total delay among these is slightly more significant than on Berlin, the overall influence is smaller as there are so few time-dependent edges. This makes this network one of the easier instances. Similar to Berlin, the weekend days appear even easier.

DIMACS Europe, with synthetic traffic, exhibits unique properties that distinguish it from real-world instances. The share of time-dependent travel time functions is tiny. Only the High instance starts to get into the region of Ger06, the real-world instance with the smallest share of time-dependent edges. Further, the time-dependent functions have very few breakpoints. Most importantly, the relative delays behave somewhat peculiar. The edges with time-dependent travel times have substantial relative delays, i.e. they may become more than two times slower to traverse. In comparison, the total delay on *all* edges is vanishingly small. We conclude that these travel time functions likely do not model recurring traffic flows. Instead, they model significant delays on a small number of edges, i.e. traffic jams. As we model jams as *dynamic* traffic, we do not expect the most conclusive results from this instance. We still include it for comparability and completeness.

Table 4.3: Real-time traffic snapshots.

Instance	Source	Date & Time	Entries [$\cdot 10^3$]
OSM Germany Lite	Mapbox	Tue. 2019/07/16 10:21	185.3
OSM Germany Heavy	Mapbox	Fri. 2019/08/02 15:41	320.7
Eur20	PTV	Wed. 2020/10/28 07:47	214.6

The newer graphs are not only larger in terms of the number of vertices and edges but also have significantly more non-constant travel time functions. Ger17 has four times as many time-dependent edges as Ger06 and about 1.5 times as many breakpoints per time-dependent function. Even though the relative delay among non-constant functions is not as high as for Ger06, the relative delay among all edges is twice as high. Eur17 exhibits similar characteristics but has 3.5 times as many vertices and edges.

Ger19 is the OSM Germany network with the Mapbox traffic predictions. It has a large network, a significant share of time-dependent edges (around one third), the time-dependent functions are quite complex and have the largest number of breakpoints among all instances, and it also has a significant relative delay on the time-dependent edges. This combination of properties makes it our most challenging instance.

Our newest instance is Eur20. It features the largest share of time-dependent edges: three quarters of edges having a non-constant travel time. With around 35%, the delay among non-constant functions is the greatest among all instances. Further, Eur20 is the only instance where the total relative delay on all edges is in the same ballpark as the delay for only time-dependent edges: With 21%, it is more than an order of magnitude higher than on Ger06.

We perform experiments on the complete set of instances. Usually, however, the results on a small subset of the instances suffice to support our claims. Thus, we often only discuss the results on a limited subset of the graphs and report the complete results in the appendix.

4.2.3 Real-Time Traffic Snapshots

Table 4.3 lists our real-world live traffic snapshots. We have two snapshots from Mapbox for our Germany instance. The first is from a Tuesday morning and contains only lite traffic. The second one is from a Friday afternoon and contains significantly heavier traffic. Finally, for the Eur20 instance, we have a traffic snapshot from PTV. It features regular traffic. Further, estimated validity durations for the live traffic data are included.

Part II

Speedup Techniques

5 Fundamental Algorithms and Data Structures

In this chapter, we discuss fundamental algorithms and data structures on which we build. We briefly reiterate how these algorithms work as far as it is necessary to understand the following chapters. Further, we introduce the associated terms and notation used throughout this work.

5.1 Representing Graphs

Unless mentioned otherwise, we represent graphs as *adjacency arrays*. To represent a graph topology $G = (\mathcal{V}, \mathcal{E})$, we have an array `firstEdge` of length $n + 1$ and an array `head` of length m. We refer to the indices of vertices and edges in these arrays as their *IDs*. The outgoing edges uv of vertex u have the ID range $[\texttt{firstEdge}[u], \texttt{firstEdge}[u + 1])$; the head vertex of each edge is stored at the corresponding position in `head`. Additional data for vertices and edges such as edge weights can be stored and accessed in additional arrays of the respective length.

5.2 Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] computes $\text{dist}(s, t, \tau^{\text{dep}})$ by exploring vertices in increasing order of distance from s until t is reached. The procedure is depicted in Algorithm 5.1. We describe the generalized variant of Dijkstra's algorithm [Dre69] solving the TD-SPP. The time-independent variant is just the special case that all edge lengths are constant.

The distances from s to each vertex u are tracked in an array $\texttt{D}[u]$, initially set to ∞ for all vertices. A priority queue `Q` of vertices ordered by their distance from s is maintained. We denote by q the minimum key in `Q`, i.e. the tentative distance of the closest remaining vertex. The priority queue is initialized with s and $\texttt{D}[s]$ set to τ^{dep}. In each iteration, the next closest vertex u

Algorithm 5.1: Dijkstra's algorithm.

Data: $\mathtt{D}[v]$: tentative distance from s to vertex $v \in \mathcal{V}$.
Data: Minimum priority queue $\mathtt{Q}$, ordered by tentative distances.
Function `Dijkstra(`s`)`**:**
  $\mathtt{D}[v] \leftarrow +\infty$ for all $v \in \mathcal{V}$
  $\mathtt{D}[s] \leftarrow \tau^{\text{dep}}$
  Make $\mathtt{Q}$ only contain s
  while $\mathtt{Q}$ not empty **do**
    $u \leftarrow$ pop minimum element from $\mathtt{Q}$
    for all edges $uv \in \mathcal{E}$ **do**
      if $\mathtt{D}[v] > \mathtt{D}[u] + \ell(uv, \mathtt{D}[u])$ **then**
        $\mathtt{D}[v] \leftarrow \mathtt{D}[u] + \ell(uv, \mathtt{D}[u])$
        Add v or decrease v's key in $\mathtt{Q}$ to $\mathtt{D}[v]$

is extracted from the queue and *settled*. Its distance $\mathtt{D}[u]$ from s is now final and optimal. Then, outgoing edges uv are *relaxed*, i.e. the algorithm checks if $\mathtt{D}[u] + \ell(uv, \mathtt{D}[u])$ improves $\mathtt{D}[v]$. If so, the position of v in $\mathtt{Q}$ is adjusted accordingly. Once t has been settled, the final distance is known, and the search terminates. The shortest path can be reconstructed by maintaining an array of parent pointers $\mathtt{P}[v]$ where for each vertex v, the predecessor on the shortest path is stored. We denote visited vertices as the *search space* of a query.

The property that a vertices distance is final once it was popped from the queue is denoted as *label-setting*. However, we also use variations of Dijkstra's algorithm where distance labels may be modified after the vertex was popped from the queue. These are denoted as *label-correcting*.

Dijkstra's algorithm with constant edge lengths can also be run from the target t on the reversed graph $\overleftarrow{G}$. We call this a *backward search*. Running two Dijkstra searches simultaneously, one from s and a backward search from t, until the searches meet is called *bidirectional* search. In this case, we denote by $\overrightarrow{\mathtt{D}}$, $\overrightarrow{\mathtt{Q}}$ and $\overrightarrow{q}$ the distances, queue and minimum queue key of the forward search and by $\overleftarrow{\mathtt{D}}$, $\overleftarrow{\mathtt{Q}}$ and $\overleftarrow{q}$ the respective data of the backward search. Typically, the searches are interleaved by alternating settling a vertex from each direction. Another common approach is to always settle a vertex from the direction with the smaller minimum queue key. The algorithm can terminate when the sum of the minimum keys in both queues $\overrightarrow{q} + \overleftarrow{q}$ is greater than the so far best known *tentative total distance* μ.

Dijkstra's algorithm can also be used to solve the TravelTimeProfileProblem. In this case, instead of scalar distances, tentative profiles are maintained in $\mathtt{D}$. Relaxing an edge uv becomes more complicated. First, one has to compute the temporary function $f = \mathtt{D}[u] \oplus \ell(uv)$. This is called *linking* the distance label $\mathtt{D}[u]$ with the travel time of the edge $\ell(uv)$. Then, the temporary function must be *merged* with the current label $\mathtt{D}[v]$, i.e. $\mathtt{D}[v](\tau) = \min(\mathtt{D}[v](\tau), f(\tau))$. The queue key of a vertex with distance $\mathtt{D}[v]$ is the lower bound $\min_\tau \mathtt{D}[v](\tau)$. This profile search is only label-correcting. The search can terminate when $q \geq \max_\tau \mathtt{D}[t](\tau)$.

5.3 A*

A* [HNR68] is a goal-directed extension of Dijkstra's algorithm. It uses a *potential* (sometimes also called *heuristic*) function π_t which maps vertices to an estimate of the remaining distance to t. A* orders the vertices in the priority queue by $\mathtt{D}[v] + \pi_t(v)$ instead of $\mathtt{D}[v]$ as Dijkstra's algorithm does it. Thus, vertices closer to the target are visited earlier, and the search space becomes smaller. A* is equivalent to running Dijkstra's algorithm on the graph with *reduced weights* [HNR68]. This reduced weight function is defined as $\ell_{\pi_t}(uv) = \ell(uv) - \pi_t(u) + \pi_t(v)$. A potential is called *feasible*, if $\ell_{\pi_t}(e) \geq 0$ for all edges e. If the employed potential is feasible, A* is label-setting and guaranteed to have found the optimal distance after settling t. It can also be guaranteed that A* has computed the shortest distance once t is settled when the estimates of the potential function are lower bounds of the remaining distances. However, with only lower bound potentials, A* is only label-correcting and the theoretical worst case running time is exponential. When the potential of the target is zero, i.e. $\pi_t(t) = 0$, feasibility also implies the lower bound property. Dijkstra's algorithm is a special case of A* with $\pi_t = 0$ for all vertices.

5.4 Contraction Hierarchies

Contraction Hierarchies (CH) [GSSV12] is a speedup technique to accelerate shortest path searches on time-independent road networks through precomputation. During the preprocessing, a total order $v_1 \prec \cdots \prec v_n$ of all vertices $v_i \in \mathcal{V}$ by "importance" is determined heuristically, where more important vertices should lie on more shortest paths. The position of a vertex in the order is also denoted as its *rank*. Vertices of higher rank are often informally referred to and visualized as "higher up" in the hierarchy. Therefore, an edge uv where $u \prec v$ is an *upward* edge. Analogously, when $v \prec u$, the edge uv is said to go *downward*.

Once such an importance ordering was obtained, all vertices are successively contracted by ascending importance. To *contract* a vertex means to temporarily remove it from the graph while inserting *shortcut edges* between more important neighbours to preserve shortest distances among them. The result is an *augmented graph* $G^+ = (\mathcal{V}, \mathcal{E}^+)$ with the original edges and the additional shortcuts and corresponding length function ℓ^+. Shortcut edges uv allow to "skip over" paths $(u, \ldots, w_i, \ldots, v)$ where $w_i \prec u$ and $w_i \prec v$. Therefore, $\ell^+(uv)$ must have the length of the shortest such path. We denote this distance through lower-ranked vertices as $\mathrm{dist}^{\prec}(u, v)$. Note that shortcut edges are only necessary if this length is also the distance between their endpoints. If there is a shorter path through higher-ranked vertices, the shortcut would be superfluous. Such unnecessary shortcuts are identified through a so called *witness search*, a local bidirectional Dijkstra search between the endpoints of a shortcut, before inserting a shortcut.

We often refer to the augmented graph G^+ split into an upward graph $G^\uparrow = (\mathcal{V}, \mathcal{E}^\uparrow)$ containing only upward edges and a downward graph $G^\downarrow = (\mathcal{V}, \mathcal{E}^\downarrow)$ containing only downward edges. The augmented graph has the property that between any two vertices s and t, there exists an *up-down-path* P with $\ell^+(P) = \mathrm{dist}_\ell(s, t)$ which uses first only edges from $\mathcal{E}^\uparrow$ and then only edges from $\mathcal{E}^\downarrow$. See Figure 5.1 for an illustration. From every shortest path in G, an up-down path of

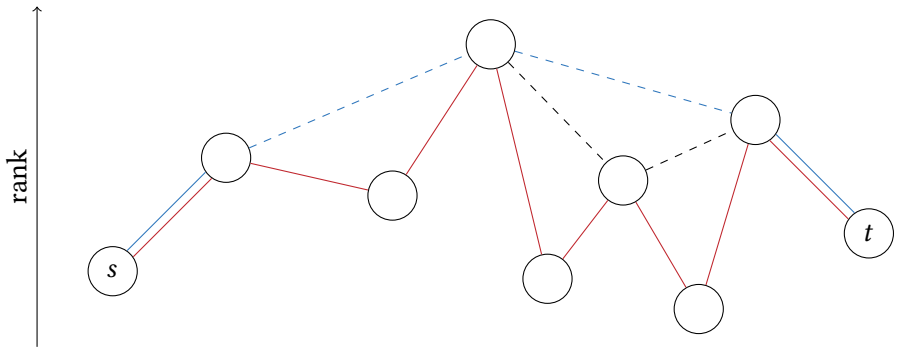


Figure 5.1: Solid lines are edges in G. Dotted lines are shortcuts. Red is a shortest st-path in G. Blue is equally long up-down st-path in G^+.

equal length in G^+ can be constructed. Such a path can be found by running the bidirectional variant of Dijkstra's algorithm from s on $G^\uparrow$ and from t on $\overleftarrow{G}^\downarrow$. By construction, at least the highest-ranked vertex on the up-down path must be in the intersection of the forward and backward search spaces. The stopping criterion of the bidirectional search must be altered slightly: The search can only terminate once the tentative distance μ exceeds *both* minimal queue keys. The set of vertices reachable in $G^\uparrow$ and $\overleftarrow{G}^\downarrow$ is called the *CH search space* of a vertex.

A shortest path in the augmented graph can be unpacked into the corresponding path in the original graph by recursively unpacking each shortcut. For this, one can, for example, store the ID of the vertex which was contracted when a shortcut was inserted with each shortcut. Alternatively, one can store the IDs of the two edges which the shortcuts represents.

In CH, vertices are sometimes organized into *levels*. Levels correspond to the contraction order but admit more than one vertex per level. A level assignment can be obtained as follows: All vertices without downward neighbours are assigned to level zero. Every other vertex goes into one level higher than the highest level of any downward neighbour.

5.4.1 (R)PHAST

PHAST [DGNW13] is a CH extension that computes distances from all vertices to one target (or vice versa, the reverse case works analogously). This is sometimes denoted as a *all-to-one* (or *one-to-all*) problem. The preprocessing phase remains the same as for CH. The query phase is split into two steps. The first step is analogue to the CH query: From t, all reachable vertices via reversed down-edges are explored by running Dijkstra's algorithm on $\overleftarrow{G}^\downarrow$. For the second step, PHAST utilizes an assignment of vertices into *levels*. These levels correspond to the importance ordering but allow multiple vertices in the same level. However, no edge must connect two vertices within the same level. Such a level assignment can be obtained by assigning all vertices without lower-ranked neighbours to the lowest level. All other vertices are iteratively assigned to one level above the highest level of any downward neighbour.

Algorithm 5.2: PHAST basic all-to-one search.

Data: $\mathtt{D}[v]$: tentative distance from any $v \in \mathcal{V}$ to t.

1 Run Dijkstra's algorithm (Algorithm 5.1) from t on $\overleftarrow{G}^{\downarrow}$, filling $\mathtt{D}$
2 **for** all CH levels $\mathcal{L}$ from most to least important **do**
3 **for** all up-edges $uv \in \mathcal{E}^{\uparrow}$ with u in $\mathcal{L}$ **do**
4 **if** $\mathtt{D}[u] > \mathtt{D}[v] + \ell^{+}(uv)$ **then**
5 $\mathtt{D}[u] \leftarrow \mathtt{D}[v] + \ell^{+}(uv)$

The main work of the second step consists in iterating over all CH levels from top to bottom. In each iteration, all up-edges starting within the current level are relaxed in reverse. Once all levels were processed, the shortest distances from all vertices to t were computed. Pseudocode is provided in Algorithm 5.2. PHAST is faster than Dijkstra's algorithm on road graphs because it is a better fit for modern processor architectures, better at utilizing data locality and, most importantly, can be parallelized very effectively.

RPHAST [DGW11], short for Restricted PHAST, is a PHAST extension for efficiently computing distances from a smaller set of source vertices to one target vertex (again, the reverse case works analogously), solving the *many-to-one* problem. Given a set of source vertices $\mathcal{S}$, the first step is to copy the combined search space of all sources into a *restricted subgraph*. Let $\mathcal{V}_{\text{restr}}$ be the set of vertices reachable in $G^{\uparrow}$ from any $s_i \in \mathcal{S}$. The restricted subgraph $G^{\uparrow}_{\text{restr}}$ is a subgraph of $G^{\uparrow}$ induced by $\mathcal{V}_{\text{rest}}$. Finding and copying the relevant edges into this restricted subgraph is called the *selection* step. In the query step, a target t is given, and the PHAST algorithm is applied, but the downward sweep (the second step of the PHAST algorithm) is performed only on the restricted subgraph. RPHAST is particularly effective when many targets are queried for the same source set $\mathcal{S}$. If the source set changes often, selection times may become problematic.

5.4.2 Bucket Query

Bucket queries are another way to approach the many-to-one problem with CH [Kno+07]. For every vertex v, a bucket is maintained containing pairs of distances to specific sources and the respective source $(\text{dist}_{G^{\uparrow}}(s_i, v), s)$. Buckets are filled by running Dijkstra's algorithm from each source vertex s_i on $G^{\uparrow}$. For every settled vertex, the respective entries are inserted into the vertices' bucket. For competitive performance, applying *stall-on-demand* [GSSV12] is crucial. Now, distances from a specific target t to all sources can be computed. Like a classical CH query, Dijkstra's algorithm is executed from t on $\overleftarrow{G}^{\downarrow}$. However, an array of tentative distances of length $|\mathcal{S}|$ is maintained. The ith position in this array contains the tentative distance for source s_i. When the backward search settles a vertex v, the algorithm iterates over the bucket entries of v and updates the tentative distances accordingly. Once the backward search terminates, the array contains the shortest distances from all sources. Like RPHAST, the bucket approach is efficient when many targets are queried for the same source set $\mathcal{S}$.

5.5 Timestamp Arrays

CH queries are fast because the search space is orders of magnitude smaller than the number of vertices in the graph. However, a naive implementation of Dijkstra's algorithm contains an initialization phase with running time in $\Theta(n)$. We avoid this problem by using *timestamp arrays* to track tentative distances [Wei97]. Beside each distance entry $\texttt{D}[i]$, we maintain a timestamp $\texttt{TS}[i]$ and a counter variable $\texttt{t}$. A distance array entry is only valid when the corresponding timestamp matches the counter. Otherwise the default value, i.e. ∞, is used. When an entry is written, the corresponding timestamp is set to the value of the counter. Now, all distances can be reset in $\mathcal{O}(1)$ by incrementing the counter variable.

5.6 Periodic Piecewise Linear Functions

We represent PPLFs as a sorted sequence of breakpoints $(\texttt{dt}_i, \texttt{tt}_i)$ of departure time and travel time. W.l.o.g. let $H = [0, p)$ be the horizon of our functions and p the period. The first breakpoint is always at time zero, the last one at a time smaller than p. We evaluate PPLFs for a time τ by first wrapping the time into the horizon, i.e. $\tau' = \tau \mod p$. Then, we perform a binary search for τ'. If this search exactly matches a point, we return its travel time. Otherwise, τ' is between $\texttt{dt}_i$ and $\texttt{dt}_{i+1}$, and we perform linear interpolation, i.e. we return:

$$\texttt{tt}_i + (\texttt{tt}_{i+1} - \texttt{tt}_i) \frac{\tau' - \texttt{dt}_i}{\texttt{dt}_{i+1} - \texttt{dt}_i}$$

It has been reported that an index array or linear search might accelerate the function evaluation [Bat14]. We could not reproduce any speedups. A reason may be, that our algorithms primarily work on functions with few breakpoints. Therefore, we only use binary search.

Linking and Merging. Linking and Merging PPLFs is practically surprisingly complicated. The implementations of KaTCH and our own code illustrate this.[1] Both operations can be implemented in linear time through coordinated linear sweeps over the breakpoints. When linking two functions $f \oplus g$, the result functions has for every breakpoint $(\texttt{dt}_i^f, \texttt{tt}_i^f)$ of f a breakpoint $(\texttt{dt}_i^f, \texttt{tt}_i^f + g(\texttt{dt}_i^f + \texttt{tt}_i^f))$, and for every breakpoint $(\texttt{dt}_i^g, \texttt{tt}_i^g)$ of g, a breakpoint $(f^{-1}(\texttt{dt}_i^g), \texttt{dt}_i^g - f^{-1}(\texttt{dt}_i^g) + \texttt{tt}_i^g)$. For merging two functions f and g, the result function has every breakpoint $(\texttt{dt}_i^f, \texttt{tt}_i^f)$ of f where $f(\texttt{dt}_i^f) \leq g(\texttt{dt}_i^f)$ and vice versa, and additional breakpoints where the functions intersect. The difficulty lies in determining these intersections and handling the numerical instabilities around this operation.

[1] https://github.com/GVeitBatz/KaTCH/blob/master/datastr/base/pwl_ttf.h
https://github.com/kit-algo/rust_road_router/blob/master/engine/src/datastr/graph/floating_time_dependent/piecewise_linear_function.rs

6 A Fast and Tight Heuristic for A* in Road Networks

Solving navigation problems through accelerated shortest path computations on graphs modelling the road network has been a very successful approach. A multitude of effective techniques has been developed for the two-phase shortest path problem [Bas+16]. However, for many real-world applications, the basic model is too simplistic. For realistic routing, many additional features need to be considered. This includes predicted and dynamic traffic but also user preferences and turn costs and restrictions. Many applications may have additional application-specific requirements. Further, it is insufficient to handle each of these features independently. Instead, all requirements must be supported in combination.

Extending Dijkstra's algorithm to support these features is comparatively easy. In contrast, extending speedup techniques is vastly more complex. The results on adopting specific speedup techniques to extended problem fill many research papers [GV11, BGSV13, BDPW16, DSW16, DGPW17, BWZZ20] and sometimes entire dissertations [Del09, Bat14, Bau18] (see Section 1.1). While techniques achieving fast query times have been successfully developed for many extended scenarios, we observe two problems: The resulting techniques are complex to implement and hard to extend further. For example, while there exist speedup techniques achieving fast queries for each of the features mentioned above, we are unaware of any work supporting their combination. In practice, a different trade-off between running times and extensibility is necessary. A unified and extensible approach with manageable implementation complexity is often more important than the fastest query performance. Therefore, in this chapter, we prioritize a simple and extensible approach over the fastest possible query times.

Attribution. This chapter is based on joint work with Ben Strasser. The results have been previously published as a conference paper at SEA 2021 [SZ21] and a journal article [SZ22].

Contribution and Outline. We revisit the A* algorithm and propose a flexible, unified framework for routing problems. It can be applied to any problem where tight lower bounds are available at preprocessing time. The core of our approach is a new CH-based A* heuristic. It allows for much tighter estimates than previous A* heuristics and, thus, significantly faster queries. While the query running times of our technique are not competitive with approaches tailored to specific problems, they are often sufficient for practical applications. Further, our approach only requires the classical CH preprocessing regardless of the specific extended routing problem. Therefore, preprocessing time and memory consumption are typically significantly better than approaches aiming for competitive query times in specific problems.

The remainder of this chapter is organized as follows: In Section 6.1, we introduce Lazy RPHAST, a simple and efficient CH-based algorithm for the incremental many-to-one shortest path problem. It is the first algorithm to efficiently support accelerated shortest path computations from *dynamic* source sets to a fixed target. Section 6.2 contains several optimizations for A* in road networks accelerating the processing of low-degree vertices (Section 6.2.1) and an improved variant of bidirectional A* (Section 6.2.2). Our main contribution, the CH-Potentials framework, is presented in Section 6.3. We first show how to use Lazy RPHAST to build a fast and tight heuristic for A* in road networks. Based on this heuristic, we provide a unified framework (Section 6.3.1 and 6.3.2) for a variety of practical route planning problems (Section 6.3.3). CH-Potentials can be applied to all of these problems without any modifications to the preprocessing. In Section 6.4, we provide an extensive experimental evaluation analyzing the performance characteristics of our algorithms. It shows that CH-Potentials achieve decent running times when tight lower bounds are available at preprocessing time. However, the strength of our approach is not in query running times but its flexibility: Problem extensions can be supported without adjustments to the preprocessing. The price for each extension is a query slowdown depending on how tight the lower bounds used during preprocessing remain.

Problem Statement. In this chapter, we do not aim at a specific model but consider a variety of two-phase routing problems. This includes the classical shortest path problem, the shortest path problem with dynamic traffic weights and the time-dependent shortest path problem.

6.1 The Incremental Many-to-One Problem

Before approaching our main problem, we first focus on a different problem model, which naturally arises from A* heuristics. Here, the target vertex t is known in the selection step, but the source vertices $s_1, \dots, s_k$ are queried one after another. We denote this problem as the *incremental many-to-one* problem. The first step is the *target selection* where the target vertex t is provided. Then, an arbitrary number of source vertices are given one after another. For each source s_i the distance $\text{dist}(s_i, t)$ has to be computed before the next source s_{i+1} is provided.

We consider the combined running time of the target selection and each incremental query as the total running time to answer an incremental many-to-one query. Since the target selection

Algorithm 6.1: The Lazy RPHAST algorithm.

Data: $\texttt{D}^{\downarrow}[v]$: tentative distance from any vertex $v \in \mathcal{V}$ to t as computed by Algorithm 5.1 on $\overleftarrow{G}^{\downarrow}$.
Data: $\texttt{D}[v]$: memoized distance from vertex $v \in \mathcal{V}$ to t, shared between invocations.

```
1 Function Select(t):
2     Run Dijkstra's algorithm (Algorithm 5.1) from t on G↓, filling D↓
3     D[v] ← ⊥ for all v ∈ V
4 Function ComputeAndMemoizeDist(u):
5     if D[u] = ⊥ then
6         D[u] ← D↓[u]
7         for all up-edges uv ∈ E↑ do
8             D[u] ← min(D[u], ℓ+(uv) + ComputeAndMemoizeDist(v))
9     return D[u]
```

time is included, computing the distances to all vertices with Dijkstra or PHAST is too slow. Also, since the source set is provided incrementally, RPHAST in its basic form is not well suited to our problem. Fortunately, we can do better.

6.1.1 Lazy RPHAST

The core idea of our algorithm is to do the RPHAST computation lazily using memoization. In the target selection, we first run the backward CH search on $\overleftarrow{G}^{\downarrow}$ from t to obtain an array $\texttt{D}^{\downarrow}$. $\texttt{D}^{\downarrow}[v]$ is the minimum down vt-path distance or $+\infty$, if there is no such path. The distances $\texttt{D}[v]$ are initialized to a sentinel value $\perp$ distinct from any other valid distance (including ∞). This value indicates that the distance from v to t has not yet been computed.

Now, distances from many sources s_i to t can be computed incrementally by calling the `ComputeAndMemoizeDist` function as shown in Algorithm 6.1. The key to doing this efficiently is reusing the distance information `D` across invocations through memoization. Thus, the first step of the algorithm is always to check if the distance for the requested vertex has already been computed. If this is the case, it immediately returns this distance. If not, the distance $\texttt{D}[s_i]$ is initialized to the shortest down-path distance $\texttt{D}^{\downarrow}[s_i]$ obtained by the backward search. Then, the algorithm iterates over all up-edges $s_i v$ and checks if the up-down path through this neighbour can improve the distance. The algorithm is invoked recursively to obtain the shortest distance from a neighbour v to t.

Correctness. Due to [GSSV12], an up-down path of shortest distance must exist in G^+. Further, G^+ can be decomposed into two directed acyclic graphs (DAGs) $G^{\uparrow}$ and $G^{\downarrow}$ and the up path

can be found in $G^{\uparrow}$ and the down path in $G^{\downarrow}$. The Lazy RPHAST selection finds the shortest down path in $G^{\downarrow}$. Now observe that the `ComputeAndMemoizeDist` function is, in fact, a recursive depth-first search (DFS) on $G^{\uparrow}$. Edges uv are relaxed once all upward neighbors of u have been processed, i.e. in DFS-post order. A DFS-post order also is a reverse topological order for $G^{\uparrow}$. Therefore, the algorithm relaxes edges uv in reverse topological order of the tail vertices u. Since $G^{\uparrow}$ is a DAG, this yields shortest up distances in $G^{\uparrow}$. Concatenated with the down paths obtained by the backward search, this yields optimal shortest distances.

6.2 Optimizations for A* in Road Networks

In this section, we propose optimizations for A* in road networks. First, we present several low-degree vertex optimizations which exploit road network characteristics to reduce the overhead of queue operations and heuristic evaluations. Second, we discuss the bidirectional A* algorithm and propose an improved pruning criterion for symmetric bidirectional potentials. These optimizations can be used with *any* A* heuristic.

6.2.1 Low-Degree A* Improvements

Preliminary experiments showed that a significant amount of query running time is spent in heuristic evaluations and queue operations. We can reduce both by keeping some vertices out of the queue, as the heuristic only needs to be evaluated when a vertex is pushed into the queue. For example, consider a chain of vertices with precisely two neighbours. Traversing this chain by successively pushing each vertex into the queue, evaluating the heuristic and popping the vertex again from the queue appears quite wasteful. Therefore, we now explore techniques to process such vertices consecutively without using the queue. The techniques discussed here are essentially a lazy variant of the ideas used in TopoCore [DSW15].

Skip Degree-Two Vertices. Recall that $\overleftrightarrow{\deg}(u)$ is the number of neighbours v such that $vu \in \mathcal{E}$ or $uv \in \mathcal{E}$. Our algorithm differs from classical A* when removing a vertex u from the queue. A* iterates over the outgoing edges uv of u and tries to reduce $\mathtt{D}[v]$ by relaxing uv. If A* succeeds, v's weight in the queue is set to $\mathtt{D}[v] + \pi_t(v)$. Our algorithm, however, behaves differently, if $\overleftrightarrow{\deg}(v) \leq 2$. Our algorithm determines the longest degree two chain of vertices $u, v_1, \ldots, v_k, w$ such that $\overleftrightarrow{\deg}(v_i) = 2$ and $\overleftrightarrow{\deg}(w) > 2$. If our algorithm succeeds in reducing $\mathtt{D}[v_1]$, it does not push v_1 into the queue. Instead, it iteratively tries to reduce all $\mathtt{D}[v_i]$. It stops if a $\mathtt{D}[v_i]$ cannot be improved. If it does not reach w, then only $\mathtt{D}$ is modified, but no queue action is performed. If $\mathtt{D}[w]$ is modified and $\overleftrightarrow{\deg}(w) > 2$, w's weight in the queue is set to $\mathtt{D}[w] + \pi_t(w)$.

As the target vertex t might have degree two, our algorithm cannot rely on stopping when t is removed from the queue. Instead, our algorithm stops as soon as $\mathtt{D}[t]$ is less than the minimum weight in the queue.

Skip Degree-Three Vertices. We can also skip some degree-three vertices. Denote by $u, v_1, \ldots, v_k, w$ a degree two chain as described in the previous section. If $\overleftrightarrow{\deg}(w) > 3$ or w is in the queue, our algorithm proceeds as in the previous section. Otherwise, there exist up to two degree chains $w, x_1, \ldots, x_p, y$ and $w, a_1, \ldots, a_q, b$ such that $x_1 \neq v_k \neq a_1$. Our algorithm iteratively tries to reduce all $\mathtt{D}[x_i]$ and $\mathtt{D}[a_i]$. If it reaches b, b's weight in the queue is set to $\mathtt{D}[b] + \pi_t(b)$. Analogously, if y is reached, y's weight is set to $\mathtt{D}[y] + \pi_t(y)$. If neither y nor b are reached, no queue operation is performed. Using this method, we avoid pushing every other degree-three vertex into the queue.

Stay in Largest Biconnected Component. Many vertices in road networks lead to dead ends. Unless the source or target is in this dead-end, it is unnecessary to explore these vertices.

In the preprocessing phase, we compute the subgraph G_C, called *core*. G_C is induced by the largest biconnected component of the undirected graph underlying G. We compute the core using Tarjan's algorithm [Tar72]. For every vertex v in the input graph G, we store an *attachment vertex* a_v to the core. For vertices in the core, $a_v = v$. For every vertex v outside of the core, the attachment vertex a_v is the cut vertex in the core that separates v's component from the core (or a sentinel value $\perp$ for vertices in components not connected to the core).

The query phase is divided into two steps. First, we run A* on the subgraph induced by the core and s's component. Formulated differently, we only consider vertices which are in the core or have the same attachment vertex as s. We achieve this implicitly by not following edges to vertices without this property. If t is part of G_C or in the same component as s, this A* search finds it. Otherwise, we find a_t. In that case, we continue by searching a path from a_t towards t restricted to t's component. The final result is the concatenation of both paths. When t is not connected to the core ($a_t = \perp$) but s is ($a_s \neq \perp$), we immediately return a distance of ∞.

6.2.2 Bidirectional A*

On road graphs, bidirectional search provides a simple way to halve the practical running time of Dijkstra's algorithm. Thus, a bidirectional variant of A* also seems desirable. However, as shown in [GH05], the necessary modifications are not straightforward. We revisit bidirectional A* and propose an alternative approach. Our experiments show that it is competitive with the solution described by [GH05].

The straightforward idea is to use two heuristics $\overrightarrow{\pi}_t(v)$ and $\overleftarrow{\pi}_s(v)$. The forward search has its queue ordered by $\overrightarrow{\mathtt{D}}[v] + \overrightarrow{\pi}_t(v)$, where $\overrightarrow{\pi}_t(v)$ estimates the distance $\mathrm{dist}(v, t)$ from v to t. Similarly, the backward search has its queue ordered by $\overleftarrow{\mathtt{D}}[v] + \overleftarrow{\pi}_s(v)$, where $\overleftarrow{\pi}_s(v)$ is an estimate of the distance $\mathrm{dist}(s, v)$ from s to v.

The problem with this straightforward approach is that these two potentials induce different reduced weights (see Section 5.3). Thus, each direction would run on a different graph in the equivalent bidirectional Dijkstra search. This breaks the usual bidirectional Dijkstra stopping criterion. To the best of our knowledge, no better stopping criterion exists than running *both*

searches until the unidirectional stopping criterion is met for one direction. The forward search can skip vertices already settled by the backward search and vice versa. Unfortunately, this straightforward bidirectional A* still performs more work than a unidirectional A* [GH05]. Thus, on its own, it is not a useful algorithm. However, it can serve as a basis for further algorithmic refinements. The authors of [GH05] refer to this as *symmetric* bidirectional A*.

To obtain a bidirectional stopping criterion, the *average potential* is proposed [GH05]. It combines a forward and a backward heuristic $\overrightarrow{\pi}_t$ and $\overleftarrow{\pi}_s$ into a combined *average heuristic* $\pi_{\overline{st}}$. The idea is to use a common reduced graph, whose weights are the average weights of the individual reduced graphs. Both searches run on the same common reduced graph. This allows stopping the searches when $\overrightarrow{q} + \overleftarrow{q} \geq \mu$. Formally, $\pi_{\overline{st}}(v)$ is defined as $(\overrightarrow{\pi}_t(v) - \overleftarrow{\pi}_s(v))/2$. The forward search uses $\pi_{\overline{st}}(v)$ as its heuristic. The backward search uses $-\pi_{\overline{st}}(v)$. Unfortunately, average potentials have two downsides. First, evaluating the average potential requires evaluating both $\overrightarrow{\pi}_t$ and $\overleftarrow{\pi}_s$. Evaluating the average heuristic is, therefore, slower than evaluating just one of the composing heuristics. Second, the bidirectional stopping criterion comes at the cost of worse estimates for each direction on its own. $\pi_{\overline{st}}(v)$ is a worse estimate for $\text{dist}(v,t)$ than $\overrightarrow{\pi}_t$. Similarly, $-\pi_{\overline{st}}(v)$ is a worse estimate for $\text{dist}(s,v)$ than $\overleftarrow{\pi}_s$.[1] The second downside can be partially mitigated through pruning with the composing heuristics. When the forward search scans an edge uv where $\overrightarrow{\mathrm{D}}[u] + \ell(uv) + \overrightarrow{\pi}_s(v) > \mu$ holds, i.e. the distance plus the estimate of the remaining distance is already greater than the currently known tentative distance, it is not necessary to push v into the queue. The pruning rule for the backward search is analogous.

To avoid the downsides of the average potential, we revisit symmetric bidirectional A* and propose a new pruning criterion. We describe the idea for the forward search. The pruning rule for the backward search is analogous. The central idea consists of using information from the backward search to prune edges in the forward search. We do not use the average heuristic. Instead of a strong stopping criterion, we use a pruning rule that gets stronger the longer the search runs. Such a pruning rule will eventually prune all remaining branches and stop the search. The stopping criteria for each direction remain the same (unidirectional) as before. However, usually, the search stops early because the queues are empty.

Let uv be an edge that we relax in the forward search. Before pushing v into the queue, we apply the new pruning rule. If we can prove that every path using uv is at least as long as the shortest known path length μ, then we do not have to push v. We therefore want to obtain a lower bound for $\text{dist}(s,u) + \ell(uv) + \text{dist}(v,t)$. As u was settled, $\overrightarrow{\mathrm{D}}[u]$ contains the shortest path length $\text{dist}(s,u)$, i.e., $\text{dist}(s,u) = \overrightarrow{\mathrm{D}}[u]$. $\ell(uv)$ is also known as it is just an edge weight. It remains to lower bound $\text{dist}(v,t)$. Vertices are removed from the backward queue ordered by $\overleftarrow{\pi}_s(v) + \text{dist}(v,t)$. If v was not yet removed from the backward queue, we know that $\overleftarrow{\pi}_s(v) + \text{dist}(v,t) \geq \overleftarrow{q}$. This gives us the required lower bound, i.e. $\text{dist}(v,t) \geq \overleftarrow{q} - \overleftarrow{\pi}_s(v)$. Thus, v does not have to be pushed if $\overrightarrow{\mathrm{D}}[u] + \ell(uv) + \overleftarrow{q} - \overleftarrow{\pi}_s(v) \geq \mu$. The vertex v might still be

[1] To obtain an actual lower bound from this average heuristic, one has to add $\overleftarrow{\pi}_s(t)/2$ in the forward case and $\overrightarrow{\pi}_t(s)/2$ in the backward. Adding any constant to a heuristic function does not change the reduced graph.

pushed into the queue when there is another edge wv for which pruning is impossible. Checking the pruning rule requires evaluating the backward heuristic. However, pruning is only possible once the searches have met, i.e. $\mu < \infty$. Before that, each direction only has to evaluate its own heuristic. Thus, our pruning improves on both downsides of the average potential.

Unless stated differently, for all bidirectional A* variants, we always alternate between removing a vertex from the forward and the backward queues. We also evaluate expanding the search with the smaller minimum queue key in our experiments. While this may sound sensible, our experiments show in Table 6.3 that it is never beneficial in terms of running time.

6.3 The CH-Potentials Framework

In this section, we introduce an algorithmic framework to apply A* with a Lazy RPHAST-based heuristic to various practical route planning problems. We call our approach *CH-Potentials*. The core idea is to compute a CH augmented graph during preprocessing and use A* with a straightforward application of Lazy RPHAST as the heuristic to answer queries. When the CH preprocessing and the A* algorithm are performed on the same graph with the same weight function, this yields a *perfect* heuristic. However, this case is, of course, not particularly interesting. One could just answer the shortest path query directly with a CH query. The approach becomes useful when the query runs on a *different but related* graph or weight function than the preprocessing. Therefore, we start by establishing a common formal framework for the use of CH-Potentials. Then, we exemplarily describe some extended routing problems and how to apply the CH-Potentials framework to them.

6.3.1 Formal Problem Setup: Inputs, Outputs, and Phases

We consider a variety of different applications with slightly different problem models. The goal is always to answer many shortest path queries quickly. To describe our framework, we establish a shared notation: Input to each query are vertices s and t, and a graph G_{q} with query weights ℓ_{q}. However, the precise formal inputs of the query and what exactly ℓ_{q} represents depends on the application. In the simplest case, ℓ_{q} will be scalar edge weights. Live traffic is an example of this. The challenge in this scenario is that values of ℓ_{q} might change between queries. However, ℓ_{q} can also represent something more complex than scalar numbers. It can be any function that computes a weight for an edge. This function can also take additional parameters from the state of the search. In the case of traffic predictions, ℓ_{q} is a function which maps the edge entry time to the traversal time, and the query takes an additional departure time parameter.

To enable quick shortest path computations, we consider a two-phase setup with an additional offline preprocessing phase before the online query phase. The input to the preprocessing phase is a lower bound graph $G_{\mathrm{pre}} = (\mathcal{V}_{\mathrm{pre}}, \mathcal{E}_{\mathrm{pre}})$ with lower bound weights ℓ_{pre} where $\ell_{\mathrm{pre}}(e)$ must be a scalar value for every edge e of G_{pre}. The preprocessing output is auxiliary data that allows to

quickly compute distances on G_{pre} with respect to ℓ_{pre}. In the applications considered in this chapter, ℓ_{pre} is always the *free-flow* travel time.

The query phase may use this auxiliary data to answer shortest path queries between vertices s and t on $G_{\text{q}} = (\mathcal{V}_{\text{q}}, \mathcal{E}_{\text{q}})$ with weights ℓ_{q}. Let $\phi : \mathcal{V}_{\text{q}} \to \mathcal{V}_{\text{pre}}$ denote a function mapping vertices in the query graph to vertices in the lower bound graph. The only requirement for a routing problem to fit into our problem framework is that the query weight of an edge $\ell_{\text{q}}(uv)$ is greater or equal to the shortest distance $\text{dist}_{\text{pre}}(\phi(u), \phi(v))$ between the corresponding vertices $\phi(u)$ and $\phi(v)$ in the lower bound graph G_{pre} with respect to ℓ_{pre}. Unless stated otherwise, G_{q} and G_{pre} are the same graph, ϕ is the identity function and only ℓ_{q} changes for the queries.

6.3.2 CH-Potentials

CH-Potentials can be used to solve any problem in this setup. The preprocessing is always the computation of the CH augmented graph G^{+}_{pre} and remains the same regardless of the specific routing problem. The query consists of A* with the heuristic function $\pi_t(v) = \text{dist}_{\text{pre}}(\phi(v), \phi(t))$ computed using Lazy RPHAST. At the beginning of each query, we perform the target selection, i.e. a backward CH search, from the target t. The heuristic function $\pi_t(v)$ is implemented by a call to the `ComputeAndMemoizeDist` for vertex $\phi(v)$ (see Algorithm 6.1). In contrast to the preprocessing phase, the exact implementation of the A* search depends on the application. Our approach only provides the heuristic π_t for the A* search. As the performance of A* depends on the accuracy of the heuristic estimates, the smaller the difference between query weights and lower bound distances, the better CH-Potentials will perform.

Correctness. Our heuristic is always *feasible*, i.e. $\ell_{\text{q}}(uv) - \pi_t(u) + \pi_t(v) \geq 0$ holds for all edges. By requirement and because of the triangle inequality, the following must hold:

$$\ell_{\text{q}}(uv) - \pi_t(u) + \pi_t(v) \geq \text{dist}_{\text{pre}}(\phi(u), \phi(v)) - \text{dist}_{\text{pre}}(\phi(u), \phi(t)) + \text{dist}_{\text{pre}}(\phi(v), \phi(t)) \geq 0$$

Thus, A* will always determine the optimal shortest distance.

6.3.3 Applications

Avoiding Tunnels and/or Highways

Avoiding tunnels or highways is a common feature of navigation devices. Implementing this feature with CH-Potentials is easy. We set ℓ_{pre} to the free-flow travel time. If an edge is a tunnel or a highway, we set ℓ_{q} to $+\infty$. Otherwise, ℓ_{q} is set to the free-flow travel time.

Forbidden Turns and Turn Costs

The classical shortest path problem allows changing edges at vertices freely. However, in the real world, turn restrictions, such as a forbidden left or right turn, exist. Also, taking a left

turn might take longer than going straight. This can be modeled using turn weights [GV11, DGPW17]. A *turn weight* $\ell_t : \mathcal{E} \times \mathcal{E} \to \mathbb{Z}^{\geq 0}$ maps a pair of incident edges onto the turning time or $+\infty$ for forbidden turns. A path with vertices $v_1, v_2, \ldots v_k$ has the following *turn-aware weight*:

$$\ell(v_1v_2) + \sum_{i=2}^{k-1} \ell_t(v_{i-1}v_i, v_iv_{i+1}) + \ell(v_iv_{i+1})$$

The objective is to find a path between two given edges with minimum turn-aware weight. The first term $\ell(v_1, v_2)$ is the same for all paths, as it only depends on the source edge. It can thus be ignored during optimization.

We solve this problem by constructing a *turn-expanded* graph [Cal61, Win02]. This construction expands the network so that road segments become vertices and allowed turns become edges; see fig. 7.2 (middle) for an example. More precisely, the expanded graph $G_e = (\mathcal{V}_e, \mathcal{E}_e)$ is obtained from G as follows. The vertices of G_e are the edges of G, i.e, $\mathcal{V}_e = \mathcal{E}$. The edges of G_e are the allowed turns of G, i.e., $\mathcal{E}_e = \{(uv, vw) : uv, vw \in \mathcal{E}, \ell_t(uv, vw) \neq \infty\}$. The cost of an edge $(uv, vw) \in \mathcal{E}_e$ is defined as $\ell_e((uv, vw)) = \ell_t(uv, vw) + \ell(vw)$. A sequence of expanded vertices in the expanded graph G_e corresponds to a sequence of edges in the input network. The weight of a path in G_e is equal to the turn-aware weight of the corresponding path in G minus the irrelevant $\ell(v_1v_2)$ term. Thus, the turn-aware routing problem can be solved by searching for shortest paths in G_e. Therefore, we set $G_q = G_e$.

For the preprocessing, we use zero as the lower bound for every turn weight in the potential. Thus, the preprocessing uses the input graph without turns, i.e. $G_{pre} = G$ and $\ell_{pre} = \ell$. As preprocessing and query use different graphs, we define the vertex mapping function ϕ as $\phi(uv) = v$. Obviously, $\ell_q((uv, vw)) = \ell_t(uv, vw) + \ell(vw) \geq \text{dist}_{pre}(\phi(uv), \phi(vw))$ and this approach yields a feasible heuristic. Sadly, the undirected graph underlying G_q is always biconnected, if the input graph is strongly connected. The BCC optimization described in Section 6.2.1 is therefore ineffective. With this setup, CH-Potentials supports turn costs without requiring turn information in the CH.

Predicted Traffic and Time-Dependent Routing

We can apply CH-Potentials to the two-phase time-dependent shortest path problem with FIFO travel times to support traffic predictions. Queries use the input graph with time-dependent weights ℓ_q. For the preprocessing, we set $\ell_{pre}(e) = \min_\tau \ell_q(e, \tau)$, that is the scalar minimum travel time of each edge. Queries and preprocessing both use the same topology, i.e. $G_{pre} = G_q$. Therefore, preprocessing weights are a trivial lower bound of the query weights.

With this setup, we keep time-dependency out of the CH preprocessing. Thus, we avoid a lot of algorithmic complications compared to [DN12, BGSV13, BDPW16] (and Chapter 8) where shortcuts of travel time functions have to be constructed. Note that TD-ALT [DW07, NDSL12] supported time-dependent routing with a similar strategy.

Live Traffic

Besides predicted traffic, we also consider dynamic real-time traffic. We propose three variants to handle this traffic information with CH-Potentials. In the simplest approach, we set ℓ_{pre} to the free-flow travel time without any traffic ℓ_{free}. ℓ_{q} consists of scalar edge weights set to the travel time accounting for current traffic. As traffic only increases the travel time along an edge, ℓ_{free} is a valid lower bound for ℓ_{q}. Values from ℓ_{q} can be updated between queries. Therefore, CH-Potentials support the two-phase shortest path problem with dynamic traffic weights.

This setup can easily be extended to the combined traffic shortest path problem. The free-flow travel times remain valid lower bounds. We set the query weights ℓ_{q} to the combined traffic function ℓ_{comb}, i.e. we use the live traffic information for some time (for example, one hour) and then switch back to the predicted traffic. See Section 3.3 for the formal definition of ℓ_{comb}. In our implementation, we evaluate the formulas according to the definition of ℓ_{comb} for each travel time evaluation. Therefore, real-time traffic updates can still be integrated instantaneously.

Finally, we can also phase out the live traffic after some time without time-dependent traffic predictions and instead switch back to free-flow weights. Using the notation from Section 3.3, this would mean setting $\ell_{\text{pred}} = \ell_{\text{free}}$, running time-dependent queries, and evaluating the ℓ_{comb} definition at query time.

With this setup, CH-Potentials supports a combination of real-time and predicted traffic. We did not make any modifications that would hinder a combination with other extensions. Further adding tunnel or highway avoidance or turn-aware routing is simple. This straightforward integration of complex routing problems is the strength of CH-Potentials.

6.4 Evaluation

Environment. Our benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 128 GiB of DDR4-2133 RAM and an Intel Xeon E5-1630 v3 CPUs, which has four cores clocked at 3.7 Ghz and 4×32 KiB of L1, 8×256 KiB of L2, and 10 MiB of shared L3 cache. All experiments were performed sequentially. Our code is written in Rust and compiled with `rustc 1.57.0-nightly` in the release profile with the `target-cpu=native` option. The source code of our implementation and the experimental evaluation can be found on GitHub.[2]

Inputs. Our primary benchmark instance for this chapter is the OSM Germany instance with the Mapbox real-world traffic data. Turn restrictions are extracted from OSM. In this chapter, we only use the heavy real-time traffic snapshot. Further, we use the DIMACS Europe instance to ensure comparability with other speedup techniques. For the same reason, we also include the time-dependent Ger06, Eur17 and Eur20 instances.

[2] `https://github.com/kit-algo/ch_potentials`

Methodology. To evaluate point-to-point queries, we generate 10 000 queries where both source and target are vertices drawn uniformly at random and report average results. For time-dependent queries, we draw the departure time uniformly at random. When using live traffic snapshots, departure times are fixed to the time of the snapshot. Lazy RPHAST is evaluated with many-to-one queries where each query consists of a source set $\mathcal{S}$ with 2^{14} sources and one target vertex t. However, instead of picking sources and targets from the full vertex set $\mathcal{V}$, we draw them from local subsets of vertices $\mathcal{B}$ of varying size $|\mathcal{B}|$ called *balls*. A ball $\mathcal{B}$ is generated by picking a centre uniformly at random and running Dijkstra's algorithm from it until the desired number of vertices $|\mathcal{B}|$ is settled. This allows us to evaluate the performance depending on the distribution of the vertices. Since we use a fixed number of sources per query, a small ball size means that the vertices are densely clustered in the same region, while large ball sizes mean that the vertices are distributed over large parts of the network. For each ball size, we generate 100 balls. We pick one set of sources from each ball to which we compute distances from 100 different targets selected uniformly at random from the same ball. Therefore, each reported running time is the mean over 10 000 queries. With this, we follow the methodology from [DGW11].

Preprocessing and Space Consumption. CH-Potentials inherit preprocessing times and space usage characteristics directly from CH. Therefore, they are not the focus of our evaluation. Preprocessing on all graphs takes at most five minutes and produces less than 1.2 GB of auxiliary data. See [GSSV12] for a detailed discussion.

6.4.1 Lazy RPHAST

To evaluate Lazy RPHAST in the incremental setting, we measure the elapsed running time after distances from 2^i sources were queried. Figure 6.1 depicts the total elapsed time and the average running time to compute a single distance. The first few distances are the most expensive since much of the CH search space has not been explored yet. With around 100 μs, the running times are comparable to standard CH queries. For later queries, little work remains to be done, and the overhead per distance becomes almost constant depending on the ball size.

Despite laziness being the distinguishing feature in Lazy RPHAST, the algorithm can still be used for non-incremental many-to-one queries. Figure 6.2 depicts average running times to compute distances between one target vertex and 2^{14} sources for different ball sizes. The query generation methodology is the same as in the previous experiment. As this is the same methodology also used in [DGW11], we can relate our results to the performance of other one-to-many algorithms. Keep in mind that algorithms like RPHAST optimize for a different setting than we do. We can compare the performance for fixed $\mathcal{S}$-t terminal sets. The difference is that with RPHAST, one can efficiently compute distances from a different t' to the same $\mathcal{S}$ set, while with Lazy RPHAST, one can efficiently extend $\mathcal{S}$ while t stays the same.

When looking at total running times for a single many-to-one query, including selection times, Lazy RPHAST delivers very competitive performance and is as fast if not faster than

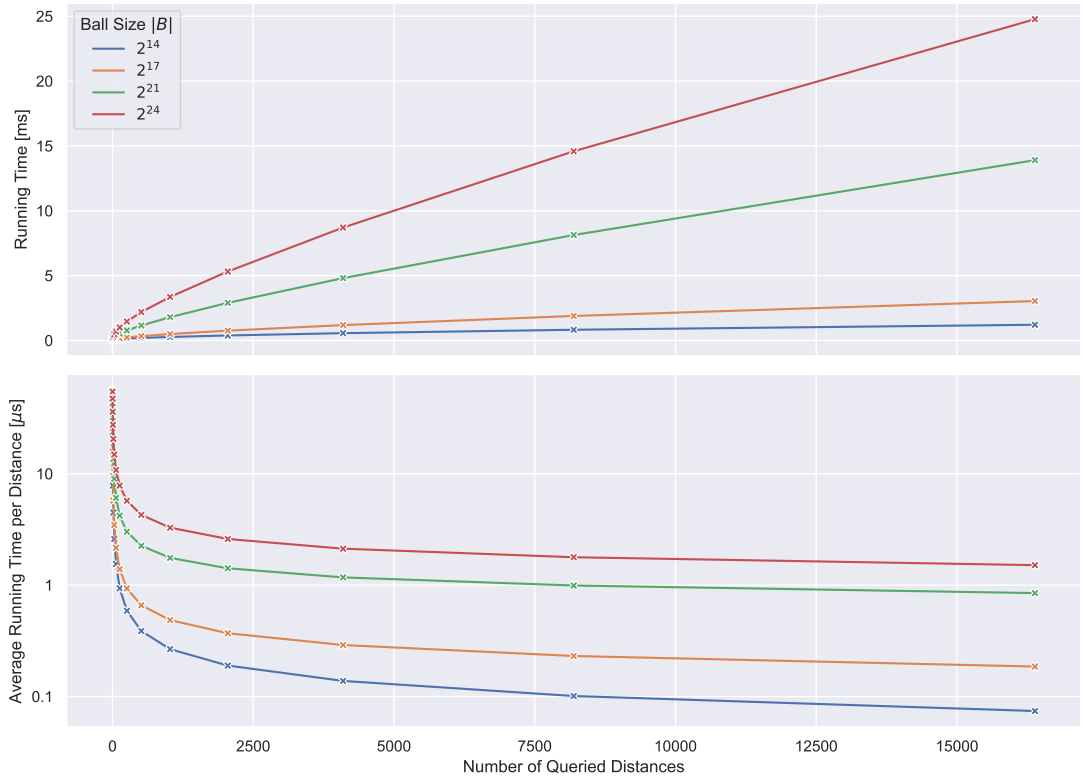


Figure 6.1: Average running times of incremental Lazy RPHAST while querying $|\mathcal{S}| = 2^{14}$ from a ball of varying size $|\mathcal{B}|$ on OSM Germany excluding selection times. The upper figure contains the total elapsed running time. The lower figure contains the averaged running time per source, i.e. y/x from the upper figure. Note the different y-axis scales and units.

all algorithms evaluated in [DGW11]. For example, Delling et al. report an average RPHAST running time of 1.97 ms for selection and query combined for $|\mathcal{B}| = 2^{14}$ and 28.52 ms for $|\mathcal{B}| = 2^{20}$ on DIMACS Europe. Lazy RPHAST takes 0.96 ms and 10.76 ms respectively, to compute the same distances. These numbers are not perfectly comparable due to different benchmark machines.[3] Therefore, in Appendix B, we also report running times of our own implementation of RPHAST. They confirm that RPHAST and Lazy RPHAST perform similar and Lazy RPHAST is for a single many-to-one query marginally faster. We can conclude that Lazy RPHAST is a valuable

[3] According to the comparison methodology from [Bas+16] (see `https://i11www.iti.kit.edu/~pajor/survey/`), the machine used in [DGW11] (SPA-2) is about 30% slower than ours (we obtained a score of 33 159 ms). However, these scaling factors have to be interpreted very carefully. They are obtained from one-to-all Dijkstra searches on continental-sized road networks. This experiment heavily emphasizes memory bandwidth while neglecting other critical factors such as CPU frequency and cache size and speed, which are likely much more critical for algorithms carefully tuned to exploit data locality.

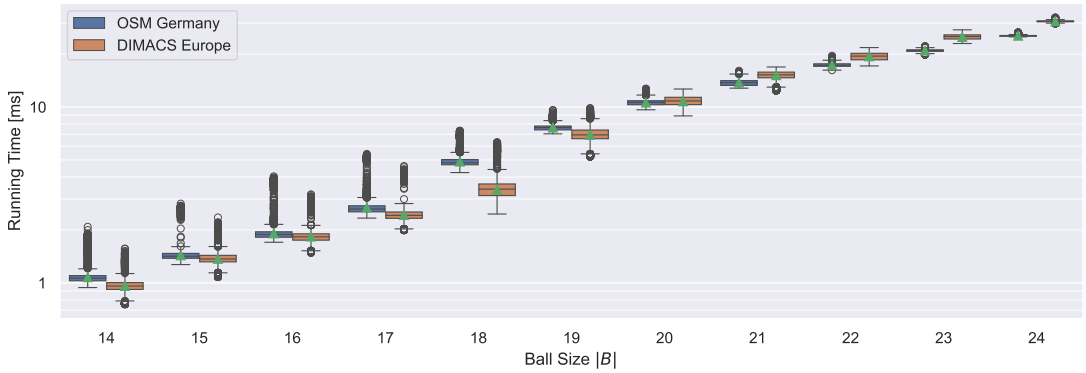


Figure 6.2: Running times of Lazy RPHAST for many-to-one queries with $|\mathcal{S}| = 2^{14}$ sources picked from a ball of varying size $|\mathcal{B}|$. The running time includes the selection and the time to compute all distances. The boxes cover the range between the first and third quartile. The band in the box indicates the median, the diamond the mean. The whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

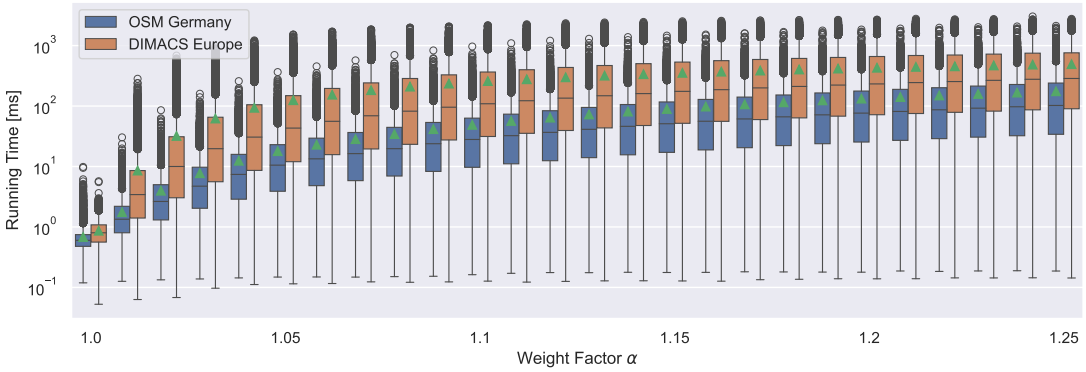


Figure 6.3: Running times on a logarithmic scale for queries on static road networks with scaled edge weights $\ell_q = \alpha \cdot \ell_{pre}$. The boxes cover the range between the first and third quartile. The band in the box indicates the median, the diamond the mean. The whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

extension to RPHAST. It allows for efficiently handling dynamic source sets and is competitive in the setting where both the target and the source set change between queries.

6.4.2 CH-Potentials Heuristic

The performance of A* depends on the tightness of the heuristic and the overhead of evaluating the heuristic. CH-Potentials computes optimal distance estimates with respect to ℓ_{pre}. However,

Table 6.1: Average query running times and number of queue pushes with different heuristics and optimizations on OSM Ger with $\ell_q = 1.05 \cdot \ell_{pre}$. The BCC, Deg2 and Deg3 columns indicate which optimizations from Section 6.2.1 were used.

	BCC	Deg2	Deg3	Zero	ALT	CH	Oracle
Running time [ms]	○	○	○	2 133.0	317.9	47.9	34.3
	●	○	○	1 355.3	233.9	36.3	24.8
	●	●	○	753.4	122.6	19.5	12.7
	●	●	●	580.7	90.8	15.9	10.1
Queue [$\cdot 10^3$]	○	○	○	8 087.1	863.1	137.1	137.1
	●	○	○	6 298.2	685.7	112.7	112.7
	●	●	○	2 901.4	303.4	43.3	43.3
	●	●	●	1 681.4	179.7	26.8	26.8

for most applications, there will be a gap between ℓ_q and ℓ_{pre} (otherwise, one could use CH without A*). We evaluate the impact of the difference between ℓ_q and ℓ_{pre} on the performance of A*. The lower bound ℓ_{pre} is set to the free-flow travel time. The query weights ℓ_q are set to $\alpha \cdot \ell_{pre}$, where $\alpha \geq 1$. Increasing α degrades the heuristic's quality. Figure 6.3 depicts the results. Clearly, α has a significant influence on the running time. Average running times range from below a millisecond to a few hundred milliseconds depending on α. Up to around $\alpha = 1.1$ the running time grows quickly. For $\alpha > 1.1$, the growth slows down. This illustrates the strengths and limits of our approach and goal-directed search in general. CH-Potentials can only achieve competitive running times if the application allows tight lower bounds at preprocessing time.

We observe that the running times for a fixed α fluctuate heavily. This is an interesting observation, as with uniform source and target sampling, nearly all queries are long-distance. The query distance is thus not the reason. After some investigation, we concluded that this is due to non-uniform road network density. Some regions have more roads per area than others. The number of vertices explored by A* depends on the density of the search space area. As the density varies, the running times vary.

Table 6.1 depicts the performance of unidirectional A* with different heuristics and optimizations on OSM Ger with $\ell_q = 1.05 \cdot \ell_{pre}$. The factor 1.05 was chosen to resemble realistic problem settings where goal-directed search can achieve reasonable speedups (compare to Table 6.5). We compare CH-Potentials to three other heuristics. The first heuristic is the *Zero* heuristic where $\pi_t(v) = 0$ for all vertices v. This corresponds to using Dijkstra's algorithm. Secondly, we compare against our implementation of ALT [GW05]. We use 16 landmarks generated with the avoid strategy [GW05] and activate all during every query. Finally, we compare against a hypothetical *Oracle-A** heuristic. This heuristic has instant access to a shortest distance array with respect to ℓ_{pre}, i.e. it is faster than the fastest heuristic possible in our model. We fill this array before each query using a reverse Dijkstra search from the target vertex but do not include

Table 6.2: Performance of different variants of bidirectional A* on OSM Ger with $\ell_q = 1.05 \cdot \ell_{pre}$. All variants alternate between the forward and the backward search.

Low Deg. Opt.	Bidir. Pot.	New Pruning	Running time [ms]				Queue pushes [$\cdot 10^3$]		
			Zero	ALT	CH	Oracle	Zero	ALT	CH/ Oracle
○	Avg.	○	1 441.41	126.46	62.61	37.29	4 493.97	292.01	125.16
○	Avg.	●	1 451.96	128.20	62.48	38.89	4 491.56	290.92	125.08
○	Sym.	○	5 779.64	795.56	122.70	88.66	16 042.82	1 688.60	259.78
○	Sym.	●	1 453.58	261.80	59.22	37.37	4 491.56	624.25	116.71
●	Avg.	○	365.82	33.22	19.34	9.96	916.15	57.27	23.60
●	Avg.	●	369.51	33.37	19.54	9.98	908.55	56.09	23.25
●	Sym.	○	1 512.48	241.27	40.98	26.36	3 317.81	334.90	44.67
●	Sym.	●	368.94	72.67	21.54	11.22	908.55	123.77	20.72

the running time for this step. Thus, the reported running times of Oracle-A* do *not* account for any heuristic evaluation. CH-Potentials computes the same distance estimates, but the heuristic evaluation has some overhead. Comparing against Oracle-A* allows us to measure this overhead. No other heuristic, which only has access to the preprocessing weights, can be faster than Oracle-A*.

We observe that the number of queue pushes roughly correlates with running time. Each optimization reduces both queue pushes and running times. All optimizations yield a combined speedup of around 3. CH-Potentials outperforms ALT by a factor of between six and seven and settle correspondingly fewer vertices. This is not surprising since ALT computes worse distance estimates. In contrast, CH-Potentials already computes exact distances with respect to ℓ_{pre}. As CH-Potentials and Oracle-A* have the same heuristic values, the number of queue pushes are also equal. The only difference between CH-Potentials and Oracle-A* is the overhead of the heuristic evaluation. This overhead leads to a slowdown of around 1.6. Thus, CH-Potentials is already very close to the best possible heuristic in this model. No competing algorithm such as ALT or CPD-Heuristics can be significantly faster.

6.4.3 Bidirectional A*

In this section, we investigate the performance of bidirectional A*. We first evaluate different variants of bidirectional A* in Table 6.2 and 6.3 then compare the best ones against unidirectional A* in Table 6.4. Table 6.2 studies the impact of our improved pruning and the low-degree optimizations. As observed in the previous section, enabling the low-degree optimizations achieves a speedup of roughly three. Symmetric bidirectional A* without our improved pruning has the worst performance. Enabling the improved pruning improves the performance of

Table 6.3: Performance of different direction selection criteria of bidirectional A* on OSM Ger with different query weights. The symmetric variant uses the improved pruning, the average variant does not. All variants use all low-degree optimizations.

ℓ_q	Bidir. Pot.	Choose Direction	Running time [ms]				Queue pushes [$\cdot 10^3$]		
			Zero	ALT	CH	Oracle	Zero	ALT	CH/ Oracle
ℓ_{pre}	Avg.	Alternating	373.18	12.83	0.79	0.18	916.15	23.08	0.60
	Avg.	Min. Key	406.35	13.68	1.44	0.56	986.40	26.39	1.15
	Sym.	Alternating	376.72	40.19	0.69	0.19	908.55	76.61	0.57
	Sym.	Min. Key	427.51	50.46	1.77	0.83	978.62	99.62	1.44
$\ell_{pre} \cdot 1.05$	Avg.	Alternating	365.82	33.22	19.34	9.96	916.15	57.27	23.60
	Avg.	Min. Key	391.70	38.06	21.76	11.30	986.41	67.65	26.42
	Sym.	Alternating	368.94	72.67	21.54	11.22	908.55	123.77	20.72
	Sym.	Min. Key	394.38	84.84	27.28	14.53	978.63	145.28	24.82
$\ell_{pre} \cdot 1.5$ if speed < 80kph	Avg.	Alternating	361.83	19.50	10.92	5.34	845.06	34.03	13.25
	Avg.	Min. Key	391.47	31.65	21.05	11.00	917.13	52.23	23.78
	Sym.	Alternating	364.55	37.33	11.89	6.00	836.44	57.93	11.53
	Sym.	Min. Key	395.04	54.90	23.36	12.54	908.12	84.33	22.01

symmetric bidirectional A* significantly. For all heuristics except ALT, symmetric A* with improved pruning has smaller search spaces than the average potential and similar running times. Without the low-degree improvements, the improved symmetric variant is marginally faster. With the low-degree improvements, the average potential remains slightly faster. This is due to the reduced impact of the heuristic evaluation overhead with the low-degree optimizations. Enabling the improved pruning for the average potential reduces the search space size marginally and slightly increases running times.

Table 6.3 shows the performance of bidirectional A* with different strategies to decide whether to advance the forward or the backward search next. The results clearly show that alternating the directions is always superior. Selecting the direction by minimum queue key may lead to huge imbalances in the progress of the searches. This causes the searches to meet later and the total search space to grow significantly.

In Table 6.4, we investigate the effectiveness of bidirectional search compared to unidirectional search depending on the query weights. Interestingly, only the zero heuristic and ALT consistently achieve speedups through bidirectional search. With $\ell_q = \ell_{pre}$, unidirectional CH-Potentials is already optimal and only traverses the shortest path. Here, the bidirectional search will only introduce unnecessary overhead. When query weights are scaled up uniformly, bidirectional search achieves some search space reduction. However, it is not enough to significantly

Table 6.4: Performance of bidirectional and unidirectional A* on OSM Ger with different query weights. The symmetric variant uses the improved pruning, the average variant does not. All variants use all low-degree optimizations.

ℓ_q		Running time [ms]				Queue pushes [$\cdot 10^3$]		
		Zero	ALT	CH	Oracle	Zero	ALT	CH/ Oracle
ℓ_{pre}	Unidirectional	584.87	43.02	0.47	0.16	1 674.35	96.21	0.66
	Average	373.18	12.83	0.79	0.18	916.15	23.08	0.60
	Symmetric	376.72	40.19	0.69	0.19	908.55	76.61	0.57
$\ell_{pre} \cdot 1.05$	Unidirectional	580.66	90.79	15.91	10.06	1 681.39	179.66	26.78
	Average	365.82	33.22	19.34	9.96	916.15	57.27	23.60
	Symmetric	368.94	72.67	21.54	11.22	908.55	123.77	20.72
$\ell_{pre} \cdot 1.5$ if speed $<$ 80kph	Unidirectional	637.24	96.62	21.78	14.62	1 674.26	171.02	36.54
	Average	361.83	19.50	10.92	5.34	845.06	34.03	13.25
	Symmetric	364.55	37.33	11.89	6.00	836.44	57.93	11.53

reduce running times due to the overhead of running a second search. This changes drastically when the query weight increases are applied non-uniformly in the third scenario. Here only weights with speed less than 80 kph were scaled up. This touches only the beginning and end of most shortest paths between randomly chosen vertices. The middle part of the shortest paths will typically use faster edges like highways. In this case, bidirectional CH-Potentials is a factor of two faster than the unidirectional variant. This is because the search space of a unidirectional search expands greatly while exploring the end of the path to the target where the reduced weights are bad. In contrast, the bidirectional searches meet in the middle of the shortest path where the reduced weights are close to zero, thus avoiding this expansion. This is also why ALT behaves like this for all query weights. By construction, the ALT heuristic has better reduced weights for edges which lie on many shortest paths like highways. Conversely, unimportant edges have bad reduced weights. This makes bidirectional search so critical for the ALT performance. In contrast, a potential as tight as CH-Potentials makes bidirectional search in many scenarios unnecessary. Bidirectional search for CH-Potentials only pays off when the reduced weights are bad around the terminals.

6.4.4 Applications

Table 6.5 depicts the running times of CH-Potentials in various applications as described in Section 6.3.3. We report speedups compared to extensions of Dijkstra's algorithm for each application. Note that we draw source and target vertices uniformly at random, i.e. we typically perform long-range queries spanning distances of at least four hours (OSM Germany). We start

Table 6.5: CH-Potentials performance for different route planning applications with random queries. Depending on the problem, we apply unidirectional or bidirectional (U/B) CH-Potentials. We report average running times and the number of queue pushes. We also report the average length increase, that is, how much longer the final shortest distance is compared to the lower bound. Finally, we report the average running time of Dijkstra's algorithm as a baseline and the speedup over this baseline.

			Running time [ms]	Queue [$\cdot 10^3$]	Length incr. [%]	Dijkstra [ms]	Speedup
DIMACS Eur	Unmodified $\ell_q = \ell_{pre}$	U	0.9	1.1	0.0	2 106.0	2 405.8
OSM Ger	Unmodified $\ell_q = \ell_{pre}$	U	0.6	0.5	0.0	2 182.6	3 795.4
	No Tunnels	U	29.2	46.8	5.2	2 198.0	75.2
		B	33.4	35.7	5.2	2 198.0	65.8
	No Highways	U	378.7	583.8	42.5	1 992.5	5.3
		B	433.1	481.6	42.5	1 992.5	4.6
	Live	U	129.4	193.9	15.0	2 119.3	16.4
		B	193.6	188.8	15.0	2 119.3	10.9
	Temporary Live	U	3.1	4.2	3.4	2 127.9	685.1
	Turns	U	3.0	5.7	1.1	4 708.2	1 556.0
		B	1.1	0.8	1.1	4 708.2	4 223.8
	TD	U	120.8	104.4	12.3	3 133.7	25.9
	TD + Live	U	117.4	99.9	19.0	3 436.5	29.3
	TD + Live + Turns	U	265.1	375.7	20.0	6 420.5	24.2
Ger06	TD	U	4.2	6.4	3.1	603.5	144.2
Eur17	TD	U	80.4	79.8	3.9	3 454.3	43.0
Eur20	TD	U	97.7	72.8	4.2	5 060.2	51.8

with the base case where $\ell_q = \ell_{pre}$. This is the problem variant solved by CH. CH achieves average query running times of 0.16 ms on OSM Ger. CH-Potentials is roughly four times slower but still achieves a speedup of 3795 over Dijkstra. This shows that CH-Potentials gracefully converges toward CH in the $\ell_q = \ell_{pre}$ special case. On DIMACS Europe, the average degree is somewhat higher, making the low-degree optimizations less effective and leading to more queue operations and a slightly slower running time.

In the other scenarios, the performance of CH-Potentials strongly depends on the quality of the estimates. We measure this quality using the length increase of ℓ_q compared to ℓ_{pre}. Avoiding highways results in the greatest length increase and the smallest speedup. The other extreme are turn costs. They have little impact on the length increase. The achieved speedups are, therefore, comparable to CH speedups. Since turn costs and restrictions appear primarily in the beginning and end of shortest paths and not in the middle on highways, utilizing bidirectional search

results in even better speedups. Mapbox live traffic has a length increase of around 15%, which yields running times of 130 ms. Applying bidirectional CH-Potentials, in this case, is detrimental to the performance because the bad reduced edge weights appear in the middle of shortest paths. The same is the case for forbidden tunnels or highways. Using live traffic data only for an hour and then switching back to free-flow weights (the Temporary Live variant) leads to much better performance with speedups over Dijkstra of almost three orders of magnitude. This is because edge weights differing from the free-flow weights appear only at the beginning of a query. Once the search reaches a distance greater than the validity of the live traffic, the potentials will yield perfect distance estimates and the remaining search is extremely fast.

The length increase of Mapbox traffic predictions is about 12.3%, and results in a running time of 120 ms. The speedup in the predicted scenario is larger than in the live setting, as the travel time function evaluations slow down Dijkstra's algorithm. Combining predicted and live traffic results in a running time of 117 ms. Surprisingly, this is faster than the pure TD scenario, even though the length increase is greater. This is due to the departure times being fixed to 15:41, the time of the live traffic snapshot. Running time-dependent queries without live traffic with this fixed departure instead of uniformly sampled departure times results in slightly faster running times of around 106 ms. Queries with this departure are faster because a large part of the query occurs during the evening and the night when the predicted traffic is closer to free-flow traffic, and the potentials are thus tighter.[4] This is why the total length increase does not correspond perfectly to the running times in this case. The performance is only fully explained when we also consider the location of the increased edge weights.

Adding turn restrictions additionally increases the running times significantly. This increase is primarily due to the graph becoming two to three times larger and the BCC optimization of Section 6.2.1 becoming ineffective when considering turns. It is not due to the length increase of using turns. With everything activated, our algorithm still has a speedup of 24.2 over the baseline. Interestingly, the PTV traffic predictions have a much smaller length increase than the Mapbox predictions. This results in somewhat faster running times of our algorithm.

Comparison with Related Work. While the query running times reported in Table 6.5 are decent in many settings, they are not competitive with techniques tailored to specific applications. In the simple $\ell_q = \ell_{pre}$ setting, Hub Labels can be used to answer queries in less than a microsecond [DGW13], more than three orders of magnitude faster than with CH-Potentials. Live traffic and arbitrary weight functions can be handled with CCH resulting in query times of around 0.1 ms [BSW19]. For time-dependent routing, TCH [BGSV13] achieves query times more than five times faster than CH-Potentials (0.75 ms compared to our 4.2 ms on Ger06). Again, these numbers are not perfectly comparable due to different benchmark machines. Nevertheless, the overall picture is clear enough. However, the advantage of the CH-Potentials framework over these fine-tuned techniques is that it is a unified and flexible approach

[4] See also the evaluation of Chapter 9 for a discussion of time-dependent A* query performance depending on the departure time.

which can handle all these applications without any adjustments to the preprocessing. Further, CH-Potentials preprocessing times are at least an order of magnitude faster than approaches like TCH (more than two hours on Eur20 with 16 cores, compared to five minutes sequentially with CH-Potentials) and require significantly less memory (Hub Labels, for example, needs 20 GB on DIMACS Europe, compared to around 770 MB with CH-Potentials). Finally, to the best of our knowledge, for problem settings such as the combination of predicted and live traffic, there does not exist any exact technique to handle this setting, let alone to integrate turn costs additionally. The key achievement of CH-Potentials is that problem extensions can be integrated by trading query performance rather than developing new algorithms.

6.5 Conclusion

In this article, we introduced CH-Potentials, a fast, exact, and flexible two-phase routing framework based on A* and CH. The approach can handle complex, integrated routing scenarios with little implementation complexity and no changes to the preprocessing algorithms. CH-Potentials provides *exact* distances for lower bound weights known at preprocessing time as an A* heuristic. Thus, the query performance of CH-Potentials crucially depends on the availability of reasonable lower bounds in the preprocessing phase. Our experiments show that this availability highly depends on the application. We also show that the overhead of our heuristic is within a factor 1.6 of a hypothetical A*-heuristic that can instantly access lower bound distances. Achieving significantly faster running times could still be possible in variations of the problem setting. The core building block of our approach is Lazy RPHAST, a new CH query variant for the incremental many-to-one problem. We showed that it also delivers competitive performance for many-to-one problems. This leads to multiple avenues for future research.

Dropping the provable exactness requirement using a setup similar to anytime A* [ZH02, LGT03] would be interesting. Another promising research avenue would be to investigate graphs other than road networks. Much research for grid maps exists, including a series of competitions called GPPC [Stu+15]. Hierarchical techniques have been shown to work well on these graphs [UK14]. It might also be worthwhile to apply CH-Potentials to even more routing applications, for example, to the shortest ϵ-smooth path problem [DSS18] or the problem of computing alternative routes [BDGS11, ADGW13, Kob15]. CH-Potentials performed very well for turn costs and restrictions and further studies in this setting might be interesting. For example, CH-Potentials could also be used on the compact turn model [GV11, DGPW17] for reduced memory consumption. It would even be possible to consider time-dependent turn costs to model traffic light patterns. Many-to-one problems also appear as subproblems in other routing algorithms, for example, in nearest neighbour computations [BW21]. Investigating whether Lazy RPHAST can be used to improve these algorithms appears to be a worthwhile direction for future research. Studying the performance of Lazy RPHAST in a many-to-many context would also be interesting.

7 The Customizable Contraction Hierarchies Framework

A routing framework for modern map and navigation applications requires far more than an effective speedup technique for the quick computation of shortest paths [DGPW17]. While the shortest path computation should not be the bottleneck of the routing engine, there are many additional requirements. A practical algorithm must handle not only car travel times but also other length functions such as walking, biking, or user-defined preferences, for example, to avoid highways. Further, *dynamic* routing considering real-time traffic is critical. Therefore, updates to the length function must be fast enough for frequent adjustments. Integrating turn costs and restrictions is also an essential requirement for high-quality routing. Another crucial practical feature is the computation of not only the shortest path but a set of reasonable alternatives. Finally, point-of-interest queries are also an important feature in such applications.

To the best of our knowledge, the only speedup technique currently supporting this entire set of requirements is CRP/MLD [DW15, DGPW17]. While the CH-Potentials framework proposed in the previous chapter also supports many of these requirements, running times may be too slow in some cases because they depend on the tightness of the preprocessing lower bounds. However, inspired by CRP, CH has also been extended to a three-phase setup [DSW16]. This variant is called Customizable Contraction Hierarchies (CCH). Classical CH uses heuristic ad-hoc methods to obtain a "good" ordering for a given weight function. Also, the construction of the augmented graph uses heuristic witness searches to achieve reasonable preprocessing running times. These techniques only work well for well-behaved weight functions with a strong hierarchy. In contrast, CCH draws on nested dissection orders for the vertex ranking and algorithms from chordal graph theory for the augmented graph construction. This yields surprisingly elegant and simple algorithms which also have stronger theoretical guarantees. CCH queries are roughly as fast as CH queries, i.e. around an order of magnitude faster than

CRP. On the downside, the CCH customization is slower than the CRP customization thus far. Also, CCH currently does not support all the features mentioned above. For example, computing alternative routes with CCH has never been studied. Still, CCH is an area of active research. In the past few years, there have been several research articles [BSW19, GHUW19, BWZZ20, BW21] improving and extending CCH, pushing it closer to feature parity with CRP.

Attribution. This chapter is partially based on joint work with Valentin Buchhold, Dorothea Wagner and Michael Zündorf. Specifically, the part on CCH with turn costs is an extended version of a part of our publication at ATMOS 2020 [BWZZ20]. The improved chordal completion preprocessing algorithm was developed in the Bachelor's thesis of Michael Zündorf [Zün19]. The improvements in customization algorithms and the nearest neighbour algorithm also profited from much back and forth with my colleague Valentin Buchhold.

Contribution. In this chapter, we give a detailed overview of the state of the art of CCH. We review recent advances on CCH, show how to combine them and propose additional refinements. We adopt the Lazy RPHAST algorithm from the previous chapter and propose a specialized variant for CCH. This allows us to extend CH-Potentials to CCH-Potentials. With this building block, CCH can support all problem variants necessary for a fully featured routing framework. An extensive evaluation confirms that a CCH(-Potentials) framework is not only comprehensive in supported features but also competitive in performance to both CH and CRP. This chapter also establishes the technical fundamentals of a well-tuned CCH implementation on which we build in the following chapters.

Problem Statement. This chapter focuses on the three-phase shortest path problem and important practical extensions.

Outline. In the next three sections, we discuss the CCH algorithms for each of the three phases. We present the fastest known variants for each phase and discuss their efficient implementation. In Section 7.4, we discuss query variants for extended problems. Section 7.5 contains an extensive experimental evaluation.

7.1 Metric-Independent Preprocessing

The CCH metric-independent preprocessing phase works purely on the topology of the input network. Since changes to the topology are reasonably rare in the context of road networks, running time requirements for this phase are relatively lax. Running times of hours would still be manageable. Nevertheless, faster algorithms are, of course, always an improvement.

CCH algorithms for this phase are defined on undirected graphs. Therefore, initially, for any edge $uv \in \mathcal{E}$ where $vu \notin \mathcal{E}$, vu is added to $\mathcal{E}$ so the graph becomes *bidirected*. Then, similarly to classical CH, an order must be computed where the vertices are sorted by their "importance".

The final step of the metric-independent preprocessing is the construction of the topology of a CH augmented graph G^+. This step is conceptually similar to the classical CH contraction. However, as there are no edge lengths, there can be no witness search, and thus, all possible shortcuts will be part of G^+. That means when a vertex v is contracted, the upward neighbours of v will be completed into a clique.

This construction opens up interesting connections to graph theory [BCRW16]. The construction directly corresponds to the so-called *elimination game* which has been extensively studied [Par61, RT78, Heg06]. In the elimination game, vertices are successively taken out of the graph while the neighbourhood of each vertex is completed to a clique. The goal is to minimize the number of inserted edges. In this context, the vertex order is also called an *elimination order*. An elimination order also induces a so-called *elimination tree*. The elimination tree is the subgraph of G^+ with only the edges from each vertex to its lowest-ranked upward neighbour. Further, we can observe that G^+ must be a *chordal graph*, i.e. a graph without induced cycles of length greater than three. Moreover, the contraction order is a *perfect* elimination order for G^+, i.e. no more additional edges will be inserted. In fact, G^+ is the minimal supergraph of G which fulfils these properties. Therefore, G^+ is the *minimal chordally completed supergraph* of G induced by the contraction order.

7.1.1 Ordering

The most crucial step of the metric-independent preprocessing is obtaining an "importance" order. In theory, any order leads to correct queries. However, good orders are critical for competitive performance. With bad orders, the augmented graph may even become prohibitively large. As no witness search is possible without a weight function, all possible shortcuts will be inserted in the contraction step. Thus, the number of additional edges in the augmented graph can quickly become too large, even for the main memory of high-end server machines. This even happens with "good" classical CH orders [Zei13].

Therefore, CCH is built on *nested dissection orders* [Geo73, BCRW16] which are a very effective heuristic for elimination orders. A nested dissection order is obtained by recursively computing small vertex separators, which split the graph into two (or more) roughly balanced *cells*. The vertices in the separator are assigned the highest ranks, i.e. appear last in the contraction order. Removing these separator vertices leaves two (or more) disconnected cells. The algorithm is applied recursively to order the vertices in these cells. The vertex order within a separator is arbitrary. Fortunately, vertices with a high rank in this order also lie on many shortest paths. All shortest paths from one cell to the other must use a vertex of the separator. Thus, a nested dissection order is also a good CH order.

Computing small balanced separators is an NP-hard problem [GJS76]. Thus, in general, one cannot expect to obtain small balanced separators efficiently. Fortunately, road networks are easily separable due to natural features such as rivers or mountain ranges. InertialFlow [SS15] is an example of a simple yet surprisingly effective partitioning algorithm for road networks. For this algorithm, vertices are projected on a geographic axis (the north-south axis, for example).

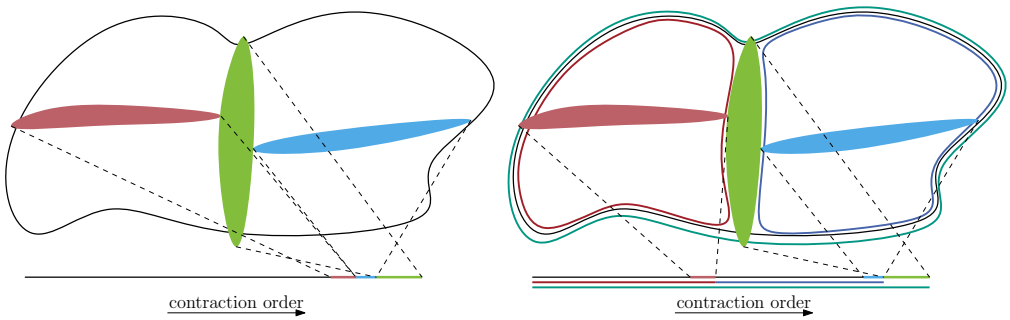


Figure 7.1: Visualization of different contraction orders derived from the same separator decomposition. Left: A BFS-post order. The top-level separator vertices appear last in the order. Directly before them all separator vertices of the level below. Right: A DFS-post order. Again, the top-level separator vertices appear last in the order. Before them are the separator vertices of the blue cell *and* the entire blue cell. The red cell completely appears at the beginning of the order.

The first and last quarter of vertices are contracted, and a minimum cut between these contracted vertices is computed. The vertices on one side of the cut edges make up a separator candidate. Repeating this process for different axes and taking the smallest separator yields surprisingly good results.

More sophisticated algorithms for partitioning [SS12, SS13] road networks [DGRW11, HS18] have also been developed. Some open-source implementations are available and can be deployed as black-box ordering algorithms. A comprehensive review is beyond the scope of this work. See [GHUW19] for a recent experimental comparison. To the best of our knowledge, InertialFlowCutter (IFC) [GHUW19], a combination of InertialFlow [SS15] and FlowCutter [HS18], currently yields the best results. We use IFC orders for all experiments in this work.

To maximize cache efficiency, the vertices of each cell should form a consecutive range in the ordering, i.e. the contraction order should be derived as a DFS-post order on the tree of the separator decomposition. See Figure 7.1 for an illustration. However, InertialFlowCutter computes a BFS-post ordering where separator vertices of each level of the decomposition appear consecutively. Fortunately, we can reconstruct the underlying separator decomposition as described in Section 7.1.4 and derive the corresponding DFS-post order from that decomposition.

7.1.2 Contraction

Once an importance order has been obtained, the remaining part of the metric-independent preprocessing is to compute the topology of the augmented graph $G^+ = (\mathcal{V}, \mathcal{E}^+)$. For cache efficiency in this and all following phases, it is crucial to permute the vertex IDs such that IDs equal the ranks. Since the graph is fully bidirected ($G^\uparrow = \overleftarrow{G^\downarrow}$), it is sufficient to store every edge only at its lower-ranked endpoint, i.e. maintain $G^\uparrow$. This graph must contain *every* shortcut edge which might be relevant for *any* weight function. Thus, the simplest way to construct this graph

is to perform the classical CH preprocessing for the given order without any witness search, i.e. iterate over all vertices by ascending rank and ensure that a shortcut edge exists between any pair of upward neighbours of each vertex. The result of this algorithm is the minimal chordal supergraph of G for which the importance ordering is a perfect elimination scheme. Because this naive approach is relatively expensive in terms of running time, the authors of [DSW16] propose a faster algorithm based on the *quotient graph* [GL78]. However, this algorithm is quite complex. Here, we describe a simpler and even faster algorithm. This algorithm has previously only been described in a Bachelor's thesis [Zün19]. It is heavily based on the linear-time chordal graph recognition algorithm of [HMPV00].

The algorithm iterates over all vertices in order of ascending rank. Let v be the current vertex and $\mathtt{N[v]}$ its upward neighborhood in $G^{\uparrow}$. If this neighbourhood is non-empty, the algorithm obtains the upward neighbour u with minimal rank. The remaining neighborhood $\mathtt{N[v]} \setminus u$ of v is merged into the neighborhood of u, i.e. $\mathtt{N[u]} \leftarrow \mathtt{N[u]} \cup \mathtt{N[v]} \setminus u$. The result is the minimal chordal supergraph of G induced by the nested dissection order.

This algorithm can be implemented by maintaining the neighbourhood of each vertex in a dynamic array. A theoretical worst-case running time of $\mathcal{O}(n + |\mathcal{E}^{\uparrow}|)$ can be achieved by appending the neighbourhoods without checking for duplicates and only performing the deduplication when extracting the minimally ranked upward neighbour. See [Zün19] for a proof of the running time and the correctness. Practically, however, it is even more efficient to ensure that the neighbourhoods never contain duplicates and are always sorted by rank. Merging neighbourhoods can then be realized with a coordinated linear sweep. The worst-case running time of this variant is slightly worse, but in practice, running times are faster.

7.1.3 Elimination Tree

The parent of a vertex v in the elimination tree is its lowest-ranked neighbour in $G^{\uparrow}$. We represent the elimination tree as an array $\mathtt{ET}$ of length n containing the parent for each vertex at the position of the ID of the vertex. As root vertices have no parent, we represent this with some sentinel value, which we denote as $\perp$.

7.1.4 Reconstructing Separator Decompositions

CCH vertex importance orderings are based on a separator decomposition of the input network. This decomposition is helpful for several CCH algorithms, such as the parallelization of the customization or nearest-neighbour searches. However, black-box partitioners such as FlowCutter or InertialFlowCutter return only the vertex order but provide no easy access to the separator decomposition. Fortunately, we can reconstruct a decomposition efficiently from the elimination tree.

Lemma 7.1. *Let v denote the highest-ranked vertex with more than one child in the elimination tree. The path from v to the elimination tree root is a chain of vertices with precisely one child. The vertices of that path are the top-level separator.*

Proof. Let T_u and T_w be two subtrees rooted at different children of v in the elimination tree. Since we claim that the elimination tree path from v upward is a separator, there must be no edge in the input graph between vertices of these subtrees. Suppose for contradiction there is such an edge between vertices u and w where u is a vertex in T_u and v a vertex in T_w. By construction, all vertices in the subtrees have a lower rank than v. Without loss of generality, we assume $u \prec w$. Because u and w are in distinct subtrees, we know $\texttt{ET}[u] \neq w$ and thus $\texttt{ET}[u] \prec w$. Otherwise, $\texttt{ET}[u]$ would not be u's lowest-ranked upward neighbour. However, due to the construction of the augmented graph, the edge $(\texttt{ET}[u], w)$ must exist, too. This holds inductively for every vertex in the elimination tree path from u to v. However, since $w \prec v$, one of the vertices on this path would have had to have w as its lowest-ranked upward neighbour and parent in the elimination tree. This is a contradiction. □

Thus, we can obtain the top-level separator of a cell as the path from the highest-ranked vertex with more than one child to the root of the cell's elimination subtree. Removing these vertices from the original graph leaves two or more disconnected cells. Each cell contains the vertices of a subtree where the root is a child of v. Recursively applying this idea lets us obtain the whole separator decomposition.

7.2 Customization

For the customization, a weight function ℓ is given, and the corresponding augmented weight function ℓ^+ for the augmented graph is computed. As a CCH augmented graph G^+ usually contains many edges which are irrelevant for a specific weight function ℓ, these can be identified and removed in an additional optional *perfect customization* step to obtain a *minimal augmented graph* G^*. There are up to four steps in the customization:

1. First, ℓ^+ is initialized and, for every edge, set to the corresponding weight in ℓ or ∞ if no such edge exists. This step is called *respecting*.

2. The *basic customization* step is the most critical part and computes the remaining ℓ^+ weights such that queries can be answered correctly. For this, all edges uv are processed in a bottom-up fashion, i.e. by ascending rank of the endpoints. For each edge, *lower triangles* (u, w, v) where $uw, wv \in \mathcal{E}^+$, $w \prec u$ and $w \prec v$ are enumerated and relaxed, i.e. the weight of uv (or vu, respectively) is decreased to the length of the path over w if it is shorter. Now, every edge has the weight of the shortest path between its endpoints which uses only lower-ranked vertices, i.e. $\ell^+(uv) = \text{dist}^{\prec}(u, v)$. This is sufficient for correctness. However, some edges have a greater weight than necessary.

3. In the *perfect customization*, edges uv are processed again but in a top-down fashion while relaxing upper and intermediate triangles. For *upper triangles* (u, w, v), w has greater rank than both u and v. When the rank of w is between u and v, (u, w, v) is an *intermediate triangle*. After these triangles have been relaxed, every edge uv in the augmented graph has the weight of the shortest distance between its endpoints $\text{dist}(u, v)$.

4. Finally, in the *construction* step, the minimal augmented graph G^* is constructed. As shown in [DSW16], edges uv whose weight was improved during the perfect customization, i.e. $\text{dist}^{\prec}(u, v) > \text{dist}(u, v)$, are superfluous and can be removed.

The first two steps are mandatory. Steps three and four only help to accelerate queries.

For the customization, we represent the augmented graph by only $G^{\uparrow}$. Each edge $uv \in \mathcal{E}^{\uparrow}$ also implicitly represents the reverse edge vu, i.e. we only store edges at the lower-ranked endpoint. The weights of the edges are stored in two arrays $\texttt{l}^{\uparrow}$ and $\overleftarrow{\texttt{l}}^{\downarrow}$ accessible by the corresponding edge IDs from $\mathcal{E}^{\uparrow}$, i.e. for $uv \in \mathcal{E}^{\uparrow}$, $\texttt{l}^{\uparrow}[uv] = \ell^{+}(uv)$ and $\overleftarrow{\texttt{l}}^{\downarrow}[uv] = \ell^{+}(vu)$. Nonexistent edges are indicated by the weight ∞. This allows us to represent a directed graph despite the topology being bidirected.

We now focus on the basic customization. The basic description from above still leaves open quite a few critical details. In theory, any edge iteration order which can guarantee that the weights of the two lower edges of each triangle are final before the top edge is processed is sufficient. In practice, the easiest method is to iterate over all vertices u by ascending rank and then process all outgoing edges uv with $u \prec v$. The way triangles are enumerated is critical for the performance. The original CCH publication [DSW16] suggests enumerating lower triangles of an edge uv by performing a coordinated linear sweep over the incoming edges $uw \in \mathcal{E}^{\downarrow}$ of u and $vw \in \mathcal{E}^{\downarrow}$ of v. However, Buchhold et al. [BSW19] noticed that enumerating upper triangles with a coordinated sweep over the outgoing edges in $\mathcal{E}^{\uparrow}$ is faster than enumerating lower triangles. Based on this observation, they suggest an improved basic customization algorithm. Their algorithm also iterates over all edges uv ordered by the lower-ranked endpoint. However, for each edge, the upper triangles (u, w, v) where $w \succ u$ and $w \succ v$ are enumerated. Now, the weights of wv (and vw) are relaxed with the length of the path over u, i.e. the triangle is relaxed as a *lower* triangle.

This method of triangle enumeration is faster because of the distribution of the vertex degrees. The classical approach will iterate for every edge over the downward edges of both endpoints. Thus, the total number of iterations is:

$$\sum_{uv \in \mathcal{E}^{\uparrow}} \left(\deg_{G^{\downarrow}}(u) + \deg_{G^{\downarrow}}(v)\right) = \sum_{v \in \mathcal{V}} \left(\deg_{G^{\uparrow}}(v) \cdot \deg_{G^{\downarrow}}(v) + \deg_{G^{\downarrow}}(v)^2\right)$$

Enumerating upper triangles as suggested in [BSW19] requires sweeping over the upward edges of both endpoints of each edge:

$$\sum_{uv \in \mathcal{E}^{\uparrow}} \left(\deg_{G^{\uparrow}}(u) + \deg_{G^{\uparrow}}(v)\right) = \sum_{v \in \mathcal{V}} \left(\deg_{G^{\uparrow}}(v) \cdot \deg_{G^{\downarrow}}(v) + \deg_{G^{\uparrow}}(v)^2\right)$$

Buchhold et al. observe that the sum of the squared upward degrees $\sum_{v \in V} \deg_{G^{\uparrow}}(v)^2$ is often smaller than the sum of the squared downward degrees $\sum_{v \in \mathcal{V}} \deg_{G^{\downarrow}}(v)^2$ even though the number of edges is the same. This is because the downward degrees are more widely dispersed.

Algorithm 7.1: Basic customization algorithm with batched triangle relaxing.

Data: $\mathtt{l}^{\uparrow}[uv]$: length of $uv \in \mathcal{E}^{\uparrow}$ in the augmented graph.
Data: $\overleftarrow{\mathtt{l}}^{\downarrow}[uv]$: length of $vu \in \mathcal{E}^{\downarrow}$ in the augmented graph.
Data: $\mathtt{ID}[v]$: edge ID of uv where u is the current vertex in the outer loop, initially $\perp$.

```
1  Function BasicCustomization:
2      for all vertices u ∈ V ordered ascending by rank do
3          for all edges uv ∈ E↑ do
4              ID[v] ← uv
5          for all edges uw ∈ E↓ do
6              for all edges wv ∈ E↑ ordered by descending rank of v do
7                  if v ⪯ u then
8                      break
9                  if ID[v] ≠ ⊥ then
10                     l↑[ID[v]] ← min(l↑[ID[v]], ←l↓[uw] + l↑[wv]) // forward
11                     ←l↓[ID[v]] ← min(←l↓[ID[v]], l↑[uw] + ←l↓[wv]) // backward
12         for all edges uv ∈ E↑ do
13             ID[v] ← ⊥
```

7.2.1 Batched Triangle Relaxation

We accelerate the triangle enumeration further by introducing an auxiliary array $\mathtt{ID}$ of size n. The entry $\mathtt{ID}[v]$ will temporarily store the ID of an edge uv for different vertices u throughout the customization. This allows us to replace the coordinated sweeps with a simple iteration over neighbours and direct access to the weights of the respective third edges. We denote this approach as *batched triangle relaxation* because the triangles for all outgoing upward edges of a single vertex are relaxed in batch. The approach pays off because *all* these triangles must be relaxed. The classical coordinated sweep would be the better choice if we were only interested in the triangles of a single edge.

Algorithm 7.1 depicts the batched lower triangle relaxation procedure for the basic customization. Vertices are processed by ascending rank. When processing vertex u, the IDs of its outgoing edges uv are stored in $\mathtt{ID}[v]$. Thus, the weight of this edge can be accessed in $\mathcal{O}(1)$ by the ID of the head vertex v. Now, lower triangles can be enumerated directly. For every downward edge uw, the algorithm iterates over all upward edges wv of w. If $\mathtt{ID}[v]$ contains an entry, a lower triangle was found, and the weights of uv and vu can be relaxed accordingly. After all lower triangles were relaxed, the entries of $\mathtt{ID}$ are reset.

Additionally, the inner loop can be terminated early (see Line 8). For this, the algorithm iterates over the upward edges wv of the lowest vertex w ordered descending by rank. Thus, when $v \preceq u$, no further lower triangles of any upward edge of u can be found.

Note that we need to iterate over the downward edges $\mathcal{E}^{\downarrow}$ of u (see Line 5). Therefore, we also

Algorithm 7.2: Perfect customization algorithm with batched triangle relaxing.

Data: $\mathtt{l}^{\uparrow}[uv]$: length of $uv \in \mathcal{E}^{\uparrow}$ in the augmented graph.
Data: $\overleftarrow{\mathtt{l}}^{\downarrow}[uv]$: length of $vu \in \mathcal{E}^{\downarrow}$ in the augmented graph.
Data: $\mathtt{ID}[v]$: edge ID of uv where u is the current vertex in the outer loop, initially $\perp$.

Function `PerfectCustomization`**:**
  for all vertices $u \in \mathcal{V}$ ordered by descending rank **do**
    for all edges $uv \in \mathcal{E}^{\uparrow}$ **do**
      $\mathtt{ID}[v] \leftarrow uv$
    for all edges $uw \in \mathcal{E}^{\uparrow}$ **do**
      for all edges $wv \in \mathcal{E}^{\uparrow}$ **do**
        if $\mathtt{ID}[v] \neq \perp$ **then**
          `// upper triangle forward`
          $\mathtt{l}^{\uparrow}[uw] \leftarrow \min(\mathtt{l}^{\uparrow}[uw], \mathtt{l}^{\uparrow}[\mathtt{ID}[v]] + \overleftarrow{\mathtt{l}}^{\downarrow}[wv])$
          `// upper triangle backward`
          $\overleftarrow{\mathtt{l}}^{\downarrow}[uw] \leftarrow \min(\overleftarrow{\mathtt{l}}^{\downarrow}[uw], \overleftarrow{\mathtt{l}}^{\downarrow}[\mathtt{ID}[v]] + \mathtt{l}^{\uparrow}[wv])$
          `// intermediate triangle forward`
          $\mathtt{l}^{\uparrow}[\mathtt{ID}[v]] \leftarrow \min(\mathtt{l}^{\uparrow}[\mathtt{ID}[v]], \mathtt{l}^{\uparrow}[uw] + \mathtt{l}^{\uparrow}[wv])$
          `// intermediate triangle backward`
          $\overleftarrow{\mathtt{l}}^{\downarrow}[\mathtt{ID}[v]] \leftarrow \min(\overleftarrow{\mathtt{l}}^{\downarrow}[\mathtt{ID}[v]], \overleftarrow{\mathtt{l}}^{\downarrow}[uw] + \mathtt{l}^{\uparrow}[wv])$
    for all edges $uv \in \mathcal{E}^{\uparrow}$ **do**
      $\mathtt{ID}[v] \leftarrow \perp$

need to store $G^{\downarrow}$. To access the weights of these edges, we store the ID of the corresponding upward edge alongside each edge.

This procedure can also be easily extended to generate unpacking information: Maintain an additional array with the information (for example, the IDs of the lower two edges of the shortest lower triangle) for each edge, and update the information together with the edge weights during relaxation.

The total number of iterations with this approach is at most

$$\sum_{v \in \mathcal{V}} \left(\deg_{G^{\uparrow}}(v)^2 + 2 \cdot \deg_{G^{\uparrow}}(v) \right)$$

where the first term is for the triangle relaxations and the second term for maintaining `ID`. The first term comes from counting the iterations at the lowest vertex of each triangle. With the stopping criterion for the inner loop, the actual number of iterations will be even smaller. As our experiments show, this yields a significant speedup.

Algorithm 7.2 depicts the batched upper and intermediate triangle relaxation approach for the perfect customization. There are a few differences to Algorithm 7.1. First, vertices are processed

top-down, i.e. by descending rank. Second, to enumerate upper triangles, the algorithm iterates over upward edges uw of u, and then over upward edges wv of w. Third, the triangles found this way are relaxed as both upper and intermediate triangles, i.e. both the distances of uv and uw are improved. The total number of iterations for this algorithm is

$$\sum_{v \in V} \left(\deg_{G^\uparrow}(v) \cdot \deg_{G^\downarrow}(v) + 2 \cdot \deg_{G^\uparrow}(v) \right)$$

where the first term is for the relaxations and the second for maintaining the `ID` array.

To identify edges which should later be removed, we use an array with one boolean value per edge, initially `false`. When a weight is improved, the respective byte is set to `true`, which indicates that the edge can be removed later. These values are represented by one byte per value such that every value can be addressed individually. With a compact representation, two threads might need to write two the same address. Unpacking information does not needs to be updated during the perfect customization. Edges with improved weights will be removed during the construction of the reduced graph.

7.2.2 Parallelization

The original CCH publication [DSW16] proposed to parallelize both the basic and perfect customization with a simple loop-based approach by processing edges on the same CH level in parallel, as they are independent[1] from each other. Unfortunately, this approach requires a synchronization step after the completion of each CH level. This is detrimental to load balancing. Buchhold et al. [BSW19] introduced a task-based parallelization approach utilizing the separator decomposition. Each task is responsible for a subgraph G'. Removing the top-level separator in G' decomposes the subgraph into two or more disconnected components. For each component, a new task is spawned. If the size of subgraph G' is below a certain threshold, the task completely processes G' sequentially without spawning subtasks. We use $n/(\eta \cdot c)$ as the threshold as suggested by [BSW19], with c being the number of cores and the tuning parameter η set to 32. During the basic customization, edges in the separator are processed once all child tasks have finished. With this approach, synchronization is still necessary, but the overhead is much smaller than the synchronization per level in the loop-based approach. During the perfect customization, the separators are processed first. Thus, no synchronization is necessary.

Our batched triangle relaxation algorithm can easily be used with the separator-based parallelization. The outer loop in Line 2 must be restricted to the current subgraph. Contrary to the customization algorithm in [BSW19], our variant requires no atomic instructions. Because only weights associated with edges of the current vertex u will be modified, no concurrent modifications can occur. This makes our basic customization even more effective to parallelize.

[1] This is only obvious for the basic customization. However, for the perfect customization, the authors of [DSW16] proved that the algorithm is correct, too, as long as weight updates (but not necessarily comparisons) happen atomically.

Parallelized Reduced Graph Construction. The original CCH publication does not provide details on the construction of the reduced augmented graph, probably because it appears trivial from an algorithmic point of view. All that needs to be done is filter out edges from a graph in adjacency array representation. However, building the reduced augmented graph makes up a significant share of the running time of the customization [BSW19]. Therefore, an efficient parallelization is essential. We propose the following parallelization scheme:

We split the graph into $\kappa \cdot c$ chunks of vertices with consecutive IDs, where c is the number of cores and κ a tuning parameter set to 4 by default. The chunk sizes are chosen such that every chunk contains roughly the same amount of *edges* (not vertices). We find the ID of the first vertex of chunk i by taking the tail of the edge with ID $i \cdot \lceil \frac{m}{\kappa c} \rceil$. Then, we build the reduced graph in three passes over all edges. In the first pass, all chunks are processed in parallel, and the edges that will remain in each chunk are counted. A sequential prefix sum over these sums yields the reduced edge ID range for each chunk. The remaining edges are copied to the new graph in the second pass, processing each chunk in parallel. Edge data consists at the very least of the head vertex of each edge and the weight. In our implementation of CCH, we also maintain arrays with each vertex's tail and the unpacking information. The unpacking information will be temporarily invalid because the referenced edge IDs are from $\mathcal{E}^+$ and not $\mathcal{E}^*$. Fixing this is the third pass's goal but requires an explicit edge ID mapping. We maintain an additional array of size $|\mathcal{E}^+|$ for this mapping. The entries are set during the second pass when an edge is copied to the reduced graph. Finally, in the third pass, we apply this mapping to the unpacking information. In this pass, the chunks are unnecessary, and the unpacking data of each edge can be processed independently.

Unpacking data consisting of the two edge IDs of the corresponding lower triangle offers the fastest unpacking at query time. However, there are alternatives with different trade-offs. The original CCH publication suggests maintaining no unpacking data and instead unpacking edges at query time by enumerating lower triangles. This reduces the customization effort but makes the unpacking somewhat slower. Another option is to store the bottom vertex of the lower triangle to unpack with each edge. This variant makes the edge ID translation in the reduced graph construction superfluous and offers some speedup for the unpacking but not as much as having the edge IDs directly accessible.

7.3 Queries

The basic CH query algorithm can be applied without modification to answer point-to-point shortest path queries. However, the construction of the CCH augmented graph G^+ admits an even simpler query algorithm [DSW16]. As proven in [BCRW16], the set of ancestors of a vertex in the elimination tree is, in fact, equal to the CH search space of this vertex, i.e. the reachable vertices in $G^\uparrow$ and $\overleftarrow{G}^\downarrow$. Since $G^\uparrow$ and $\overleftarrow{G}^\downarrow$ are directed acyclic graphs and the contraction order is a topological ordering on these graphs, traversing the path in the elimination tree from a vertex to the root while relaxing outgoing edges yields shortest distances. Thus, by combining the

distances from an elimination tree walk from s on $G^{\uparrow}$ and t on $\overleftarrow{G}^{\downarrow}$, we can obtain a shortest up-down path with the length of the distance between s and t.

This *elimination tree query* is faster than the classical CH query algorithm because no priority queues are required. However, the downside is that no simple stopping criterion can be applied. The path to the elimination tree root must always be traversed completely. Therefore, short-range queries become unnecessarily slow. Buchhold et al. [BSW19] propose a simple optimization to mitigate this issue: As soon as a tentative total distance μ was found, only relax the outgoing edges of vertex v if $\text{dist}_{G^{\uparrow}}(s, v) < \mu$ (or $\text{dist}_{\overleftarrow{G}^{\downarrow}}(t, v) < \mu$ in the backward search). With this, elimination tree queries are consistently faster than classical CH queries (on CCH) across all distances. Buchhold et al. present another optimization which accelerates queries further. Because the search graphs are acyclic, the distance of a vertex will never be read again after its outgoing edges were relaxed. Thus the stored distance can immediately be reset to ∞. A separate distance resetting step as proposed in [DSW16] becomes unnecessary. This optimization assumes that both directions are interleaved and that the search with the lower-ranked next vertex will always be advanced first.

7.4 Extended Queries

7.4.1 Lazy RPHAST on CCH

In the previous chapter, we introduced the Lazy RPHAST algorithm (see Section 6.1). Similarly to a standard CH query, it can be applied to CCH without modifications. However, we can also design an improved version utilizing the elimination tree.

To compute the distance $\texttt{D}[v]$ of a vertex v, the distances of all upward neighbours must be final. In Algorithm 6.1, these upward neighbour distances are computed recursively. Thus, the search space is explored in a DFS-like fashion and distances are finalized in DFS post order. However, the elimination tree path from a vertex to the root also is a topological order for the search space. This is because the ancestors in the elimination tree contain the entire CH search space [BCRW16]. Therefore, iterating over the vertices on the elimination tree path from the root to v while relaxing outgoing upward edges of each vertex also yields shortest distances. Further, with this approach, when a vertex v has a distance $\texttt{D}[v] \neq \perp$, all ancestors of v must already have their final distance, too. Thus, as soon as the algorithm encounters a vertex with a final distance, the remaining search space is known to have final distances.

We obtain the procedure described Algorithm 7.3, which utilizes the elimination tree. For any vertex which already has a memoized distance $\texttt{D}[v] \neq \perp$, the algorithm immediately returns this distance. Otherwise, the algorithm follows the elimination tree upward until a node with a finalized distance is encountered. The elimination tree can be traversed via the parent array `ET`. The visited nodes are pushed onto a stack `S`. This enables the algorithm to enumerate the vertices in reversed order by popping them from the stack. While popping the nodes, all outgoing upward edges are relaxed. This finalizes the shortest distances for all vertices on the elimination tree path, including the desired query vertex v.

Algorithm 7.3: Elimination tree based Lazy RPHAST algorithm.

Data: $\mathtt{D}^{\downarrow}[v]$: tentative distance from any vertex $v \in \mathcal{V}$ to t as computed by Algorithm 5.1 on $\overleftarrow{G}^{\downarrow}$.
Data: $\mathtt{D}[v]$: memoized distance from any vertex $v \in \mathcal{V}$ to t, shared between invocations.
Data: S: stack with vertices to compute distances, empty initially.

Function `Select(`t`)`:
  Execute Algorithm 5.1 from t on $\overleftarrow{G}^{\downarrow}$, filling $\mathtt{D}^{\downarrow}$
  $\mathtt{D}[v] \leftarrow \bot$ for all $v \in \mathcal{V}$

Function `ComputeAndMemoizeDist(`u`)`:
  `// Determine the vertices` v `for which` $\mathtt{D}[v]$ `needs to be computed`
  $v \leftarrow u$
  while $\mathtt{D}[v] = \bot$ **do**
    Push v onto S
    if $\mathtt{ET}[v] = \bot$ **then**
      break
    $v \leftarrow \mathtt{ET}[v]$
  `// Compute D for those vertices`
  while S not empty **do**
    $v \leftarrow$ pop top element from S
    $\mathtt{D}[v] \leftarrow \mathtt{D}^{\downarrow}[v]$
    for all up-edges $vw \in \mathcal{E}^{\uparrow}$ **do**
      $\mathtt{D}[v] \leftarrow \min(\mathtt{D}[v], \ell^{+}(vw) + \mathtt{D}[w])$
  return $\mathtt{D}[u]$

A natural application of this algorithm is to use it as a potential function for A*. Because this variant is CCH-based, quick updates to the lower bound weights ℓ_{free} become possible. This allows us to obtain much better distance estimates with dynamic routing data such as real-time traffic. Therefore, we can support various extended problem scenarios in conjunction with CCH. We denote A* with a potential realized by Elimination Tree Lazy RPHAST in combination with the low-degree optimizations as *CCH-Potentials*. However, as shown in the next section, the approach is also advantageous as an incremental one-to-many algorithm.

7.4.2 Nearest Neighbor Queries

An essential feature for practical routing applications is answering *point-of-interest* (POI) queries, for example, finding gas stations. Typically, users want a few options close to their current position. This scenario can be formalized as the k-nearest-neighbor problem. Given a graph $G = (\mathcal{V}, \mathcal{E})$ with edge lengths ℓ, a set of targets $\mathcal{T} \subseteq \mathcal{V}$ and a source vertex s, compute a target

subset $\mathcal{T}' \subseteq \mathcal{T}$ with $|\mathcal{T}'| = k$ such that $\text{dist}(s,t) \leq \text{dist}(s,t')$ for and $t \in \mathcal{T}'$ and $t' \in \mathcal{T} \setminus \mathcal{T}'$. For this problem, we consider four phases: Preprocessing and update are the same as before, i.e. G and ℓ are given, respectively. In the additional *selection* phase the targets $\mathcal{T}$ are provided. Finally, in the query phase, s is given, and $\mathcal{T}'$ should be computed as quickly as possible.

In [BW21], an efficient k-nearest-neighbor query algorithm for CCH was introduced. It utilizes the separator decomposition of the network. Here, we present an improved version of this algorithm. The original algorithm performs many point-to-point searches from the same source vertex. We accelerate these searches with our elimination tree-based Lazy RPHAST routine. As our experiments show, this results in speedups of up to an order of magnitude.

The algorithm works as follows: A max-heap of the k closest targets found so far is maintained, which is initially empty. The graph is explored recursively using the separator decomposition. The algorithm begins with the top level cell which is the entire graph. Let $\text{dist}(s,\mathcal{C}) = \min_(v \in \mathcal{C})\text{dist}(s,v)$ denote the shortest distance from from s to any vertex in a cell $\mathcal{C}$. This cell distance determines the order in which the algorithm recursively descends into subcells. It can be obtained efficiently and provides a rough lower bound to the distance to any target in the cell. If this lower bound is greater than the kth-closest distance already found (accessible in the max-heap) or if there are no targets in a cell, the cell can be skipped entirely. When exploring a cell $\mathcal{C}$, first, the distance from s to target vertices in the separator of this cell is checked. If one of these targets is close enough to s, it is inserted into the max-heap, possibly replacing a vertex further away. Then, the distance $\text{dist}(s,\mathcal{C}')$ from s to each subcell $\mathcal{C}'$ is computed. If s is included in this cell, this distance is zero. The subcells $\mathcal{C}'$ are then sorted increasingly by $\text{dist}(s,\mathcal{C}')$. Finally, the algorithm is invoked recursively on each subcell in this order. When a cell contains fewer target vertices than a certain threshold (experimentally determined to be 8), the distances to each target are computed directly; the separator decomposition exploration is stopped at this branch.

Computing distances from s to targets and cells makes up a significant share of the running time of this algorithm. We, therefore, propose to utilize Elimination Tree Lazy RPHAST for this: Initially compute distances from s on $G^{\uparrow}$ with an elimination tree walk. Then, use the `ComputeAndMemoizeDist` function to quickly obtain distances from s to any vertex v on demand while utilizing the already computed distances. To efficiently compute distances to a cell $\mathcal{C}$, we build on observations made in [BW21]. The perimeter $\mathcal{P}(\mathcal{C})$ of a cell is the set of vertices adjacent to a vertex in $\mathcal{C}$: $\mathcal{P}(\mathcal{C}) = \{v \mid uv \in \mathcal{E}^+, u \in \mathcal{C}, v \in \mathcal{V} \setminus \mathcal{C}\}$. The minimum distance from s to any vertex in the perimeter $\min_{v \in \mathcal{P}(\mathcal{C})} \text{dist}(s,v)$ is a very good lower bound on the distance from s to any vertex in $\mathcal{C}$. This lower bound is tight enough and can be computed efficiently: All perimeter vertices must have a higher rank than the cell vertices and lie on the elimination tree path from any cell vertex to the root. Thus, running Algorithm 7.3 once from the lowest-ranked perimeter vertex is sufficient to ensure that the distance from s to all perimeter vertices has been computed. As proven in [BW21], the upward neighbours in G^+ (but not G^*) of the highest-ranked vertex in a cell are exactly the perimeter vertices. Therefore, the minimum distance to any perimeter vertex can be quickly obtained by iterating over the outgoing edges of this vertex and checking the distances computed by Algorithm 7.3.

The algorithm also requires efficient access to the targets and vertices in a cell. For this, one can utilize the fact that the vertices in each cell appear as a consecutive range in the contraction order, and the separator vertices are last in this range. Thus, these vertices can be easily identified by storing the ID of the first vertex, the last vertex and the first separator vertex for each cell. The targets inside a separator or cell can be quickly obtained by performing a binary search in the sorted target list. Sorting the target array makes up the selection phase.

Note that Buchhold et al. [BW21] proposed to fill an auxiliary array during the selection for $\mathcal{O}(1)$ access to the targets. However, filling this array takes $\Theta(n)$ running time. This constitutes a non-negligible overhead in the selection phase, which is problematic in online settings. In contrast, our approach takes $\mathcal{O}(|\mathcal{T}| \cdot \log |\mathcal{T}|)$ which we found significantly faster for all but the largest target sets. Further, we could not observe any measurable overhead from the binary searches compared to the constant-time access.

7.4.3 Alternative Routes

Providing users with multiple good alternative routes to choose from is another critical feature for practical routing applications. To the best of our knowledge, no alternative route algorithms based on CCH have been proposed so far. Some CH-based algorithms exist [ADGW13, Kob15], but it is not immediately clear if these algorithms could be easily adapted to CCH. Therefore, in this section, we propose a simple CCH-based alternative route algorithm based on a variation of the penalty method and CCH-Potentials. The penalty method is an established framework for computing alternative routes [BDGS11, KRS13, PZ13, Kob15]. It works iteratively. The shortest path between s and t is computed in each iteration. This path represents a possible alternative. To avoid this path in future iterations, all edge weights of the path and adjoined edges are penalized, i.e. the edge weights are increased. Shortest paths are computed and penalized until an obtained path becomes too long with respect to the unmodified edge weights. Typically, the found paths are combined into an alternative graph from which alternative routes are extracted in a second step. However, for simplicity and inspired by a variation of the penalty method presented in [ADGW13], we return some of the obtained paths directly as alternative routes. We refer to this approach as *penalty routes*. In the following, we briefly discuss our method's implementation details and penalization configuration.

We compute shortest paths with CCH-Potentials, i.e. we run A* with Lazy RPHAST on CCH as the heuristic. The initial weight function without penalization will typically be travel times adjusted to the current traffic situation. When applying the penalization, we do *not* run the customization. We only run A* with the penalized weights but use the non-penalized distances for the potential function.

Having obtained a path for the current iteration, we check which edges were present in any previous path. If the combined non-penalized weight of the non-shared edges makes up more than 20% of the total weight of the path, we report the path as a viable alternative.

Our penalization scheme follows the configuration presented by Kobitzsch in his dissertation [Kob15, Kob21]. To the best of our knowledge, this is the latest, most thoroughly engineered

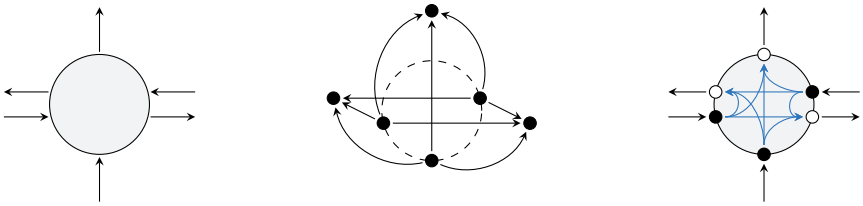


Figure 7.2: Turn representations (from left): simplified model, edge-based model, compact model.

and evaluated iteration of the penalty method. We limit the stretch of alternatives routes, i.e. for any path $P = (s, \dots, t)$ the length $\ell(P)$ must not exceed $(1+\epsilon) \cdot \text{dist}(u, v)$ with $\epsilon = 0.25$. Shortest path edges are penalized with a multiplicative factor of $\psi = 1.1$, i.e. if an edge e was on the shortest path in k iterations, its weight will be $\ell_{\text{pen}}(e) = \ell(e) \cdot \psi^k$ where ℓ_{pen} are the penalized weights. We also penalize edges incident to shortest paths with an asymmetric additive rejoin penalty. An edge uv where u is on a shortest path but v is not will have its weight increased by $\psi_{\text{r}} \cdot \text{dist}(s, t) \cdot \frac{\text{dist}_{\text{pen}}(s,u)}{\text{dist}_{\text{pen}}(s,t)}$. Analogously, an edge uv where v is on a shortest path but u is not will have its weight increased by $\psi_{\text{r}} \cdot \text{dist}(s, t) \cdot \frac{\text{dist}_{\text{pen}}(v,t)}{\text{dist}_{\text{pen}}(s,t)}$. ψ_{r} is set to 0.01. To avoid over-penalization, we count the times a *vertex* was on a shortest path and do not apply penalties when this number exceeds $k_{\max} := \lceil \log_{\psi}(1 + \epsilon) \rceil + 2$. We terminate the algorithm when any of the following conditions becomes true:

- The desired number of alternatives has been obtained.
- All nodes on the current shortest path have been penalized $k_{\max}$ times.
- The shortest path is longer than $(1 + \epsilon) \cdot \text{dist}(s, t)$ with respect to the original weight function.
- The shortest path is longer than $(1 + \epsilon) \cdot \psi + 2\psi_{\text{r}} \cdot \text{dist}(s, t)$ with respect to the penalized weight function.

The last two stopping criteria can be used for pruning in A*. Further, we run the forward and backward search in parallel.

7.4.4 Turn Costs and Restrictions

So far, we have focused on a *simplified model* of the road network where turn costs and restrictions are ignored; see Figure 7.2 (left). Since turn costs can be critical for realistic routing, we now show how to extend CCH to support turn costs efficiently.

Edge-Based Model. The *edge-based model* [Cal61, Win02] expands the network so that road segments become vertices and allowed turns become edges; see Figure 7.2 (middle) for an

example. We have used this model already in the previous chapter; see Section 6.3.3 for the precise construction of the expanded graph. The main advantage of this edge-based model is that most route planning algorithms can be used on it as is, without further modifications.

Compact Model. Turns can alternatively be represented with the *compact model* [GV11, DGPW17]. In this model intersections are kept as vertices, but a $p \times q$ *turn table* T_v is associated with each vertex v, where p and q are the numbers of incoming and outgoing edges, respectively; see Figure 7.2 (right) for an example. The entry $\mathsf{T}_v(i, j)$ represents the cost of the turn from the i-th incoming edge e to the j-th outgoing edge f, i.e., $\mathsf{T}_v(i, j) = \ell_t(e, f)$. The compact model's main advantage is its low space overhead since turn tables can be shared among vertices. The number of distinct turn tables for continental instances such as the road network of Western Europe used in our experiments is in the thousands rather than millions [DGPW17]. However, as shown in [GV11, BWZZ20], (C)CH and the compact model do not work well with each other. The preprocessing algorithms become rather complex and the achieved performance is not competitive. Therefore, we focus on CCH and the edge-based model in the following.

CCH on the Edge-Based Model

CCH can be applied to the edge-based model without modifications. However, preprocessing running times suffer significantly. Therefore, we propose optimizations to reduce the slowdown.

Contraction Order. Obtaining the nested dissection order is the most expensive part of preprocessing. We can apply the same ordering algorithms as for a non-turn CCH without modification to the edge-based graph. We refer to this order as the *edge-based order*. Sadly, this approach is quite slow.

Fortunately, we can also exploit that vertices in G_e are edges in G and compute an edge order on G. Algorithms for obtaining separator decompositions in road networks like InertialFlow [SS15] and InertialFlowCutter [GHUW19] compute separators by finding a small balanced cut and deriving a separator from that cut. However, a cut in G corresponds directly to a separator in G_e. Thus, we compute *cut-based orders* by computing a small balanced cut in G, using the vertices corresponding to the cut edges as the highest-ranked vertices in the contraction order for G_e and recursing on the sides of the cut. We extend InertialFlowCutter with this method.

Infinite Shortcuts. Recall that CCH algorithms do not work on the original directed graph $G = (\mathcal{V}, \mathcal{E})$, but on the corresponding bidirected graph $G' = (\mathcal{V}, \mathcal{E}')$ that is obtained from G by adding all edges $\{wv : vw \in \mathcal{E} \wedge wv \notin \mathcal{E}\}$. This can lead to the insertion of unnecessary shortcuts; see Figure 7.3 for an example. We denote these unnecessary shortcuts as *infinite shortcuts* as the edges in both directions always have weight ∞. Infinite shortcuts can be identified by customizing with the weight function $\ell(e) = 0, e \in \mathcal{E}$. Afterwards, every bidirected edge with weight ∞ in both directions is an infinite shortcut and can be removed. After obtaining the elimination tree, we identify and remove infinite shortcuts in an additional preprocessing step.

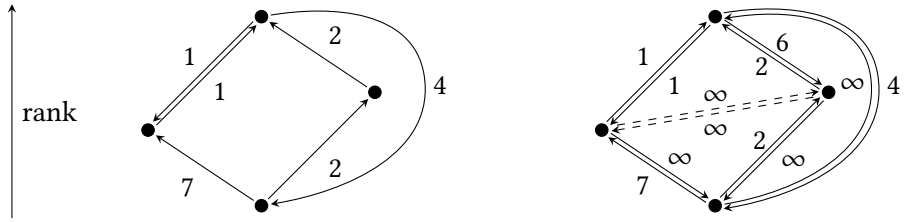


Figure 7.3: Original graph and preprocessing result after the basic customization. The dashed shortcut has always weight ∞ in both directions and can be removed.

Having removed some edges, we loose the guarantee that the perfect customization will set every edge weight to the shortest distance between its endpoints. Due to removed edges, some triangles necessary to find these shortest distances might be missing. In the example depicted in Figure 7.3, the edge with length seven is longer than the shortest path between its endpoints. However, for the perfect customization to find this path, the infinite shortcut is necessary. Fortunately, this does not affect correctness. We will only potentially miss edges to remove. The perfect customization algorithm can still be applied without modification. However, we now do not obtain a minimal augmented graph but only a reduced one. As our experiments show, losing this minimality guarantee has little impact on query running times. Applying the perfect customization still yields worthwhile speedups.

Directed Hierarchies. In the simplified model, many edges have a corresponding reversed edge. This changes in the edge-based model, and the number of edges without a corresponding reversed edge increases. Thus, the customized graph contains many edges with weight ∞ but a finite weight for the reversed edge. Like infinite shortcuts, these edges can be identified by customization with the zero-length function. We remove these edges after obtaining the elimination tree. The result is a *directed hierarchy*. Customization algorithms must now enumerate triangles in both directions separately. However, directed hierarchies contain fewer triangles in total. Therefore, the customization becomes faster. No adjustments are necessary for the query.

Reordering Separator Vertices. The vertices inside a separator can be ordered arbitrarily in a nested dissection order. We exploit this to generate more infinite shortcuts. Separator vertices in G_e correspond to cut edges in G. We order them according to the side on which the tail vertex of the corresponding cut edge lies. For example consider a cut in G with a left and a right side (Figure 7.4). Cut edges going from the left to the ride side (i.e. their respective vertices in G_e) are assigned the lower ranks, and cut edges from right to left are assigned the higher ranks. This way, shortcuts between the lower-ranked vertices (left to right in the example) can never have a finite weight. Any directed path between them must use one of the higher-ranked vertices (to go back from right to left). As shortcuts are assigned the weight of the shortest path through lower-ranked vertices, this will always be ∞, and these shortcuts can be removed later.

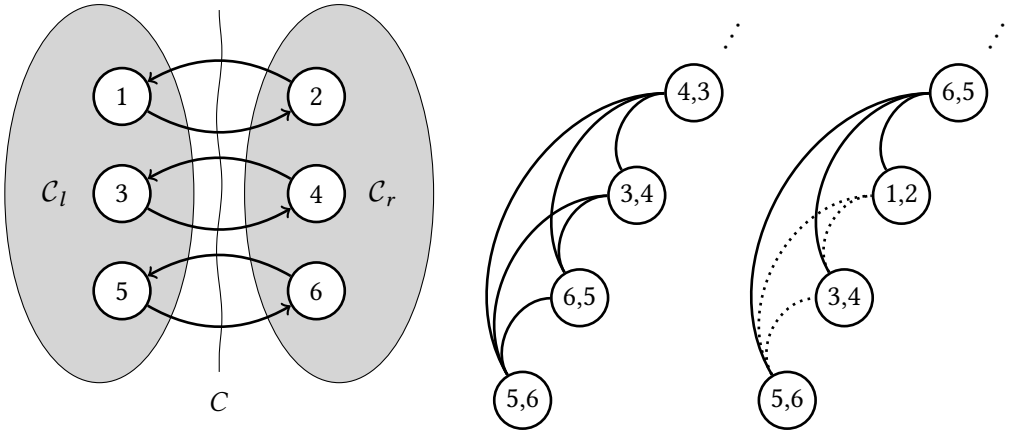


Figure 7.4: On the left is a visualization of a cut in G. In the middle is an arbitrary contraction order which results in no infinite edges after the first four contractions. On the right, the edges in the order are grouped which results in three infinite edges after the first four contractions (the dotted edges).

Turns with CCH-Potentials

It is also possible to sidestep the preprocessing and customization slowdown at the cost of some query performance. The previous chapter shows that CH-Potentials are very effective when applied to turn costs and restrictions. Thus, we can also support turn costs by running bidirectional A* on the expanded graph G_e with Lazy RPHAST on G^* as the potential function.

7.5 Evaluation

Environment. Our primary benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 64 GiB of DDR3-1600 RAM and two Intel Xeon E5-2670 CPUs, each of which has eight cores clocked at 2.6 Ghz and 8 × 64 KiB of L1, 8 × 256 KiB of L2, and 20 MiB of shared L3 cache. We use this machine to ensure the comparability of our results to previous works [DSW16, BSW19] which were also evaluated on this computer. Additionally, this machine allows a rough comparison with the results on CRP presented in [DGPW17].[2] Since the machine is already

[2]According to the comparison methodology from [Bas+16] (see `https://i11www.iti.kit.edu/~pajor/survey/`), the machine used in [DGPW17] (SPA-2) is about 20% slower than ours (compute11). We reran the benchmark for our machine and obtained a score of 38914 ms, which is somewhat slower than the previously reported 36582 ms. This is likely due to the mitigations for side-channel attacks utilizing speculative execution such as Meltdown and Spectre. SPA-2 would only be about 12% slower than our machine compared to this updated score. Generally, these scaling factors have to be interpreted very carefully. They are obtained from one-to-all Dijkstra searches on continental-sized road networks. This benchmark heavily emphasizes memory bandwidth while neglecting other critical factors such as CPU frequency, cache size and cache speed. Also, note that the authors of [DGPW17] even used our machine to evaluate their turn-aware CH implementation, stating that it achieves

almost a decade old at the time of writing, we expect our results to be a reasonably conservative approximation of the performance that can be expected when deploying these algorithms today.

We implemented our algorithms in Rust[3] and compiled them with `rustc 1.64.0-nightly (830880640 2022-06-28)` in the release profile with the `target-cpu=native` option. For the computation of contraction orders, we use InertialFlowCutter (IFC)[4] [GHUW19]. As shown in the extensive evaluation in [GHUW19], IFC currently achieves the best contraction order quality while taking only about two times as long as InertialFlow [SS15], the fastest approach with good quality. We extend IFC with the computation of cut-based orders and the reordering of separator vertices. These extensions were published as a pull request on GitHub[5] and have since been merged into the project. InertialFlowCutter is written in C++ and were compiled with GCC 10.3.0 using optimization level 3.

Inputs. We use our main benchmark instances for time-independent route planning: DIMACS Europe and OSM Germany. See Section 4.2.1 for a description and discussion of these networks. We use synthetic turn costs of 100 s for U-Turns and free turns otherwise for DIMACS Europe as suggested in [DGPW17]. For OSM Germany, we extract real-world turn restrictions from the OSM data. Additionally, we include a city-scale network of the Stuttgart region provided by PTV. This network includes real-world production-grade turn cost and restriction data. It has roughly 110 k vertices and 252 k edges, which makes it about two orders of magnitude smaller than the Germany and Europe instances. Our evaluation in [BWZZ20] included additional city-scale networks. Reproduced and extended results for these instances can be found in Appendix A.4.

Methodology. For the computation of the contraction order, we perform the partitioning ten times and report the average running time. Contraction and customization running times are obtained as averages over 100 runs. For the queries, we perform 1 000 000 point-to-point queries where both source and target are vertices drawn uniformly at random. We utilize parallelization for all phases except the queries.

7.5.1 CCH Performance

Table 7.1 depicts running times of the metric-independent preprocessing. The ordering, performed by IFC, is two orders of magnitude slower than the contraction (with chordal completion) and thus dominates the running time. Still, computing the contraction order of a continental-sized network takes only about six minutes, thanks to the efficient parallelization of IFC. This is faster than the preprocessing time of a classical non-customizable CH [GSSV12].[6] We only

"comparable" performance. We conclude that our machine and SPA-2 yield running times in the same order of magnitude and that our machine is probably, in most cases, slightly faster.

[3] The code for this paper and all experiments is available at `https://github.com/kit-algo/cchpp`

[4] `https://github.com/kit-algo/InertialFlowCutter`

[5] `https://github.com/kit-algo/InertialFlowCutter/pull/6`

[6] The CH preprocessing is typically performed sequentially, which makes this a somewhat unfair comparison. Parallelization approaches for CH preprocessing have been described in the context of time-dependent CH [Vet09,

Table 7.1: Running time in seconds of the metric-independent preprocessing algorithms. Our contraction orders were computed with IFC. For contraction, we compare our own chordal completion algorithm against the contraction graph approach and the naive baseline as reported in [DSW16]. Our results and the numbers from [DSW16] were obtained on the same machine. The total running time includes the ordering, the contraction with our chordal completion approach and additionally the reconstruction of the elimination tree and separator decomposition, modifying the order into an DFS post order and all other setup operations.

	Ordering	Contraction			Total
		Chordal completion	Contraction graph [DSW16]	Dyn. adjacency array [DSW16]	
Stuttgart	0.8	0.0	–	–	0.9
Germany	203.9	1.3	–	–	222.9
Europe	341.2	1.6	15.5	305.8	361.1

have running times of competing contraction algorithms for Europe[7]. Still, the speedups are so significant that we can safely conclude that the chordal completion algorithm is the best approach for the contraction. Chordal completion is two orders of magnitude faster than the naive baseline, one order of magnitude faster than the engineered contraction graph approach, and also much simpler to implement.[8] On Stuttgart, the running time is in the single-digit milliseconds. In practice, the running time of the chordal completion is so fast that it disappears behind memory allocation for the graph data, reconstruction of the separator decomposition and other setup/management operations.

In Table 7.2, we investigate the performance of the customization algorithms on Stuttgart and Europe. The results for Germany can be found in Table A.1 in the appendix. Considering the total running time, we observe that our batched triangle relaxation-based customization in the sequential case is roughly two times faster than the results observed in [BSW19] and about four times faster than the baseline [DSW16]. With full parallelization with 16 threads, the picture remains similar on Europe. However, on the smaller Stuttgart instance, the difference to the approach from [BSW19] becomes much smaller. Interestingly, the performance differences are fueled primarily by the construction step, where our approach is sequentially four to five times faster than [BSW19]. One important reason is that the batched triangle relaxation allows us to record unpacking information during the basic customization without any synchronization

BGSV13]. However, these approaches do not scale very well in the classical setting. Further, to the best of our knowledge, there is neither a publication on parallelized CH preprocessing in the time-independent case nor any open source implementation.

[7] Note that the orders used in [DSW16] were obtained with KaHiP, which finds slightly worse orders than IFC. However, according to [GHUW19], the advantage for our algorithms from this should be at most 10%.

[8] See `https://github.com/RoutingKit/RoutingKit/pull/75/commits/16de474b2c3` where we replace the contraction graph approach in RoutingKit with the chordal completion algorithm.

Table 7.2: Running times by number of threads of different steps of the customization on Stuttgart and Europe in comparison with the baseline results reported in [DSW16] and the improvements proposed in [BSW19]. Our results and the numbers from [DSW16, BSW19] were obtained on the same machine. Note that the orders used in [DSW16, BSW19] were obtained with KaHiP and InertialFlow, respectively which find slightly worse orders than IFC. However, according to [GHUW19], the advantage for our algorithms from this should be at most 10%.

		Stuttgart [ms]				Europe [s]			
Impl	Threads	Basic	Perfect	Construct	Total	Basic	Perfect	Construct	Total
[DSW16]	1					10.88	22.02	≈ 9.39	≈ 42.35
	16					5.47		≈ 9.39	≈ 14.86
[BSW19]	1	20.51	20.77	48.64	89.93	5.60	6.48	9.39	21.47
	16	4.91	4.41	4.35	13.66	1.11	0.63	0.80	2.54
[ours]	1	19.85	18.95	10.33	49.13	4.09	4.72	1.95	10.76
	2	11.34	9.90	5.69	26.93	2.05	2.42	0.99	5.46
	4	7.39	5.65	3.57	16.61	1.22	1.26	0.54	3.02
	8	5.70	3.51	2.46	11.67	0.81	0.69	0.35	1.86
	16	6.25	3.10	2.59	11.94	0.58	0.37	0.30	1.25

issues. Thus, in contrast to [BSW19], we do not need to enumerate lower triangles while constructing the minimal augmented graph. We observe that our algorithms utilize additional threads reasonably well, as long as the instances are sufficiently large. On Europe, the total speedup with eight threads is 5.8 and 8.6 with 16 threads. In contrast, on Stuttgart, our best achieved speedup is 4.2 with eight threads. Adding more threads even starts to degrade the performance gradually. In terms of absolute numbers, our approach enables customization of continental-sized instances in about 10 s sequentially and a little more than a second fully parallelized. These improvements finally bring CCH customization running times into a similar range to CRP customization running time (10.55 s sequentially, 1.05 s with 12 cores [DGPW17]). CRP customization times can even be outperformed when only applying the basic customization.

Table 7.3 depicts elimination tree query running times and search space statistics for different networks and weight functions. As shown in [BSW19], elimination tree queries with all optimizations proposed in [BSW19] outperform Dijkstra-based CCH queries across all query distances. Thus, we focus on elimination tree queries and do not evaluate Dijkstra-based queries.

We observe that with only the basic customization, the number of relaxed edges and the distance query running times are very robust against different weight functions. The only changes are due to the pruning criterion [BSW19] which sometimes allows skipping the relaxation of some edges. However, as we use random, i.e. primarily long-range queries for this experiment, this happens only seldomly and has little influence on the results. Despite Germany being a slightly smaller graph, queries take somewhat longer (around 440 μs compared to around 300 μs

Table 7.3: Search space statistics and running times for elimination tree queries on different graphs and weight functions. We evaluate queries on G^+ with the basic customization and on G^* after performing the perfect customization. The number of visited vertices remains the same because elimination tree queries always traverse the full path to the root. The number of edges indicates the combined number of edges relaxed in both directions. The final column contains the number of vertices on the unpacked shortest path. All numbers are averages over 1 000 000 random queries.

		Search space			Running time [μs]				# Path
		Vertices	Edges		Distance		Path		Vertices
			G^+	G^*	G^+	G^*	G^+	G^*	
Stuttgart	Travel time	216.4	8 846.5	3 837.1	22.9	14.5	5.6	5.4	185.9
	Geo distance	216.4	8 888.6	3 303.3	22.2	13.0	4.4	4.2	149.8
Germany	Travel time	1 277.9	274 113.1	78 474.2	442.0	163.2	234.6	134.4	4 681.0
	Heavy traffic	1 277.9	274 478.3	85 298.9	435.0	174.2	241.7	179.5	5 363.4
	Geo distance	1 277.9	274 788.5	131 985.4	438.3	246.3	383.4	336.2	6 174.7
Europe	Travel time	1 041.2	186 006.5	69 312.9	300.2	137.5	95.9	71.3	1 389.8
	Geo distance	1 041.2	185 701.3	92 616.8	303.3	177.9	281.3	252.9	3 158.9

on Europe). This is because the greater modelling detail of the OSM-based Germany instance corresponds to the number of vertices on the obtained paths and the search space size. Path unpacking times also correlate with the number of vertices on the result paths, are in the same order of magnitude as the distance computation times and are often slightly faster.

Running the perfect customization improves query performance significantly in all aspects. The number of relaxed edges is at least halved and the query running times are also roughly halved. Path unpacking times also improve, but not as much. This is because, for path unpacking, the work to perform remains the same, and only the cache locality improves. These improvements are more significant for weight functions with a more pronounced hierarchy. The different weight functions on Germany illustrate this clearly. Stuttgart behaves differently because it is a city network that is much less hierarchical.

Note that our path unpacking times are significantly faster than the numbers reported in [DSW16]. There, an unpacking time of 253 μs is reported for Europe with travel times and 524 μs with geo distances (without perfect customization). Our unpacking times are faster because we store explicit edge unpacking information with each edge while the implementation of [DSW16] enumerates lower triangles for unpacking. However, maintaining this unpacking information during the customization costs some performance, specifically in the G^* construction step. Thus, there is a trade-off between path unpacking times and customization times. If path unpacking times are not critical and a slowdown of a factor of two is tolerable, the customization times could be accelerated further.

Table 7.4: Running times for Dijkstra's algorithm, CH, CRP and CCH on Europe with different weight functions. Preprocessing and customization were executed in parallel, queries sequentially. For CH and CRP we list unscaled results as reported in [DGPW17].

	Travel time			Geo distance		
	Prepro. [s]	Custom. [s]	Queries [ms]	Prepro. [s]	Custom. [s]	Queries [ms]
Dijkstra	–	–	2 359.14	–	–	1 972.93
CH [DGPW17]	109	–	0.11	726	–	0.87
CRP [DGPW17]	654	1.05	1.65	654	1.04	1.91
CCH G^+	367	0.58	0.30	367	0.58	0.30
CCH G^*		1.25	0.14		1.25	0.18

Table 7.4 depicts our results compared to other related routing algorithms on the most prominent research benchmark instance, the DIMACS Europe graph. Dijkstra's algorithm, the non-accelerated baseline, requires no preprocessing but has prohibitively slow query running times. CH [GSSV12], the non-customizable predecessor and foundation to CCH, achieves fast queries on travel times. In this case, the (parallelized) preprocessing is 3.4 times faster than the CCH preprocessing, and queries are also marginally faster. However, CH is not robust against weight functions with a weaker hierarchy. Both preprocessing and query times degrade significantly when applied to a geo distance weight function. In contrast, CRP [DGPW17] is very robust against different weight functions and can introduce arbitrary weight functions with a customization taking only around a second. However, queries are somewhat slower than CH queries. With CCH, we achieve robustness against different weight functions while retaining most of the CH query performance. The optimizations proposed in this work and [BSW19] accelerate the CCH customization such that the basic customization is slightly faster than the CRP customization. Queries with only the basic customization are around five times faster than CRP queries. For even faster queries, the perfect customization can be applied. Then, the total customization time is marginally slower than CRP, but queries are an order of magnitude faster and roughly as fast as classical CH queries. Nevertheless, CRP still has some advantages over CCH. First, the space required per customized weight function is smaller for CRP. Second, CRP has been augmented to more extended problem settings and so far appeared more flexible. In the following, we will investigate how CCH performs in these extended problem settings.

7.5.2 Lazy RPHAST

First, we investigate the performance of our elimination tree-based Lazy RPHAST realization. For this, we pick 100 centre vertices uniformly at random. From each of these, we obtain a ball of vertices $\mathcal{B}$ by running Dijkstra's algorithm until the desired number of vertices have

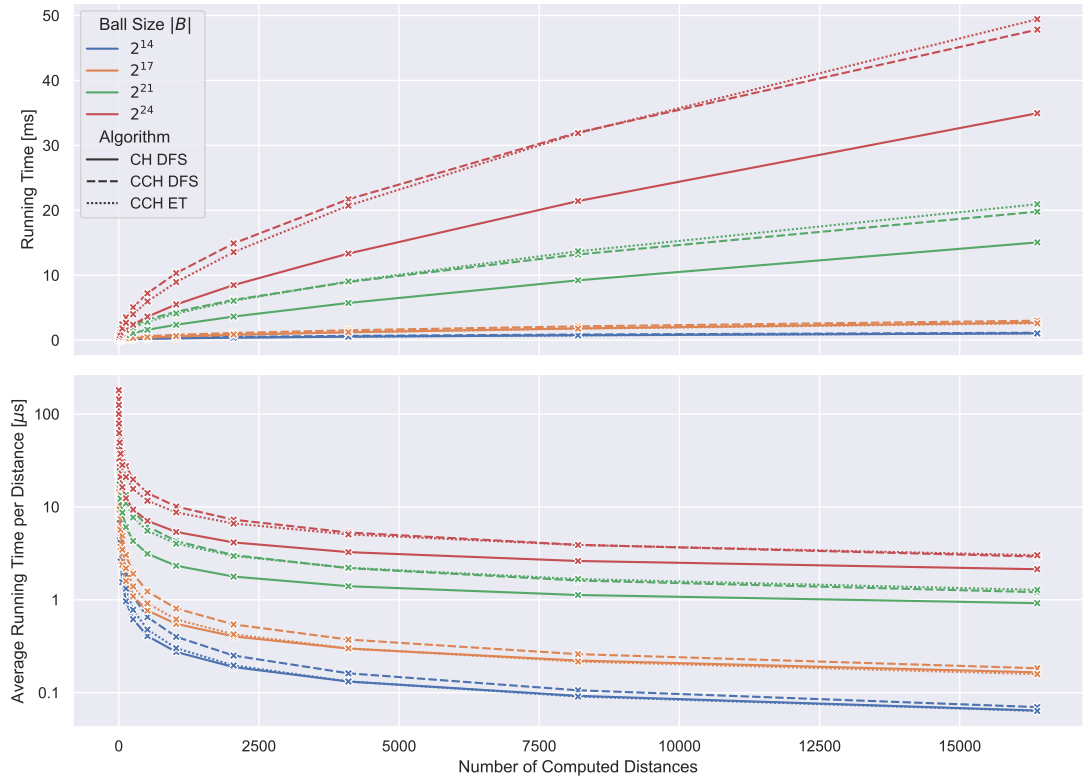


Figure 7.5: Average running times of Lazy RPHAST on CH and CCH while incrementally querying $|S| = 2^{14}$ sources from a ball of varying size $|\mathcal{B}|$ on Europe, excluding selection times. The upper figure contains the total elapsed running time. The lower figure contains the averaged running time per source, i.e. y/x from the upper figure. Note the different y-axis scales and units.

been settled. We then pick 2^{14} source and 100 target vertices uniformly at random from $\mathcal{B}$ and incrementally compute the distances from all sources to each target. Figure 7.5 depicts the average running times of the incrementally queried distances on Europe. We also performed the experiment on Germany. The results are reported in Figure A.1 in the appendix and give the same overall picture.

Lazy RPHAST on CCH is somewhat slower than on CH. This is because CCH has a denser search space than CH. On large ball sizes, the slowdown goes up to a factor of 1.5. However, on smaller ball sizes, the slowdown becomes less significant. The elimination tree-based variant even becomes marginally faster than the CH variant on the two smallest ball sizes. On CCH, we observe a minor advantage for the elimination tree-based realization due to the more efficient implementation. Recall that for the DFS-based variant, all upward neighbours need to be checked to determine if the distance of a vertex can be computed. In contrast, with the elimination

tree variant, it is sufficient to check the parent in the elimination tree. However, there is one exception where the elimination tree variant is less efficient: When $\mathcal{B}$ is sufficiently large, and many sources are queried, the DFS-based variant becomes marginally faster. This is because the path to the elimination tree root might contain vertices which are not reachable in the CCH search space because some edges were removed during the perfect customization. We confirmed this experimentally by running the algorithms without perfect customization on G^+. In this case, the DFS traverses the same vertices as the elimination tree-based variant, and the effect disappears. Usually, however, the number of vertices unnecessarily visited by the elimination tree variant is small compared to the total work. It is thus more than compensated by the more efficient implementation. We, therefore, conclude that the elimination tree-based Lazy RPHAST variant is the better default choice on CCH. Regarding absolute numbers, the average time per distance to compute starts in a similar range as standard (C)CH (around 0.1 ms) queries and quickly becomes faster as more distances are queried. When computing more than 2^{10} distance to the same target on a small ball, the average time per distance even becomes faster than HL queries (roughly 0.5 μs). We conclude that applying Lazy RPHAST on CCH allows us to carry over all the results from the previous chapter to a customizable setting at the cost of a minor slowdown when the targets are distributed over large parts of the graph. We confirmed this by repeating the experiments from the previous chapter with CCH-Potentials. These results are reported in Appendix B.

7.5.3 Point-of-Interest Queries

We now evaluate CCH-based algorithms to answer POI queries efficiently. Figure 7.6 depicts mean running times for finding the four closest targets from a set of varying size with the different algorithms. We drew 100 target sets uniformly at random, performed the selection, and executed nearest neighbour queries from 100 source vertices drawn uniformly at random for each target set. Non-uniform target distributions are evaluated in Section A.3 in the Appendix. There, we also present a direct comparison between the original separator-based algorithm and our optimized version. Here, we focus on evaluating which CCH-based algorithm is most suitable in different scenarios.

We observe that the performance of the baseline, Dijkstra's algorithm, strongly depends on the number of POIs. Since targets are drawn uniformly at random, having more targets corresponds to the closest targets being much closer. With 2^{11} or more targets, Dijkstra's algorithm achieves interactive query times and may present a feasible, practical option. The selection phase for Dijkstra's algorithm only consists of marking the targets in a bitvector so it can be quickly determined at query time if a settled vertex is a target. Thus, it is extremely fast. Lazy RPHAST, in contrast, does not have a selection phase and computes distances to all targets without any stopping criterion. Thus, it is fast for small target sets. The bucket query approach [Kno+07] (here called BCCH; see Section 5.4.2) has the fastest queries (well below 100 μs) across all target set sizes. Further, query times profit from larger target sets. However, the selection phase is relatively slow and goes into the minutes for large target sets. Thus, the

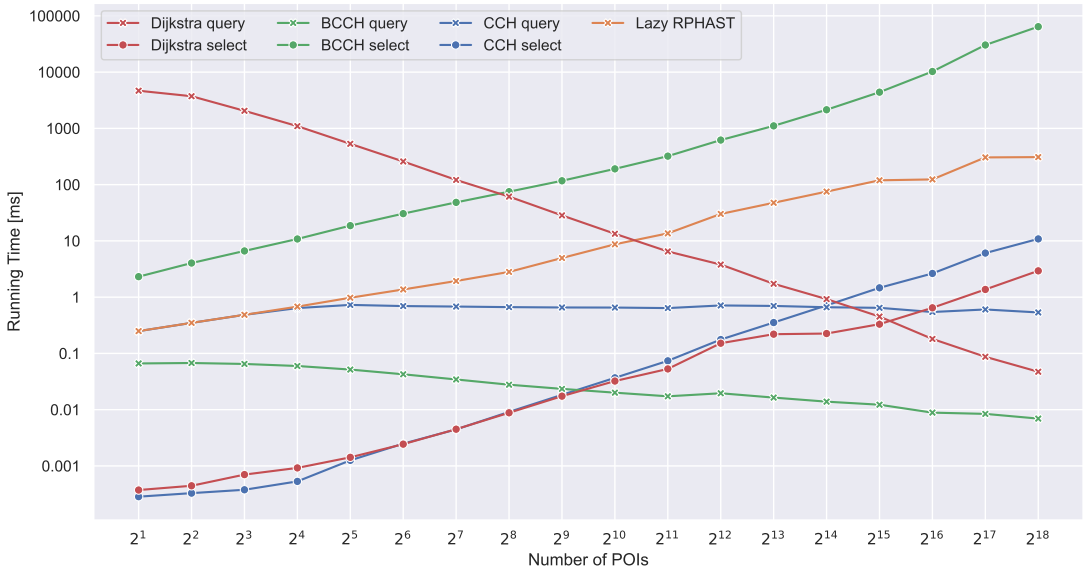


Figure 7.6: Average running times for different nearest-neighbor algorithms on Europe to find the $k = 4$ closest targets from a POI set of varying size.

usefulness of BCCH depends on the scenario. It is the best choice in an offline scenario where targets are known in advance. In an online scenario where the targets are part of the query (for example, a user types "burger restaurants" in the search field of his maps application), BCCH is not feasible. Finally, the separator-based algorithm presents a very robust trade-off. Selection times (consisting of sorting the targets by CCH rank) are barely slower than the selection for Dijkstra's algorithm. Query times are consistently below 1 ms, which is more than sufficient for typical map applications. Therefore, the separator-based algorithm is the strongest contender in online scenarios. Even though these query times are an order of magnitude slower than BCCH, the absolute times are so fast that the algorithm is also a reasonable choice in offline scenarios. Note that our optimizations are crucial in making the separator-based algorithm competitive for the online scenario. Without them, the selection takes 10-20 ms [BW21]. Thus, the separator-based algorithm would be dominated by Lazy RPHAST for few targets and Dijkstra's algorithm for many targets.

7.5.4 Alternative Routes

Next, we evaluate the performance of finding alternative routes with CCH-Potentials and our penalty routes method. Table 7.5 depicts the results. We observe excellent success rates and sharing values. On Europe, we can successfully obtain four additional routes for roughly 60% of the queries. Further, for each route, almost half of it is not shared with any other route. In

Table 7.5: Performance of the penalty routes method with CCH-Potentials. We report results on each graph for finding up to 4 alternative routes. The success rate indicates the share of queries for which the desired number of alternatives could be found. The running time states the total running time for the respective number of alternatives. The iterations column denotes the number of shortest path computations on the penalized graph in addition to the initial shortest path search. Sharing indicates the distance of the current alternative shared with any previous path, including the initial shortest path.

	# Alt.	Success rate [%]	Running time [ms]	# Iterations	Sharing [%]
Stuttgart	1	82.9	1.0	2.2	41.0
	2	63.9	2.4	4.1	45.9
	3	48.0	4.0	5.8	51.2
	4	34.6	6.0	7.4	54.5
Germany	1	95.7	97.4	1.5	34.4
	2	87.1	294.0	3.0	43.3
	3	71.3	595.2	4.3	49.2
	4	53.6	973.5	5.4	52.9
Europe	1	97.3	213.4	1.4	41.9
	2	91.1	548.8	2.7	47.8
	3	77.8	996.8	4.0	52.1
	4	58.7	1 473.9	5.1	55.1

contrast, the CRP-based approach presented in [DGPW17] even finds third alternative routes with only a success rate of 40% and fourth routes for only about 20% of the queries. Germany exhibits similar results. On Stuttgart, we achieve slightly worse results. This is expected since finding alternative routes becomes harder as routes become shorter [Kob15]. While our approach is very competitive in quality, it is relatively slow. Even for the first alternative, running times are only interactive on the Stuttgart instance. Luckily, short-range queries make up the bulk of queries to be answered in practical applications [DSS18]. Therefore, our approach will be feasible in many practical cases.

7.5.5 Turn Costs

We evaluate the impact of incorporating turn costs into CCH. Table 7.6 depicts the results. The simplest way to integrate turn costs is to use CCH-Potentials, i.e. use Lazy RPHAST on CCH without turns as an A* potential. In this case, queries are a bit more than an order of magnitude slower than non-turn CCH but still in the single-digit milliseconds. This is fast enough for interactive applications. The advantage of this approach is that preprocessing and customization remain unchanged.

In contrast, the fastest queries can be achieved when applying preprocessing and customiza-

Table 7.6: Performance of different CCH variants and optimizations to support turn costs. We report the number of directed edges in the augmented graph and running times. Directed hierarchies imply the removal of infinite shortcuts. Reordering separator vertices builds on both directed hierarchies and the removal of infinite shortcuts. All these three variants used a cut order. Preprocessing and customization were executed in parallel, queries sequentially.

		CCH Edges	Prepro.	Customization [ms]		Query [μs]	
		[$\cdot 10^3$]	[s]	Basic	Perfect	Basic	Perfect
Stuttgart	No turns	724.2	1.0	6.4	6.1	22.0	13.7
	CCH-Pot.	724.2	1.0	6.4	6.1	–	441.0
	Naive exp.	3 214.9	3.6	18.0	23.4	55.7	25.3
	Cut order	3 359.7	1.5	17.4	22.5	54.3	25.8
	Infinity	2 874.6	1.6	16.5	20.1	53.4	26.0
	Directed	1 550.5	1.6	13.2	16.2	40.0	26.4
	Reorder	1 440.8	1.6	11.1	13.0	28.9	25.4
Germany	No turns	93 443.1	220.3	552.1	541.4	426.9	155.4
	CCH-Pot.	93 443.1	220.3	552.1	541.4	–	1 659.5
	Naive exp.	440 812.8	2 087.7	2 295.5	3 131.0	1 221.3	289.4
	Cut order	467 191.1	371.1	2 733.2	3 400.6	1 427.9	361.0
	Infinity	388 201.7	390.3	2 658.8	2 858.1	1 438.6	365.0
	Directed	206 970.0	391.7	1 760.9	2 541.8	938.8	371.4
	Reorder	190 597.1	391.4	1 360.7	1 899.2	647.6	378.0
Europe	No turns	117 727.5	367.0	587.6	666.5	297.0	135.1
	CCH-Pot.	117 727.5	367.0	587.6	666.5	–	2 691.9
	Naive exp.	692 995.8	3 817.5	3 169.9	4 629.9	908.3	247.7
	Cut order	737 433.4	407.5	3 210.3	4 670.8	928.8	264.8
	Infinity	651 921.7	435.2	3 257.5	4 451.7	951.8	265.0
	Directed	363 663.3	434.6	2 184.1	3 365.7	618.1	264.8
	Reorder	334 755.9	436.7	1 819.9	2 663.0	453.1	295.8

tion without any modification to the expanded graph (see the "Naive exp." rows). With the perfect customization, queries are only around a factor of two slower than queries without turn costs. However, this comes at the cost of a significant slowdown for both preprocessing phases of up to 12. Our optimizations from Section 7.4.4 help to reduce this slowdown without sacrificing as much query performance as CCH-Potentials. Computing cut orders on the original graph rather than a vertex separator order on the expanded graph reduces the preprocessing slowdown from an order of magnitude to at most 1.7. The resulting order is marginally worse, which leads to around 5% more shortcuts and roughly corresponding slowdowns in customization and query running times. The loss in quality is likely due to certain optimizations in InertialFlowCutter for

optimal vertex orders for specific subclasses of graphs which we did not implement for cut-based orders. We expect that implementing them would close the gap in quality between edge-based and cut-based orders. The remaining optimizations help accelerate the customization further by removing edges that can be guaranteed as unnecessary as part of the preprocessing. Identifying these edges costs extra preprocessing time but computing the order is still the dominant factor. All optimizations combined roughly achieve a speedup of 1.8 for the customization. Removing undirected infinite shortcuts alone yields only minor improvements. Combining this with directed hierarchies and removing all directed infinite shortcuts has a much more significant impact. This impact can be further amplified by reordering separator vertices, which produces even more infinite shortcuts. It is noteworthy that even though our optimizations primarily aim for the customization running time, we also achieve a significant speedup for query running times on G^+. In that case, removing infinite edges also reduces the number of edges in the query search space. However, with the perfect customization and queries on G^*, we observe the opposite: Each optimization is slightly detrimental to query running times. This is because removing more edges as part of the preprocessing leads to the perfect customization missing more and more edges to remove. Thus, the search space on G^* becomes larger with each additional optimization. Nevertheless, the perfect customization still yields significant query speedups on the larger graphs of up to 1.7.

See Appendix A.4 for results on the remaining instances evaluated in [BWZZ20] and sequential customization running times

Comparison with Related work. Table 7.7 summarizes our results on turn costs and depicts them in comparison to running times achieved by competing approaches as reported in [DGPW17]. The experiments were performed on the publicly available Europe instance, the only instance considered in related work. We observe that incorporating turns impacts all algorithms except CRP significantly. Dijkstra becomes at least 2.5 times slower. CH queries remain comparatively fast (at least on the edge-based model), but preprocessing slows down by more than an order of magnitude. The CRP non-turn variant is realized as free turns in the compact model, which explains why incorporating turns leaves the performance unaffected. While CCH without turns achieves faster running times than CRP in all phases, without our modifications, it is outperformed by CRP on graphs with turns. However, when using cut-based orders and all optimizations, CCH again outperforms CRP in all phases except the customization. Regarding customization times, CCH is slower by a factor of 1.6 or more, depending on the configuration. We do not list CCH on the compact model here, as it was shown to be outperformed by the optimized edge-based variant in all phases [BWZZ20]. Note that both the CRP and CCH customization times can be further decreased by two related techniques known as microcode [DW13] (for CRP) and triangle preprocessing [DSW16] (for CCH). However, both techniques require significantly more space, and we choose not to use them to keep the space requirement low. We conclude that while CCH may not be as robust against turn costs as CRP, achieving competitive performance with CCH with turn costs is possible.

Table 7.7: Performance of Dijkstra, CH, CRP and CCH in the compact model, in the edge-based model as is and with our optimizations (Edge-based*) on Europe with and without turns. Preprocessing and customization were executed in parallel, queries sequentially. For CH and CRP, we list unscaled results as reported in [DGPW17].

	No turns			Turns			
	Prepro. [s]	Custom. [s]	Queries [ms]	Repr.	Prepro. [s]	Custom. [s]	Queries [ms]
Dijkstra	-	-	2 359.14	Edge-based	–	–	6 225.60
				Compact	–	–	12 699.32
CH [DGPW17]	109	-	0.11	Edge-based	1 392	–	0.19
				Compact	1 753	–	2.27
CRP [DGPW17]	654	1.05	1.65	Compact	654	1.10	1.67
CCH G^+	367	0.59	0.30	Edge-based	3 817	3.17	0.91
				Edge-based*	437	1.82	0.45
CCH G^*	367	1.25	0.14	Edge-based	3 817	7.80	0.25
				Edge-based*	437	4.48	0.30
CCH-Pot.				Edge-based	367	1.25	2.69

7.6 Conclusion

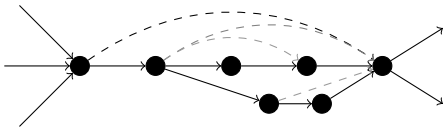
In this chapter, we reviewed and improved the state of the art on Customizable Contraction Hierarchies. Our algorithmic contributions include a novel contraction algorithm, the batched triangle enumeration method for faster customization, an elimination tree-based Lazy RPHAST variant, various improvements for the separator-based CCH nearest neighbour search, and a simple alternative routes algorithm for CCH, and a set of optimizations for CCH on turn-expanded graphs. Further, we provide an extensive experimental evaluation which demonstrates that CCH can be used to build a comprehensive and competitive routing framework. For future work, it would be very interesting if the CCH separator decomposition can be utilized to efficiently obtain *via-vertices* [DGPW17]. This would enable additional algorithmic extensions. Further, we would like to apply CCH-Potentials to more extended dynamic problem settings.

8 Space-Efficient, Fast and Exact Routing in Time-Dependent Road Networks

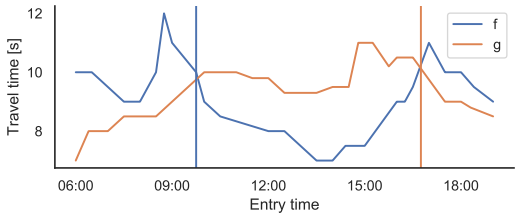
Shortcut edges are a crucial ingredient for many speedup techniques [BD09, Bau+10, GSSV12, Bas+16, DSW16]. Recall that shortcuts are additional edges introduced during preprocessing, which bypass parts of the input graph such as in Figure 8.1a. The weight of a shortcut is set to the length of the shortest path between its endpoints. When computing shortest paths, only few shortcuts are explored instead of many edges in the input graph. The path represented by a shortcut can be obtained lazily, for example by running local Dijkstra searches [DGPW17], or by iterating over possible middle vertices when shortcuts always represent two other (shortcut) edges [GSSV12, DSW16].

This approach has been extended to the time-dependent setting [BGSV13, BDPW16]. In this case, shortcuts are no longer associated with scalar weights. Instead, *travel time functions* are used that map the entry time into a shortcut to the travel time through it. Unfortunately, these functions can become very complex. Computing and storing them is expensive. In the case of periodic piecewise linear functions, the number of breakpoints in a shortcut's function practically corresponds to the accumulated number of breakpoints of the functions of bypassed edges. Contrary to the classic setting, shortcuts aggregate the complexity of paths they represent, rather than skipping it. This leads to slow preprocessing and prohibitive memory consumption.

In this chapter, we explore an alternative approach to time-dependent shortcuts. Rather than explicitly storing travel time functions and obtaining paths lazily, we store paths and obtain travel times lazily. We expect that the shortest path between two vertices changes less frequently than the travel time. Intuitively, going via a highway may be slower due to congestion but is usually still the fastest option. Consider the functions f and g in Figure 8.1b. These functions are travel time functions of two paths between the same endpoints and have many breakpoints. If we want to store the travel time function of a shortcut between these endpoints, we need to



(a) A shortcut edge (dashed, black) bypassing several vertices. In this work, shortcuts always skip over exactly one vertex and two edges, which may in turn be shortcut edges (dashed gray edges).



(b) Travel time functions for two different paths between the same start and end vertex.

Figure 8.1: Shortcuts and their travel time functions.

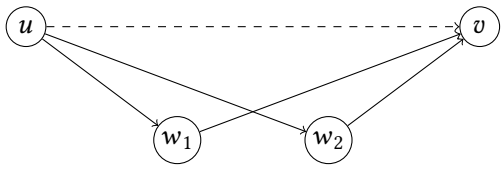
store the function $h = \min(f, g)$. Storing h explicitly requires roughly a number of breakpoints proportional to the number of breakpoints in f and g. However, if we only store which path is the fastest, we only need to store the points in time when the faster path switches. We expect significantly fewer path switches than travel time function breakpoints. Therefore, in this chapter, we explore a variant of time-dependent Contraction Hierarchies, where shortcuts store paths instead of travel times.

Attribution. This chapter is based on joint work with Dorothea Wagner and Ben Strasser. The results have been previously published as a conference paper at ESA 2020 [SWZ20] and a journal article in MDPI Algorithms [SWZ21].

Contribution. We introduce CATCHUp (Customizable Approximated Time-dependent Contraction Hierarchies through Unpacking), a time-dependent generalization of Customizable Contraction Hierarchies [DSW16]. At the heart of our approach are path unpacking shortcuts and we carefully engineer our implementation around this idea. Preprocessing is a few times faster than TCH [BGSV13], the only other technique we are aware of with fast, exact queries. Moreover, our preprocessing produces up to 40 times less auxiliary data. We achieve this by carefully employing approximation without sacrificing exactness. Further, our query running times are interactive on all instances, i.e. take only few milliseconds.

Outline. We begin by describing our data structures, algorithms and implementation. Section 8.1 introduces our shortcut unpacking data structure and algorithms for the efficient reconstruction of represented paths. The preprocessing, which computes auxiliary data from a road network with traffic predictions, is discussed in Section 8.2. In Section 8.3, we present query algorithms which utilize the auxiliary data to efficiently compute shortest travel times and paths between two given location. Section 8.4 contains an extensive experimental evaluation.

Problem Statement. In this chapter, we aim to design an efficient speedup technique for the two-phase TD-SPP$_{\mathbb{R}}$ with FIFO periodic piecewise linear functions over a horizon H of one day.



00:00	uw_1	w_1v
07:32	uw_2	w_2v
15:42	uw_1	w_1v

Figure 8.2: A shortcut with associated time-dependent expansions.

8.1 Shortcut Unpacking Data

CATCHUp is build on CCH and also computes an augmented graph G^+ during preprocessing. See the previous chapter for an extensive discussion of CCH. The key ingredient of CATCHUp is ~~tomatoes~~ the information we store with each edge of the augmented graph. We store time-dependent unpacking information, which allows us to efficiently reconstruct the original path represented by a shortcut for a given instant. Recall that a shortcut uv always skips over a triangle (u, w, v). However, there may be several triangles and which one is the fastest may change over time. There may also be an edge uv in the input graph which might sometimes be the fastest path. This is the information our unpacking data structure has to capture.

For each edge $uv \in \mathcal{E}^+$, we store a set of time-dependent *expansions* $\mathcal{X}(uv)$ for unpacking. Figure 8.2 presents an example. For an expansion $x \in \mathcal{X}(uv)$, we denote the time during which x represents the shortest path as the *validity interval* $V(x)$ of x. When formally referring to the path represented by an expansion, we use the *expand function* $\text{exp} : \mathcal{X} \to \mathcal{V} \cup \mathcal{E}$. exp either maps to an original edge $e \in \mathcal{E}$ or to the middle vertex w_x of the lower triangle (u, w_x, v). Knowing the middle vertex for each expansion is also sufficient to obtain longer paths. These can be computed by unpacking shortcuts recursively.

In our implementation, the expansion information is represented as an array of triples (τ, uw_x, w_xv). τ is the beginning of the validity interval and uw_x and w_xv are edge IDs. This information can be stored in 16 bytes for each entry – 8 bytes for the timestamp and 4 bytes for each edge ID. An expansion can also represent an original edge or no edge at all during a certain time interval. Both these cases are represented as special edge ID values. Two invalid edge IDs indicate no edge at all. One invalid ID indicates that the other edge ID represents an original edge. For pruning, we also additionally maintain a scalar lower bound $\underline{\mathsf{b}}[uv]$ and an upper bound $\overline{\mathsf{b}}[uv]$ on the travel time for each edge.

During preprocessing, we have to compute the expansion sets for each edge in the augmented graph. This is done using the same scheme as in CCH. We iterate over all edges and relax their lower triangles. Algorithm 8.1 depicts the routine for each triangle. The routine requires travel time functions for all involved edges. We maintain these functions during preprocessing but discard them later. To relax the lower triangle (u, w, v), the functions $\ell^+(uw)$ and $\ell^+(wv)$ are linked and the result is merged with the current function of $\ell^+(uv)$. Where (u, w, v) is faster, new expansions are inserted into $\mathcal{X}(uv)$. Where the current uv travel time is faster, the current expansions are kept.

Algorithm 8.1: Time-dependent lower triangle relaxation.

Input: Preliminary unpacking data for edge uv: $\mathcal{X}(uv)$ and travel time function $\ell^+(uv)$.
Travel time functions of lower triangle edges $\ell^+(uw)$ and $\ell^+(wv)$.
Output: Improved expansions $\mathcal{X}(uv)$ and function $\ell^+(uv)$.

Function `LowerTriangleRelax`**:**
  $f \leftarrow \ell^+(uw) \oplus \ell^+(wv)$
  $V_{uv} \leftarrow \{\tau \mid \ell^+(uv,\tau) \leq f(\tau)\}$
  $V_{uwv} \leftarrow \{\tau \mid f(\tau) < \ell^+(uv,\tau)\}$
  $\mathcal{X}(uv) \leftarrow \{\texttt{NewExpansion}(x, V_{uv} \cap V(x)) \mid x \in \mathcal{X}(uv)\} \cup \{\texttt{NewExpansion}((u,w,v), V_{uwv})\}$
  $\ell^+(uv) \leftarrow \min(\ell^+(uv), f)$
  return $\mathcal{X}(uv), \ell^+(uv)$

Algorithm 8.2: Evaluating the travel time of a CATCHUp shortcut.

Input: Expansions $\mathcal{X}(uv)$ for edge uv, instant τ.
Output: Travel time when traversing uv at τ.

Function `Eval`**:**
  $x_\tau \leftarrow x \in \mathcal{X}(uv)$ such that $\tau \in V(x)$ `// binary search`
  if $\exp(x_\tau) = uv \in \mathcal{E}$ **then**
    return $\ell(uv,\tau)$
  else
    $w \leftarrow \exp(x_\tau)$
    `tt` $\leftarrow$ `Eval`$(\mathcal{X}(uw), \tau)$
    return `tt` + `Eval`$(\mathcal{X}(wv), \tau +$ `tt`$)$

Once the unpacking information for an edge is complete, we can use it to compute the edge's travel time, unpack it to the path in the original graph, or compute the travel time function for the edge. All these operations follow the same basic scheme: Determine the relevant expansions and apply the operation recursively until edges from the input graph are reached. The simplest case is the travel time evaluation. Algorithm 8.2 depicts this operation. First, the relevant expansion is determined using binary search. If it points to an original edge, this edge's travel time can be evaluated and returned. If the expansion points to a lower triangle, we first recursively evaluate the first edge of the triangle. Then, the second edge can be evaluated at the entry time plus the travel time of the first edge.

Algorithm 8.3 depicts the procedure for determining the path represented by an expansion set for a given time. The recursive scheme is the same as for `Eval` but the result is a path instead of a travel time. Nevertheless, when unpacking lower triangles, we still need to evaluate the

Algorithm 8.3: Unpacking a CATCHUp shortcut into the represented original path at a certain time.

Input: Expansions $\mathcal{X}(uv)$ for edge uv, instant τ.
Output: Unpacked path $(u, \dots, v)$.

```
1 Function Unpack:
2     x_τ ← x ∈ X(uv) such that τ ∈ V(x) // binary search
3     if exp(x_τ) = uv ∈ E then
4         return (u, v)
5     else
6         w ← exp(x_τ)
7         P ← Unpack(X(uw), τ)
8         return P · Unpack(X(wv), τ + ℓ(P, τ))
```

first edges travel time to determine the time for unpacking the second edge.

Constructing the travel time function is also similar and shown in Algorithm 8.4. We recursively unpack expansions until we reach edges of the original graph where exact travel time functions are available. However, we may need to unpack several expansions for different times and combine them. For each expansion, we check if its validity overlaps with the time range for which we want to construct the travel time function. If so, we recursively retrieve the function for the first edge during this overlap. Then, we calculate the function for the second edge during the overlap. For the second edge, the time interval must be shifted by the travel time of the first edge at the start and end of the time interval. Both functions are then linked and appended to the final function.

Implementing this algorithm naively may cause performance issues since many memory allocations are performed for intermediate results. We avoid this by keeping all intermediate results in two buffers which are reused for all invocations of this algorithm. The buffers are stored as dynamically sized arrays (C++ vectors) and can grow dynamically but will never shrink. Once they have grown to an appropriate size, no more memory allocations will be necessary. Each buffer can contain many travel time functions stored consecutively. The link operation will read the last two functions from one buffer and append the result to the other buffer. Then, the two input functions will be truncated from the first buffer. After swapping, the buffers can be used again for the next link operation. Swapping is necessary, because it is not possible to read from and write to the same buffer during the same operation. The same scheme can be employed for joining partial functions (see Figure 8.3 for a visualization).

Algorithm 8.4: Reconstructing the travel time profile of a shortcut.

Input: Expansions $\mathcal{X}(uv)$ for edge uv, time interval T.
Output: Exact travel time function $\ell(uv)|_T$ for time T.

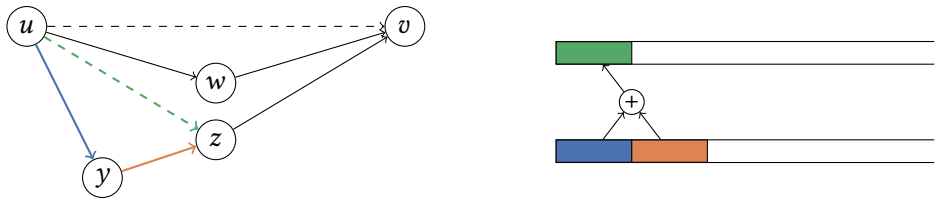
```
1  Function ReconstructProfile:
2      Initialize f_uv as function with empty domain
3      for x ∈ X(uv) do
4          [τ, τ'] ← T ∩ V(x)
5          if exp(x) = uv ∈ E then
6              f_x ← ℓ(uv)|_[τ,τ']
7          else
8              w ← exp(x_τ)
9              g ← ReconstructProfile(uw, [τ, τ'])
10             h ← ReconstructProfile(wv, [τ + g(τ), τ' + g(τ')])
11             f_x ← g ⊕ h
12         f_uv ← f_uv ∪ f_x
13     return f_uv
```

8.2 Preprocessing

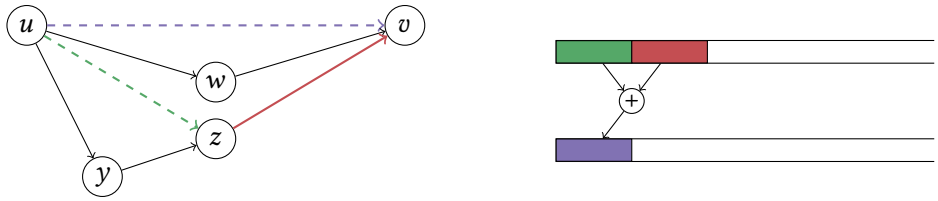
In this section, we present our preprocessing algorithms. As discussed in the previous chapter, the CCH preprocessing phase is performed only on the topology of the graph. We therefore apply the algorithms described in Section 7.1 without modification.

The goal of the CCH customization is to compute the shortcut weights ℓ^+. For our approach, we have to compute the travel time bounds and time-dependent expansions for all edges in the augmented graph. Recall that a shortcut $uv \in \mathcal{E}^+$ always bypasses triangles (u, w_i, v) for different vertices w_i, where w_i has lower rank than u and v. For the bounds, we want to find the minimum and maximum travel time of the fastest travel time function between u and v over any w_i. For the expansions, we need to determine for each point in time which lower triangle is the fastest. Assuming we know the final travel time functions of all uw_i and w_iv, we can compute this using the `LowerTriangleRelax` algorithm (see Algorithm 8.1). This leads to the following algorithm: Maintain a set of necessary travel time functions in memory, starting with the functions from the input graph. Iterate over all edges in the augmented graph in a bottom-up fashion. For each edge enumerate lower triangles. Link and merge their functions to compute the function, bounds, and expansions of the current edge. Keep the current edge's travel time function in memory until it is no longer needed.

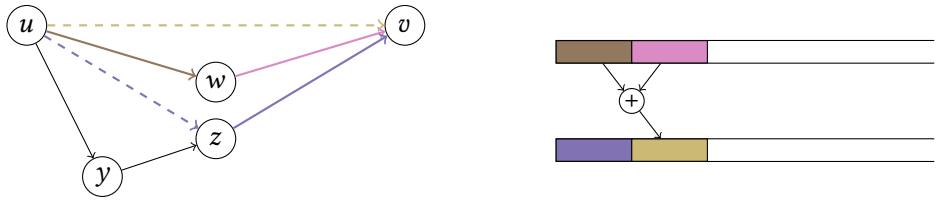
We implement this scheme as follows: In the respecting step, we initialize temporary travel time functions and expansion information for all edges in the augmented graph. Then, we process all edges uv ordered ascending by their lower-ranked endpoint. Since the middle vertex w of a lower triangle (u, w, v) has always lower rank than u and v, the edges uw and wv will



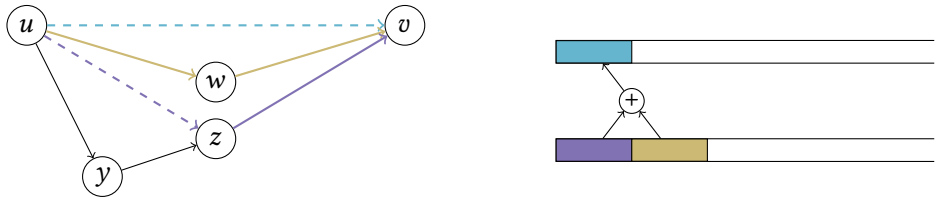
(a) Linking uy and yz to compute uz.



(b) Linking uz and zv to compute a part of uv with inverted buffer roles.



(c) Linking uw and wv to compute the other part of uv.



(d) Combining both parts of uv into one function which yields the final result.

Figure 8.3: Avoiding allocations when reconstructing shortcut travel time functions with two reusable buffers.

have been processed already. To process an edge uv we enumerate lower triangles (u, w, v) using the algorithms described in Section 7.2. Then, we execute the LowerTriangleRelax function for each triangle in both directions. Once all edges uv have been processed where u is the lower-ranked endpoint, we drop the travel time functions of all edges wu where u is the higher-ranked endpoint. This is crucial to keep memory consumption reasonable. Algorithm 8.5 depicts this in pseudocode.

So far, we described a straightforward adaptation of CCH algorithms to the time-dependent scenario. Making this approach practically efficient requires a few engineering tricks which we

Algorithm 8.5: Outline of the CATCHUp customization.

Input: Augmented graph G^+, input graph G with travel time functions ℓ.
Output: Expansion data $\mathcal{X}(uv)$ for all edges uv.

```
1  Function Respecting:
2      for uv ∈ E+ do
3          ℓ+(uv) ← ∞
4          X(uv) ← ∅
5      for uv ∈ E do
6          ℓ+(uv) ← ℓ(uv)
7          X(uv) ← {NewExpansion(uv, H)}
8  Function BasicCustomization:
9      for each vertex u ∈ V ordered by ascending rank do
10         for each edge uv ∈ E↑ do
11             for each lower triangle (u, w, v) of uv do
12                 RelaxLowerTriangle((u, w, v), X(uv), ℓ+(uv), ℓ+(uw), ℓ+(wv))
13                 RelaxLowerTriangle((v, w, u), X(vu), ℓ+(vu), ℓ+(vw), ℓ+(wu))
14         for uw ∈ E↓ do
15             Drop(ℓ+(uw))
16             Drop(ℓ+(wu))
```

describe in the following. A crucial ingredient to this are the scalar lower and upper bounds which we additionally maintain with each edge $uv \in \mathcal{E}^+$.

8.2.1 Pruning

Triangle Sorting. When enumerating triangles, we order them ascending by $\underline{\mathtt{b}}[uw] + \underline{\mathtt{b}}[wv]$. This way, we process triangles first which are likely faster. This gives us preliminary bounds on the travel time of uv. Before linking the time-dependent functions of a triangle $\ell^+(uw)$ and $\ell^+(wv)$, we check if $\overline{\mathtt{b}}[uv] \leq \underline{\mathtt{b}}[uw] + \underline{\mathtt{b}}[wv]$. If so, the linked path would be dominated by the shortcut, and we can skip linking and merging completely. If not, we link $\ell^+(uw)$ and $\ell^+(wv)$ and obtain the temporary function f. We still can skip merging if one function is strictly smaller than the other, i.e. either $\overline{\mathtt{b}}[uv] \leq \min_\tau(f(\tau))$ or $\max_\tau(f(\tau)) \leq \underline{\mathtt{b}}[uv]$. Even if the bounds overlap, one function might still dominate the other. To check for this case, we simultaneously sweep over the breakpoints of both functions, determining the value of the respectively other function by linear interpolation. Only when this check fails, we perform the merge operation.

Precustomization. Before the time-dependent customization, we first use the classical CCH basic and perfect customization algorithms on lower and upper bounds $\underline{\ell}$ and $\overline{\ell}$ obtained from

the input functions ℓ. This yields preliminary scalar lower and upper bounds $\underline{\ell}^+$ and $\overline{\ell}^+$ for all edges in the augmented graph. With these bounds, we can skip additional linking and merging operations. Note that the scalar bounds maintained during the time-dependent customization $\underline{\mathsf{b}}$ and $\overline{\mathsf{b}}$ are typically tighter than $\underline{\ell}^+$ and $\overline{\ell}^+$ because they are updated based on the temporary time-dependent travel time functions ℓ^+ used throughout the customization.

8.2.2 Perfect Customization

Recall that shortcut edges $uv \in \mathcal{E}^+$ allow skipping over paths of lower-ranked vertices and that we denote the length of the shortest such path by $\text{dist}^<(u,v) = \ell^+(uv)$. So far, we used that the customization deals with the weight function ℓ^+ for the augmented graph G^+ with $\ell^+(uv) = \text{dist}^<(u,v)$. This is, of course, only the *basic customization*. Because the augmented graph G^+ is valid for all possible weight functions, it contains many unnecessary edges for any concrete weight function. This is addressed by the perfect customization where the edges are processed again such that every edge gets the distance between its endpoints. Here, we refer to the weights after the perfect customization as $\ell^*(uv) = \text{dist}(u,v)$. The authors of [DSW16] proved that edges uv where $\ell^+(uv) > \ell^*(uv)$ are not necessary to answer queries correctly.

CATCHUp does not support a *time-dependent perfect* customization because we drop the time-dependent functions ℓ^+ as soon as possible and only maintain the expansions $\mathcal{X}$. However, a decent number of unnecessary edges can be identified by running the time-independent perfect customization on $\overline{\ell}^+$ to obtain $\overline{\ell}^*$. Each edge uv where $\text{dist}(u,v,\tau) \leq \overline{\ell}^*(uv) < \underline{\ell}^+(uv) \leq \text{dist}^<(u,v,\tau)$ can be removed. Therefore, before the time-dependent customization, we run the perfect customization on $\overline{\ell}^+$ and compare the obtained weights $\overline{\ell}^*$ to $\underline{\ell}^+$. We remove any edges uv where $\overline{\ell}^*(uv) < \underline{\ell}^+(uv)$. After, the time dependent customization, we do the same with the refined shortcut bounds $\underline{\mathsf{b}}$ and $\overline{\mathsf{b}}$.

8.2.3 Parallelization

We employ the separator based parallelization schema [BSW19] described in the previous chapter; see Section 7.2.2. With this approach, however, edges in the top-level separators are processed sequentially. Since these make up a significant share of the CATCHUp customization work, we employ loop-based parallelization [DSW16] inside the top-level separators and process the edges of each CH level independently in parallel.

8.2.4 Approximation

As we process increasingly higher-ranked edges, the associated travel time functions become increasingly complex, i.e. contain large numbers of breakpoints. This leads to two problems. First, linking and merging becomes very time-consuming as running times scale with the complexity of the functions. Second, storing these functions – even though it is only temporary – requires a lot of memory. We employ approximation to mitigate these issues. However, for

exact queries, we need exact unpacking information. We achieve this by lazily reconstructing parts of exact travel time functions during merging.

When approximating, we do not store one approximated function but two – a lower bound function ℓ^+_{lb} and an upper bound function ℓ^+_{ub} with maximum difference ϵ where ϵ is a configurable parameter. These approximations replace the exact functions ℓ^+ stored for later merge operations and will also be dropped when no longer needed. To obtain the bound functions, we first compute an approximation using the algorithm of Douglas and Peucker [DP73][1]. Then, we add or subtract ϵ to the value of each breakpoint to obtain an upper or lower bound, respectively. This bounds are valid, but they may not be as tight as possible. Therefore, we iterate over all approximated points and move each point back towards the original function. Both adjacent segments in the approximated functions have a minimum absolute error to the original function. We move the breakpoint by the smaller of the two errors. This yields sufficiently good bounds.

When linking approximated functions, we link both lower and both upper bound functions. Linking two lower bounds yields a valid lower bound of the linked exact functions because of the FIFO property. The same argument holds for upper bounds.

Merging approximated functions is more involved. Our goal is to determine the exact expansions for each edge. We use the approximated bounds to narrow down the time ranges when intersections are possible. For this, we merge the first function's lower bound with the second function's upper bound and vice versa. Where the bounds overlap, an intersection might occur. Then, we obtain the exact functions in the overlapping times using Algorithm 8.4 and merge them exactly. To obtain approximated upper and lower bounds of the merged function, we merge both lower bounds and both upper bounds (see Figure 8.4 for a visualization).

We approximate whenever a function has more than β breakpoints after merging. This includes already approximated functions. Both β and the maximum difference ϵ are tuning parameters which influence the performance (but not the correctness). We evaluate different choices in Section 8.4.1.

8.3 Queries

8.3.1 Earliest Arrival Queries

Our query algorithm is based on the CCH elimination tree query algorithm [DSW16]. Recall that, for an time-dependent shortest path query, we are given a source vertex s, a target t and a

[1] Previous works [BGSV13, BDPW16] have reported using the algorithm of Imai and Iri [II87] for approximation. Given a maximum error bound ϵ, this algorithm can compute in linear time the piecewise linear function with the minimum number of breakpoints within the given bound. The Douglas-Peucker algorithm has a quadratic worst case running time and no such guarantees on the number of breakpoints in the approximation. However, the theoretic guarantees of the Imai-Iri algorithm come at the cost of considerable implementation complexity and high constant runtime factors. Preliminary experiments showed that, compared to Imai-Iri, our Douglas-Peucker implementation actually produces insignificantly more breakpoints and also runs faster due to better constants. In addition, the implementation needs 30 instead of 400 lines of code, so we use the Douglas-Peucker variant.

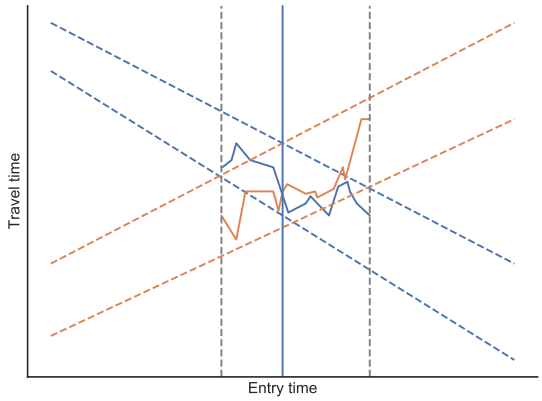


Figure 8.4: Merging approximated travel time functions by reconstructing the exact functions where bounds overlap.

departure time τ^{dep} from s. The goal is to obtain the earliest arrival at t and the respective path. Compared to a classical CCH query, our algorithm has to deal with two challenges. First, we cannot simply perform a backwards search, as we do not know the arrival time at the target vertex. Second, to evaluate the travel time of a shortcut, we need to obtain the path in the original graph which is an expensive operation. To address the first challenge, the query is split in two steps. In the first step, we obtain a subgraph on which we can run a forward-only Dijkstra-like search in the second step. We now present the basic query algorithm and later introduce optimizations to address the second challenge.

In the first step, the union of the subgraphs reachable from s in $G^{\uparrow}$ and t in $\overleftarrow{G}^{\downarrow}$ is obtained. We construct these subgraphs by traversing the elimination tree starting from both s and t to the root and marking all encountered edges as part of the search space. The backward search from t maintains parent pointers to represent the subgraph: For each encountered edge uv (where v has the higher rank), we store the edge ID and the tail u at v. This allows efficiently traversing these downward edges in the forward-only Dijkstra in the second step. In the second step, we run Dijkstra's algorithm on the combined search spaces. Shortcut travel times are evaluated with Algorithm 8.2.

By CH construction, the search space contains the shortest path. Thus Dijkstra's algorithm will find it and our algorithm will determine the optimal arrival at t. However, the search space is bigger than necessary. This slows down the query. In the next paragraph, we discuss how to construct smaller subgraphs using an elimination tree interval query.

Elimination Tree Interval Query

The elimination tree interval query is a bidirectional search starting from both the source vertex s and the target vertex t. It constructs a smaller subgraph for the second step. We denote this

subgraph as a *shortest path corridor*. Vertex labels contain an upper $\overline{\mathrm{D}}[v]$ and a lower bound $\underline{\mathrm{D}}[v]$ on the travel time to/from the search origin and a parent pointer to the previous vertex and the respective edge id. The bounds $\overline{\mathrm{D}}[s]$, $\overline{\mathrm{D}}[t]$, $\underline{\mathrm{D}}[s]$, $\underline{\mathrm{D}}[t]$ are all initialized to zero in their respective direction, all other bounds to infinity. We also track tentative travel time bounds for the total travel time from s to t. For both directions, the path from the start vertex to the root of the elimination tree is traversed. For each vertex u, all edges uv to higher-ranked neighbors are relaxed, that is checking if $\overline{\mathrm{D}}[u] + \overline{\mathrm{b}}[uv] < \overline{\mathrm{D}}[v]$ or $\underline{\mathrm{D}}[u] + \underline{\mathrm{b}}[uv] < \underline{\mathrm{D}}[v]$ and improving the bounds of v if possible. When the new travel time bounds from an edge relaxation overlap with the current bounds, more than one label has to be stored. As an optimization [BSW19], vertices can be skipped if the lower bound on the travel time to it is already greater than the tentative upper bound on the total travel time between s and t. After both directions are finished, we have several vertices in the intersection of the search spaces. Where the sum of the forward and backward distance bounds of such a vertex overlaps with the total travel time bounds, the parent pointers are traversed to the search origin and all edges on the paths are marked as part of the shortest path corridor.

Lazy Shortcut Unpacking

In the second query step, we perform Dijkstra's algorithm on the corridor obtained in the first step. In the basic query, shortcuts are unpacked completely to evaluate their travel time. However, this may cause unnecessary and duplicate unpacking work. We now describe an optimized version of the algorithm which performs unpacking lazily. The algorithm starts with the same initialization as a regular TD-Dijkstra. All earliest arrivals are set to infinity, except for the start vertex which is set to the departure time. The start vertex is inserted into the queue. Then, vertices are popped from the queue until it is empty or the target vertex is reached. For each vertex, all outgoing edges within the shortest path corridor are relaxed. When an edge is from the input graph, its travel time function can be evaluated directly. Shortcut edges, however, need to be unpacked. The lazy unpacking algorithm defers as much work as possible: Only the first edge of the triangle of each shortcut will be recursively unpacked until an input edge is reached, the second edge will be added to the corridor. Figure 8.5 shows an example. This way, we unpack only the necessary parts and avoid relaxing edges multiple times when shortcuts share the same paths.

Corridor A*

The query can be accelerated further, by using the lower bounds obtained during the elimination tree interval query as potentials for an A*-search. For vertices in the CH search space of t, the lower bounds from the backward search can be used. For vertices in the CH search space of s, we start at the meeting vertices from the corridor search and propagate the bounds backwards down along the parent pointers. This yields potentials for all vertices in the initial corridor. However, we also need potentials for vertices added to the corridor through unpacking. These

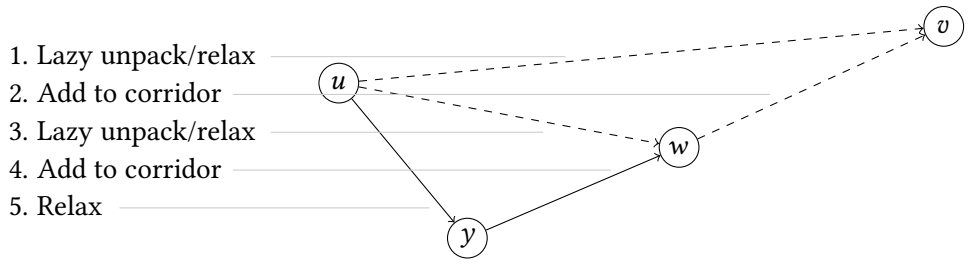


Figure 8.5: Lazy relaxation of edge uv. Since uv is a shortcut, it needs to be unpacked. This causes wv to be added to the corridor and uw to be relaxed. Relaxing uw causes yw to be added to the corridor and uy to be relaxed. In this example, uy is an original edge and the recursion stops. yw will be relaxed (or unpacked) only once y is popped from the queue.

potentials are computed during the shortcut unpacking. When unpacking a shortcut uv into the edges uw and wv, then $\pi_t(w)$ will be set to $\min(\pi_t(w), \pi_t(v) + \underline{b}[wv])$.

Justifying that A* with these potentials will always find the correct earliest arrival is surprisingly non-trivial. In fact, these potentials are not *feasible* in the sense that $\forall \tau \in T, uv \in \mathcal{E}$: $\ell(uv, \tau) - \pi_t(u) + \pi_t(v) \geq 0$. Figure 8.6 shows an example where the term becomes negative and the same vertex has to be popped several times from the queue. Assume that all edges have a constant travel time for the departure time of this query and lower bounds are equal to the travel time. The exception is v_3t which has constant travel time 100 during this query but the lower bound is zero. We use zero weights to simplify the example. They are not strictly necessary for such an example. The shortest path from s to t is (s, w_2, w_1, v_2, t) and has length 22. After s is settled, the queue will contain v_3 with key $1 + 0$ (distance plus lower bound to t), v_1 with key $1 + 2$ and w_2 with key $1 + 21$. Then, v_3 will be settled which will insert t with key $101 + 0$ into the queue. Then, v_1 will be settled and w_1 will be inserted into the queue with key $3 + 0$. Then, w_1 will be settled even though the current distance of 3 is greater than the actual shortest distance of 2. This will insert v_2 with key $13 + 10$ into the queue. Now, w_2 will be popped and the distance to w_1 will be improved and it will be reinserted into the queue with key $2 + 20$. w_1 will be popped immediately afterwards which improves the distance and key of v_2 which is the next vertex to be popped from the queue. After it has been processed, the final distance to t (22) is known, and t is the final vertex to be settled.

Nevertheless, we claim that once t is popped from the queue the algorithm *always* has found the correct earliest arrival. The reason is the following lemma.

Lemma 8.1. *For all vertices $v \in \mathcal{V}$ on the shortest path* $\text{dist}(s, v, \tau^{\text{dep}}) + \pi_t(v) \leq \text{dist}(s, v, \tau^{\text{dep}})$ *holds, i.e. the potential fulfills the lower bound property.*

Proof. Let $P = (s, \ldots, t)$ be the desired shortest path from s to t when departing from s at τ^{dep}. For vertices in the initial corridor obtained by the interval query, $\text{dist}(s, v, \tau^{\text{dep}}) + \pi_t(v) \leq \text{dist}(s, t, \tau^{\text{dep}})$ always holds because the potential $\pi_t(v)$ is set to the global lower bound $\underline{\text{dist}}(v, t)$. However, vertices u added later to the corridor through unpacking may have a greater potential

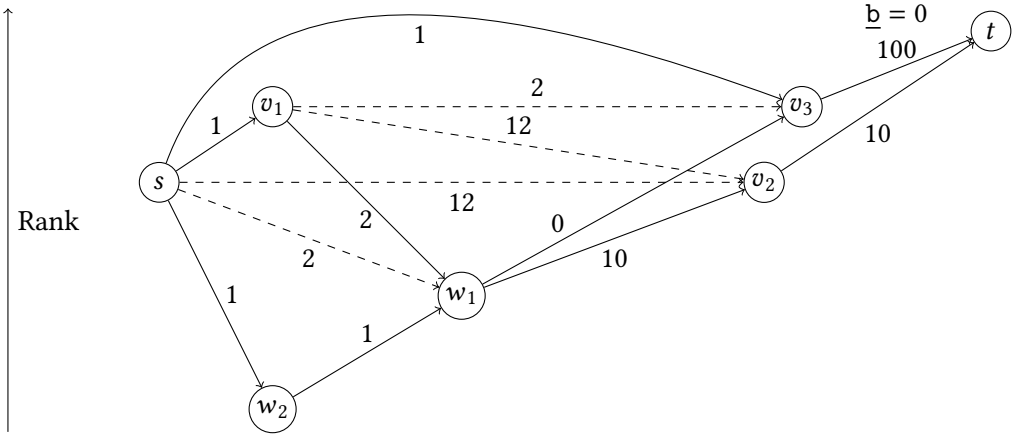


Figure 8.6: Example of a query where our A* potentials lead to a non-label-setting query. Dashed edges are shortcuts. The shortcut weights are not known to the query algorithm.

than $\underline{\text{dist}}(u,t)$ as depicted in the example. However, their final potential cannot be greater than the lower bound of the travel time on the shortest path $\underline{\ell}(P_{u,t})$. This is enough to satisfy $\text{dist}(s,u,\tau^{\text{dep}}) + \pi_t(u) \leq \text{dist}(s,t,\tau^{\text{dep}})$. In addition, the potential value will be set to this final value before t is settled. Assume for contradiction that p_i is the first vertex on P for which this statement does not hold. Clearly, p_i cannot be a vertex from the initial corridor. However, $\pi_t(p_i)$ will be set at most to $\underline{\ell}(P_{p_i,t})$ once p_{i-1} is settled which by assumption happens before t is settled. This is a contradiction. Thus, $\text{dist}(s,v,\tau^{\text{dep}}) + \pi_t(v) \leq \text{dist}(s,t,\tau^{\text{dep}})$ holds for all vertices $v \in P$ and the query algorithm always finds the correct earliest arrival when terminating once t is popped from the queue. □

8.3.2 Profile Queries

A profile query asks for the function of the fastest travel time between two vertices over a given time period T. Here, we assume that the T equals the entire horizon H covered by the input network. As discussed in Chapter 5, such a query can be answered with a variation of Dijkstra's algorithm. However, this algorithm exhibits both prohibitive running time and memory consumption. Consider a path $(v_0, \ldots, v_k)$ where the travel time function of each edge has p breakpoints. In general, linking two functions f and g creates a new function with $\Theta(|f| + |g|)$ breakpoints. Thus, the travel time function from v_0 to vertex v_i contains $\Theta(i \cdot p)$ breakpoints. Therefore, when computing the function between v_0 and v_k, the total memory consumption and the running time is in $\Theta(\sum_{i=1}^{k} pi) = \Theta(pk^2)$, i.e. grows quadratically with the length of the path. We conclude that Dijkstra-based approaches to profile queries are not a promising direction. The experiments with Dijkstra-based TCH profile queries in [BGSV13]

support this conclusion. We also performed preliminary experiments with a proof-of-concept implementation where we adapted our earliest arrival query algorithm to the profile query setting. The Dijkstra-based approach was more than an order of magnitude slower than the approach described in the following. Instead of a Dijkstra-like search, we employ balanced contraction using the methods from the preprocessing. With balanced contraction, the running time to compute the travel time profile of the path example is in $\Theta(p \cdot k \log k)$.

Our profile algorithm has four steps. The first step uses the elimination tree interval query to obtain a shortest path corridor. It is the same as for earliest arrival queries. In the second step, we obtain travel time functions for all edges in the shortest path corridor. During the third step, additional shortcuts will be inserted and their unpacking data will be computed, reusing the preprocessing algorithms. The result is a new st shortcut with exact expansions $\mathcal{X}(st)$. From this unpacking information, the exact travel time function can be obtained in the fourth step. We now describe steps 2–4 in detail.

Reconstruction. In this step, we obtain travel time functions for all edges in the shortest path corridor. Similar to the customization, the obtained travel time functions may be either exact, or approximated upper and lower bound functions. This keeps the memory consumption low and linking and merging operations fast. We obtain these functions by first recursively reconstructing the travel time functions of all edges referenced by expansions. The reconstructed functions will be saved for both edges in the corridor and unpacked edges, in case another reconstruction operation might reuse an edge. If all edges referenced by the expansions have an exact function available, we can compute an exact travel time function for the current edge. If not, we end up with an approximation. After reconstruction, we check if a function has more than β breakpoints. If that is the case, we approximate it to reduce the complexity (see Section 8.2).

Contraction. In the third step, we insert additional shortcuts and compute their unpacking data by simulating the contraction of the vertices in the shortest path corridor. We reuse the existing nested dissection order. The ranks of s and t are increased such that they are higher in the hierarchy than all other vertices. This construction leads to a shortcut between s and t, one between s and each vertex in the corridor on the path from s to the elimination tree root, and one from each vertex in the corridor on the path from t to the root to t. Some of these shortcuts may already exist See Figure 8.7 for an illustration.

These new shortcuts are now processed as in Algorithm 8.5 to compute their unpacking data. We initialize the shortcut bounds with the bounds obtained from the elimination tree query. This allows to prune unnecessary operations. We process shortcuts ordered by their lower-ranked endpoint. For each shortcut we enumerate and relax lower triangles using Algorithm 8.1. We can enumerate these triangles efficiently using the parent pointer from the interval query. Each shortcut has an endpoint vertex in the corridor. The parent pointers of this vertex correspond to the triangles that need to be relaxed. The shortcuts from s and the shortcuts to t are independent

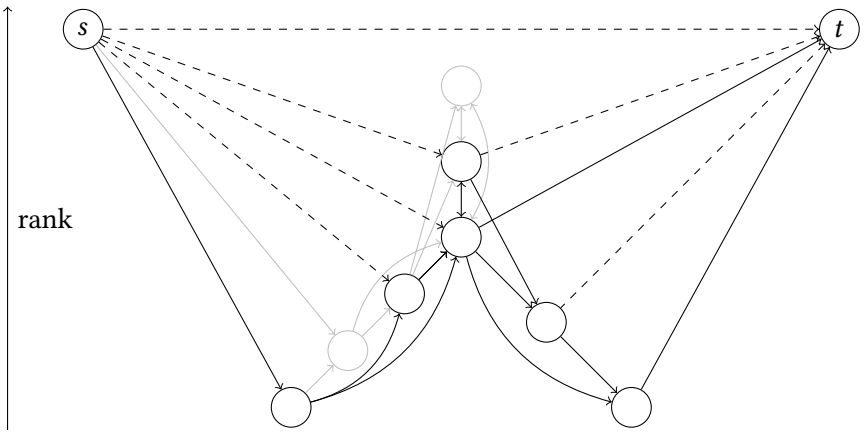


Figure 8.7: Example of profile query search space with inserted shortcuts. Gray vertices and edges are not in the shortest path corridor. Dashed edges are new shortcuts.

of each other and can be processed in parallel. The lower triangles of the *st* shortcut can be enumerated by iterating over the meeting vertices from the interval query. We also employ the triangle sorting optimization.

Extraction. In this final step, we can use the unpacking information of the *st* shortcut to efficiently compute the final result. The shortcut already contains a possibly approximated travel time function from the contraction step. This may suffice for some applications. If the shortcut contains only an approximation, but we need an exact travel time profile, we can use Algorithm 8.4 to compute it. For some practical applications, the different shortest paths over the day may be more useful than the travel time profile. Algorithm 8.6 depicts a routine to compute path switches and the associated shortest paths. The algorithm follows the same scheme as all unpacking algorithms. The operation is recursively applied to all expansions limited to the validity time of the expansion. Only the `Combine` operation is more involved. It performs a coordinated linear sweep over the path sets from uw and wv and appends the paths where the validity intervals overlap. For the paths from w to v, we only know the validity times with respect to departure at w. To obtain the corresponding departure time at u, we reverse evaluate the current uw path, i.e., we successively evaluate the inverted arrival time function of all edges on the path in reverse order.

8.4 Evaluation

In this section, we present our experimental results. We first discuss the experimental setup and the input road networks. Then, we discuss the performance of each of our presented algorithms in turn. Finally, we compare our approach to related work.

Algorithm 8.6: Reconstructing the represented path set by a shortcut.

Input: Expansions $\mathcal{X}(uv)$ for edge uv, time interval T.
Output: Set $\mathcal{P}$ of unpacked paths P with associated validity times V_P.

```
1  Function UnpackPaths:
2      P ← ∅
3      for x ∈ X(uv) do
4          [τ, τ'] ← T ∩ V(x)
5          if exp(x) = uv ∈ E then
6              P_x ← {((u, v), [τ, τ'])}
7          else
8              w ← exp(x)
9              P_uw ← UnpackPaths(uw, [τ, τ'])
10             P_wv ← UnpackPaths(wv, [τ + ℓ(P_uw, τ), τ' + ℓ(P_uw, τ')])
11             P_x ← Combine(P_wv, P_uw)
12         P ← P ∪ P_x
13     return P
```

Environment. Our benchmark machine runs openSUSE Leap 15.2 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.5 GHz and 8 × 64 KiB of L1, 8 × 1 MiB of L2, and 24.75 MiB of shared L3 cache. Hyperthreading was disabled and parallel experiments use 16 threads. We implement our algorithms in Rust[2] and compile them with `rustc 1.49.0-nightly (cf9cf7c92 2020-11-10)` in the release profile with the `target-cpu=native` option[3]. To compile competing implementations written in C++, we use GCC 9.3.1 using optimization level 3 and the `-march=native` option.

Methodology. We investigated the performance of our preprocessing and query algorithms and compared it to competing algorithms. Our experiments were focused on but not limited to space consumption and running times. We performed preprocessing five times for each input network and report arithmetic means of the running times. Unless reported otherwise, preprocessing utilized all 16 cores. For queries, we generated 100 000 source, target, departure time triples chosen uniformly at random for each graph. These were executed in bulk. Competing algorithms were evaluated with the same query set. For profile queries, we only used 1000 queries (and discarded the departure time). We report arithmetic means of query running times and machine independent measures such as number of vertices popped from the queue and number of evaluated travel time functions.

[2]The code for this paper and all experiments is available at `https://github.com/kit-algo/catchup`.
[3]We disable AVX512 instructions, as they caused misoptimizations.

Table 8.1: Preprocessing statistics. Running times are for parallel execution with 16 cores.

	CCH edges	Expansions per edge			Data	Running time [s]	
	$[\cdot 10^3]$	Avg.	Max.	= 1 [%]	[GB]	Prepr.	Custom.
Ber	1 977	1.039	31	98.6	0.09	1.5	6.2
Ger06	22 519	1.075	44	98.4	1.06	30.1	21.6
Ger17	31 488	1.090	107	98.5	1.50	35.0	107.4
Eur17	114 857	1.099	115	98.4	5.47	189.6	557.0
Ger19	75 800	1.668	369	96.1	4.30	135.7	11 581.1
Eur20	128 921	1.191	109	96.9	6.32	209.6	1 039.5

Inputs. We perform experiments on all instances listed in Section 4.2.2. However, for the main evaluation of this chapter, we limit our discussion to a subset: The city scale Ber instance and the Ger06 instance for comparability to related work, the Ger17, Eur17, Eur20 instances by PTV as examples of modern networks with increasingly complex but well-behaved traffic predictions, and the OSM/Mapbox-based Ger19 instance which is even harder as it is based more on raw speed observations rather than modelled traffic predictions. Further, we use only one weekday per instance. Results for the full instance set are reported in Appendix C.

8.4.1 Preprocessing

Table 8.1 reports the results for preprocessing. On Ger06, the preprocessing takes longer than the customization. However, for the newer instances with more time-dependent edges and more breakpoints per function this changes and the customization becomes more expensive. Despite that, the amount of auxiliary data corresponds only to the number of edges in the augmented graph and does not grow as much for the newer instances. The high complexity of the input function on Ber also does not depict any negative influence on the data amount or the number of expansions per shortcut. On average, only up to 1.2 expansions per edge need to be stored for all graphs expect Ger19 where the average goes to a little under 1.7. About 96% or more of all edges have only a single expansion. The maximum number of expansions per edge is only 115 on the PTV instances and 369 for Ger19. We determined the travel time functions of the specific shortcuts with these numbers of expansions and observed that the number of breakpoints in their travel time functions is still two orders of magnitude larger. On Eur20, the total preprocessing time is about 20 minutes, roughly twice as much as for Eur17. However, the space consumption grows by less than 1 GB and is in fact smaller than the input graph. On Ger19, the preprocessing time is around three and a half hours but the space consumption remains small. This clearly demonstrates the advantage in space efficiency of expansion information over explicitly storing travel time functions.

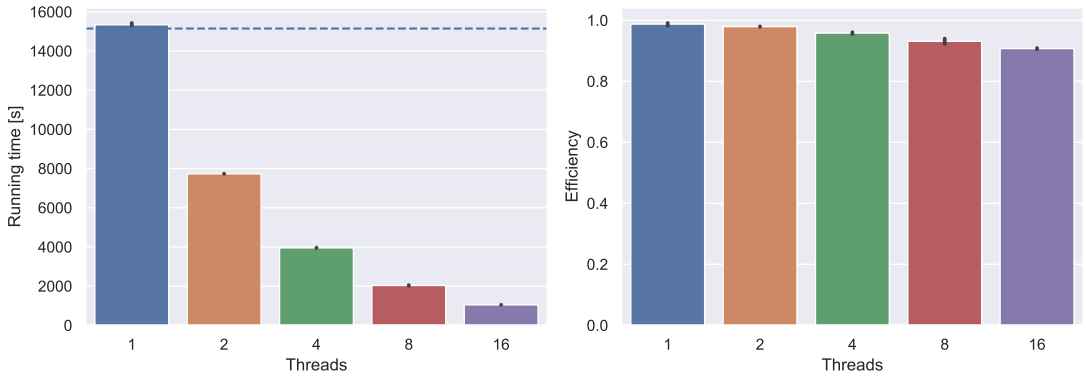


Figure 8.8: Average customization running times and parallelization efficiency (speedup/number of threads) on five customization runs of Eur20. The black bars (barely visible) indicate the standard deviation. The dashed line indicates running time with all parallelization code disabled.

Customization

We now focus on the customization. The first part of the preprocessing did not use any time-dependent information and only reused existing algorithms which have been evaluated in the previous chapter. For the scope of this subsection, we only use the Eur20 graph.

To evaluate their impact, we selectively disable the triangle sorting and precustomization optimizations. Even though both optimizations speed up the customization by improving bounds, both have a significant impact on their own. Disabling the precustomization increases the overall customization running time by about five minutes to 1311 s. The effect of triangle sorting is even stronger. Disabling it roughly doubles customization running time to 2156 s.

Parallelization. We evaluate the effectiveness of our parallelization scheme and run the customization with a varying number of threads. Figure 8.8 depicts the results. As a baseline, we run the experiment with all parallelization code disabled. The baseline running time is indicated by the dashed line. Enabling parallelization but running with only one thread causes only little overhead. Running with more threads introduces more overhead due to synchronization. With 16 threads, parallel efficiency is still around 0.9. We conclude that our parallelization scheme scales well and that customization times could be reduced further by utilizing additional cores.

Approximation. We perform customization experiments with different approximation parameters. Over the course of the customization, we track the progress over time, the memory consumption and the average travel time function complexity. Figure 8.9 displays these measurements. We use the number of processed triangles to measure the progress because it corresponds roughly linearly to the time passed (though different parameters lead to different slopes).

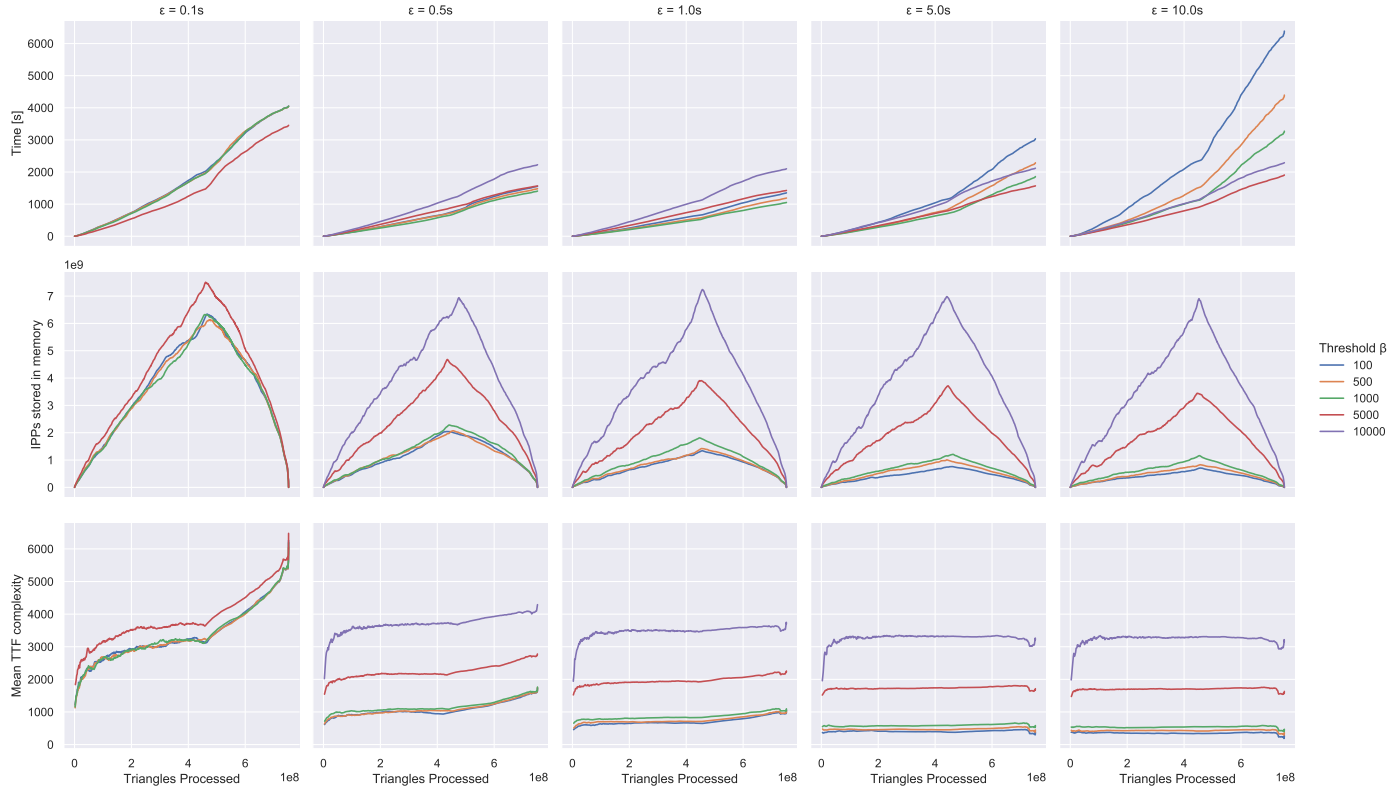


Figure 8.9: Customization behavior depending on the approximation parameters difference ϵ (varying by column) and threshold β (indicated by color). The x-axis in all plots indicates the progress of the customization by number of processed triangles. The y-axis is the passed time in the first row, the memory usage in the second row (measure by the total number of stored breakpoints), and the mean travel time function complexity in the third row. There are many more elements which contribute to memory consumption. However, the breakpoints for travel time functions are the biggest chunk and are the easiest to measure. A breakpoint has a size of 16 bytes in memory. Thus, $8 \cdot 10^9$ breakpoints correspond to 128 GB memory consumption for travel time functions alone. The configuration $\beta = 10000$, $\epsilon = 0.1\,s$ caused an out-of-memory error and is not listed.

After about 60% of the triangles, the slope, i.e. the time per triangle changes slightly. At this point only high-level separator vertices/edges remain. These have complex travel time functions so linking and merging becomes more expensive. Also, we switch from task based to loop based parallelization which is less effective. Measuring progress by processed vertices (the for loop in Line 1 in Algorithm 8.5) or processed edges (for loop in Line 2) is also insightful. However, for these, the correspondence is not linear. The last couple of thousand vertices and the last million edges take almost half the total time.

We observe that the choice of approximation parameters has a huge influence on running time and memory consumption. The best running time of around 1000 s is achieved with $\beta = 1000$ and $\epsilon = 1.0\,s$. Thus, we use it as our default configuration. The worst running time is over six times higher. In the best configuration, we use only around 25 GB for travel time functions while a bad parameter choice or no approximation leads to crashes with out-of-memory errors. Generally, a larger ϵ leads to looser approximation, lower travel time function complexity, and thus less memory consumption. Conversely, a larger β causes the approximation to be executed less often and the memory consumption and function complexity increases. The average function complexity is usually well below β except for very small values of ϵ or β. In that case the complexity cannot be reduced sufficiently to keep the complexity below β. If β is large, approximation is performed seldom and the influence of ϵ becomes smaller. Similarly, when ϵ is small, the influence of β is limited because the complexity cannot be reduced enough no matter how often approximation is performed. Clearly, the right choice of approximation parameters is essential to the performance of the preprocessing. When travel time functions are too complex, too much memory is used, and linking and merging are very expensive. However, when functions are approximated too loosely, a lot of time is spent in the reconstruction of exact functions for the times when bounds overlap. Thus, extreme parameter choices for ϵ and β are detrimental to the running time, even though they may reduce memory consumption.

Through all configurations, the memory usage peaks after around 60% of the triangles have been processed. The reason is the way we maintain travel time functions in memory during the customization. An edge's travel time function is stored once its lower-ranked endpoint has been processed until its higher-ranked endpoint has been processed. In the beginning, we process vertices with low rank and store many travel time functions. Only once we reach the higher-ranked vertices, we start dropping a significant amount of the stored functions. This causes the observed peak.

8.4.2 Queries

In this section, we investigate the performance of our query algorithms. Table 8.2 depicts the influence of the query optimizations. The basic approach does not achieve competitive running times. Queries take almost a second on average on the newer Europe graphs. On Ger19, the basic approach is even slower than Dijkstra's algorithm. Clearly, naively evaluating shortcut travel times with Algorithm 8.2 is too slow. Limiting the search space to a shortest path corridor using the elimination tree interval query significantly reduces running times. The speedup

Table 8.2: Query performance with different optimizations. We report the number of vertices popped from the queue, the number of evaluated travel time functions (TTFs) and the running time. All values are arithmetic means over 100 000 queries executed in bulk with source, target and departure time drawn uniformly at random.

		Queue pops	Evaluated TTFs	Running time [ms]
Ber	Basic	167.4	100 820.5	8.8
	+ Corridor	38.1	5 224.1	0.6
	+ Lazy	1 603.6	1 747.4	0.6
	+ A*	635.2	691.5	0.3
Ger06	Basic	492.3	818 721.3	46.4
	+ Corridor	79.7	31 740.8	2.3
	+ Lazy	3 323.2	3 838.0	1.7
	+ A*	831.0	995.1	0.6
Ger17	Basic	510.3	2 100 731.8	169.7
	+ Corridor	143.4	164 372.5	13.7
	+ Lazy	18 450.0	19 910.5	9.1
	+ A*	3 099.2	3 495.5	1.7
Eur17	Basic	861.6	9 951 623.1	808.6
	+ Corridor	229.3	806 727.8	62.3
	+ Lazy	39 714.8	43 581.1	20.8
	+ A*	6 876.5	7 911.0	4.1
Ger19	Basic	894.2	40 675 596.0	4 329.0
	+ Corridor	618.1	11 563 911.1	1 166.1
	+ Lazy	193 779.2	220 416.1	151.2
	+ A*	23 218.1	29 816.5	16.5
Eur20	Basic	871.0	10 527 072.7	813.2
	+ Corridor	335.6	1 222 655.6	92.9
	+ Lazy	62 677.7	70 145.4	33.7
	+ A*	7 231.9	8 844.7	4.7

is between a factor of 20 on Ger06 and 4 and Ger19. The effectiveness of this optimization corresponds inversely to the relative delay (see Table 4.2). Greater relative delays lead to bigger corridors and thus smaller speedups.

The lazy evaluation optimization has a smaller but still significant impact on the running time (speedups between 1.3 and 8). Further, it drastically shifts the balance between queue

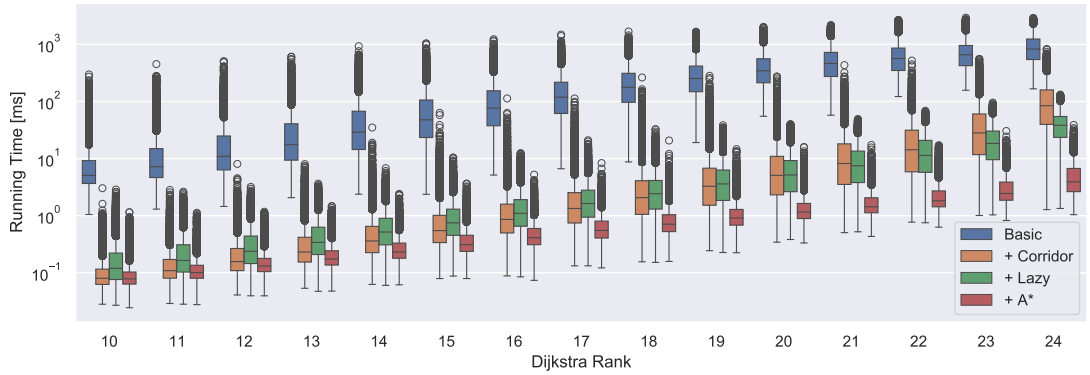


Figure 8.10: Query running times in milliseconds with different optimizations by Dijkstra ranks on Eur20. The boxes cover the range between the first and the third quartile. The band in the box indicates the median. The whiskers indicate 1.5 times the inter quartile range. Running times outside this range are considered as outliers and displayed separately.

operations and travel time function evaluations. On the newer graphs, the number of queue pops increases by more than two orders of magnitude, while the number of travel time function evaluations decreases by up to a factor of 50. The additional queue operations introduce some overhead. However, this is mitigated by the avoided unnecessary and duplicate evaluations. Reusing the lower bounds from the corridor search for goal directed search yields an additional speedup of factor two to nine. On all instances, running times with all optimizations are fast enough for interactive applications.

Local Queries

We generate another set of queries to investigate the performance of our algorithms depending on the distance of source and target. We draw 10 000 start vertices and departure times uniformly at random and perform time-dependent Dijkstra without a specific target. For every 2^ith settled vertex, we store it as the target of a query of *Dijkstra rank i*. This methodology was introduced by [SS05]. Figure 8.10 shows query running times by rank for the query algorithm with the various query optimizations enabled successively.

Obviously, query running times scale with the distance. The fully optimized algorithm takes only fractions of milliseconds for short range queries, except for some outliers which take up to a millisecond. For long range queries, we usually achieve query times within a couple of milliseconds and the maximum query time was 39 ms. The basic query algorithm is around two orders of magnitude slower across all ranks. The impact of lazy optimization appears to depend on the rank of the query. For lower ranks, it introduces some overhead but reduces outliers compared to only the corridor optimization. This is due to the overhead of the queue operations. For long range queries, this is completely amortized by the reduction in edge relaxations.

Table 8.3: Running times of profile queries and characteristics of the obtained profiles. We report total running times and running times of each step (Corridor, Reconstruction, Contraction, Extraction) of the query. The total running time is slightly larger than the sum of all steps as it includes some additional initialization and cleanup work. We report the number of breakpoints in the obtained travel time profile (Column $|\ell|$). Column $|\mathcal{X}|$ contains the number of times the shortest path changes during the day. Since the same path may be the fastest for several times, we also report the number of distinct paths in the last column. All values are arithmetic means over 1000 queries executed in bulk with source and target vertices drawn uniformly at random.

	Running time [ms]								
	I	II	III	IV		Total	$\lvert\ell\rvert$	$\lvert\mathcal{X}\rvert$	Distinct paths
				TTF	Paths				
Ber	0.1	37.8	13.9	2.8	0.4	55.6	30 974.9	2.7	2.3
Ger06	0.4	56.5	23.0	0.7	0.8	83.6	9 359.6	6.9	3.3
Ger17	0.8	452.2	189.0	6.1	2.4	660.1	66 146.0	9.7	3.7
Eur17	1.7	1 135.8	732.9	13.0	7.8	1 913.2	122 192.1	16.5	6.8
Ger19	2.3	34 345.4	230 643.8	79.3	51.8	265 453.7	525 196.4	91.4	34.3
Eur20	2.8	3 166.6	1 507.7	12.3	10.3	4 747.5	107 690.7	24.4	11.6

Profile Queries

We perform experiments for profile queries and report the results in Table 8.3. The total running time depends on the amount of time-dependent information in the instance. From Ger06 to Ger17 the total running time increase by a factor of 8 even though the network grows only little. The same can be observed between Eur17 and Eur20.

The total running time is dominated by the reconstruction and contraction steps. On the PTV instances, reconstruction of the travel time functions of the existing shortcuts takes roughly twice as long as computing the functions for the new shortcuts in the contraction step. In contrast, on Ger19, the contraction takes almost an order of magnitude longer than the reconstruction. The corridor step with the elimination tree interval query takes a negligible amount of time. Surprisingly, the time required to compute a final exact travel time profile is not much greater than to compute a path profile (on Ger06, the path profile is even slower than the travel time profile).

The average complexity of the final travel time function varies by orders of magnitude across the different instances. This confirms that Ger06 is a relatively simple instance. Surprisingly, the average complexity on Eur17 is higher than on Eur20. We suspect that this is caused by the higher average complexity in the input graph and that many important edges that cover many shortest paths already have a non-constant travel time function in Eur17. The number of path switches is magnitudes smaller. It ranges from three path switches on Ber to 91 on

Ger19. The number of distinct paths during the day are roughly between a third and half of that. The arithmetic mean of these numbers is slightly skewed upwards by a few very high values. However, the median is still fairly close: For example, the median number of distinct paths is two on Ger06 and ten on Eur20.

Total running times of our algorithm are practical but not interactive on all instances except Ger19. Computing full day profiles for long-range queries on Ger19 takes minutes and slower queries even take up to half an hour. Still, even being able to compute profiles for long-range queries on this instance is something that is likely not possible with any other technique.

8.4.3 Comparison with Related Work

In the following, we provide an overview over different techniques, their preprocessing and query times, space overhead and average query errors where approximation is used. Where possible, we obtained the code of competing algorithms[4] and evaluated them with same methodology, instances and queries as our algorithms. For other competitors, we report available numbers from the respective publications.

Table 8.4 depicts the results for the Ger06 instance used in many older works. KaTCH, heu SHARC, CFLAT and CATCHUp all achieve query times around 0.6 ms. The original research implementation TCH reports slightly slower times than KaTCH. This may be because experiments were run on an older machine, but also because according to the KaTCH documentation, the newer query is somewhat more efficient. TCH pays for this speed with 4.7 GB of auxiliary data. Reducing the KaTCH memory consumption while keeping exactness (ATCH) brings query times up to 1.24 ms. ATCH also feature a configuration where they only keep upper and lower bounds for each travel time function (ATCH ∞). This configuration uses even less memory than CATCHUp because the optimized order results in fewer shortcuts. However, query running times degrade to 1.66 ms. Giving up on exactness allows keeping the query times at 0.7 ms (inex. TCH) but introduces noticeable errors.

While achieving competitive query times for acceptable memory consumption, heu SHARC suffers from preprocessing times of several hours. The publication does not report average query errors, only a maximum error of 0.61%. TD-CALT has small memory consumption but no competitive query times, even when approximating. Further, the exact variant is outperformed by CH-Potentials in all dimensions. CH-Potentials has the smallest memory consumption, but queries are relatively slow. FLAT and CFLAT both suffer from extreme preprocessing times and memory consumption despite having no exact queries. CATCHUp offers competitive query times for exact results while keeping memory consumption reasonable. TD-CRP offers even lower memory consumption. However, this is only possible through the use of approximation. TD-CRP queries depict a noticeable error and perform somewhat worse than KaTCH or CATCHUp queries. TD-S+9 depicts the smallest average error of all non-exact approaches[5].

[4] KaTCH: `https://github.com/GVeitBatz/KaTCH`
TD-S: `https://github.com/ben-strasser/td_p`

[5] [Kon+17] reported another CFLAT configuration with even smaller errors but significantly slower queries.

Table 8.4: Comparison with related work on Ger06. We list unscaled numbers as reported in the respective publications for algorithms we could not run ourselves. Values not reported are indicated as n/r. A similar overview with scaled numbers can be found in [BDPW16].

	Preprocessing			Query		
	Time	Cores	Data	Time	Rel. error	
	[s]		[GB]	[ms]	Avg. [%]	Max. [%]
TD-Dijkstra	–	–	–	719.26	–	–
TDCALT [DN12]	540	1	0.23	5.36	–	–
TDCALT-K1.15 [DN12]	540	1	0.23	1.87	0.050	13.840
eco L-SHARC [Del11]	4 680	1	1.03	6.31	–	–
heu SHARC [Del11]	12 360	1	0.64	0.69	n/r	0.610
KaTCH	169	16	4.66	0.64	–	–
TCH [BGSV13]	378	8	4.66	0.75	–	–
ATCH (1.0) [BGSV13]	378	8	1.12	1.24	–	–
ATCH (∞) [BGSV13]	378	8	0.55	1.66	–	–
inex. TCH (0.1) [BGSV13]	378	8	1.34	0.70	0.020	0.100
inex. TCH (1.0) [BGSV13]	378	8	1.00	0.69	0.270	1.010
TD-CRP (0.1) [BDPW16]	289	16	0.78	1.92	0.050	0.250
TD-CRP (1.0) [BDPW16]	281	16	0.36	1.66	0.680	2.850
FLAT [Kon+17]	158 760	6	54.63	1.27	0.015	n/r
CFLAT [Kon+17]	104 220	6	34.63	**0.58**	0.008	0.918
TD-S+9	542	1	3.61	2.07	0.001	1.523
CH-Potentials	57	1	**0.21**	4.36	–	–
CATCHUp	**52**	16	1.06	0.72	–	–

Path retrieval in the time-dependent scenario is not as easy as in the static setting. Table 8.4 reports running times for the earliest arrival query and the path retrieval combined. We only have separate numbers for KaTCH and CATCHUp. For CFLAT, [Kon+17] reported that path retrieval takes a third of the total query time. Our experiments show a similar amount for KaTCH. For CATCHUp, path retrieval takes up less than 10% of the query time. TD-CRP and FLAT do not support path retrieval.

In Table 8.5, we report results for the newer instances for all algorithms were we could an implementation and perform experiments. Compared to the Ger06 instance, the Ger17 network provides a significantly harder challenge, despite being only slightly larger. This is because the amount of and the complexity of the time-dependent travel times (compare Table 4.2). KaTCH query times increase by a factor of about two. However, memory usage grows by almost an order of magnitude. For TD-S, both the growth in space consumption and query

Table 8.5: Comparison with related work on newer instances for algorithms were we could obtain source code and run them ourselves. OOM means that the program crashed while trying to allocate more memory than available.

		Preprocessing			Query		
		Time	Cores	Data	Time	Rel. error	
		[s]		[GB]	[ms]	Avg. [%]	Max. [%]
Ger17	TD-Dijkstra	–	–	–	814.60	–	–
	KaTCH	859	16	42.81	**1.26**	–	–
	TD-S+9	601	1	5.28	2.61	0.001	0.963
	CH-Potentials	**64**	1	**0.30**	16.14	–	–
	CATCHUp	142	16	1.50	2.02	–	–
Eur17	TD-Dijkstra	–	–	–	2 929.72	–	–
	KaTCH	3 066	16	146.97	OOM	–	–
	TD-S+9	3 149	1	18.84	**4.70**	0.002	1.159
	CH-Potentials	**340**	1	**1.08**	81.09	–	–
	CATCHUp	747	16	5.47	4.92	–	–
Ger19	TD-Dijkstra	–	–	–	2 564.18	–	–
	KaTCH	> 24 h	16	> 314.00	–	–	–
	TD-S+9	2 538	1	11.83	**4.27**	0.027	4.953
	CH-Potentials	**291**	1	**0.68**	128.61	–	–
	CATCHUp	11 717	16	4.30	16.48	–	–
Eur20	TD-Dijkstra	–	–	–	3 784.11	–	–
	KaTCH	7 149	16	239.78	OOM	–	–
	TD-S+9	3 352	1	20.65	**4.23**	0.006	1.733
	CH-Potentials	**365**	1	**1.19**	104.47	–	–
	CATCHUp	1 249	16	6.32	5.60	–	–

times corresponds roughly to the growth of the graph size, but not to the increased number of breakpoints. CH-Potentials preprocessing and space consumption is affected only slightly but query times quadruple. The space consumption of CATCHUp grows by a similar factor. Query times get about 2.7 times slower.

On Eur17, the memory consumption of KaTCH becomes prohibitive. While KaTCH is still able to finish preprocessing and generade 150 GB of data, queries crash since the 192 GB RAM of our machine are not enough. Using ATCH or inexact TCH, the memory consumption could likely be reduced sufficiently to perform queries. However, this would either introduce errors or slow down queries significantly. With only 5.5 GB of auxiliary data, CATCHUp is still able to

perform exact queries in less than 5 ms on average. This is fast enough to enable interactive applications. Total preprocessing for CATCHUp takes less than a quarter of the time KaTCH needs. TD-S+9 is also able to handle this instance with similar query times but only with a small average error. CH-Potentials features the fastest preprocessing and the smallest space consumption but query times start to become problematic.

The Ger19 instance is the hardest instance in terms of amount of time-dependent information and function complexity. Preprocessing times and space consumption of TD-S and CH-Potentials are robust against this because these techniques completely avoid time-dependency in the preprocessing. However, this is only possible at the cost of slow queries (CH-Potentials) or inexact results (TD-S). On this instance, TD-S errors also become more noticeable. KaTCH is not able to even finish its preprocessing in reasonable time on this instance: We terminated the preprocessing after a day. At this point, 314 GB of data had already been written to the disk. Thus, performing queries would be infeasible even if the preprocessing finished. CATCHUp is robust enough to handle this instance and achieve interactive running times for exact queries. The space consumption also remains low. The auxiliary data needs only a third of disk space as the input graph. Nevertheless, preprocessing takes a little more than three hours which is relatively slow compared to the times on other instances. These results clearly demonstrate the robustness of CATCHUp even on very hard instances.

On Eur20, the behavior is similar to Eur17, only more extreme. Despite the amount of time-dependent information in the instance, the functions are far more well-behaved than on Ger19. KaTCH preprocessing time takes about two hours and the space overhead is around 240 GB. The TD-S+9 numbers remain relatively stable compared to Eur17. Query times get slightly faster but errors become larger. CH-Potentials has fast preprocessing and and requires little space but has slow queries. Compared to Eur17, CATCHUp preprocessing times become slower but by less than a factor of two. Query times increase to 5.6 ms. The space overhead is only 6.3 GB, which is smaller than the input network.

8.5 Conclusion

We introduce CATCHUp, a speedup technique for routing in time-dependent road networks. It features a small space overhead and fast, exact queries. To the best of our knowledge, our approach is the first to simultaneously achieve all three objectives. We perform an extensive experimental study to evaluate the performance of CATCHUp and compare it to competing approaches. Our approach achieves fast preprocessing, interactive query running times and uses up to 38 times less space than other algorithms with competitive query performance. This demonstrates the advantage of storing expansion information instead of travel time functions.

Revisiting ATCH, TCH, and TD-CRP with the insights gained in this work could be fruitful. Combining ATCH with our A* query extension could reduce ATCH query running times. CATCHUp makes use of travel time independent vertex orders. Combining CATCHUp with TCH-like vertex orders could result in further reduced space consumption and faster query running times.

9 Combining Predicted and Live Traffic with Time-Dependent A* Potentials

In this chapter, we turn to the combination of predicted and real-time traffic, i.e. we aim to efficiently and exactly solve the three-phase combined traffic shortest path problem. CH-Potentials, introduced in Chapter 6, can be applied in this model. However, the achieved speedups are too small for interactive query running times. The CCH framework, discussed in Chapter 7, has very fast queries but cannot deal with time-dependent travel times. CATCHUp, introduced in Chapter 8, achieves fast queries in a time-dependent setting and even has a customization algorithm, as its CCH-based. However, the CATCHUp customization is too slow to apply it frequently in an update phase. Further, this would only allow updates to the predictions instead of a combination of both traffic types.

Therefore, approaching the combined traffic model is only possible with a careful combination of these approaches. In the CH-Potentials chapter, we showed how A* allows us to decouple the problem to be solved from the speedup technique used for acceleration. The only problem with this approach is that its performance is limited by the tightness of the preprocessing lower bound used for the heuristic. In this chapter, we extend and improve this approach for time-dependent routing by introducing *time-dependent A* potentials*. These allow us to obtain tighter potentials and thus faster queries in time-dependent problems and variations thereof such as the combined traffic problem.

Attribution. This chapter is based on joint work with Nils Werner. The results have been published at ESA 2022 [WZ22]. Nils Werner developed a preliminary version of time-dependent A* potentials throughout his Master's thesis and applied them in a different context, i.e. cooperative route planning.

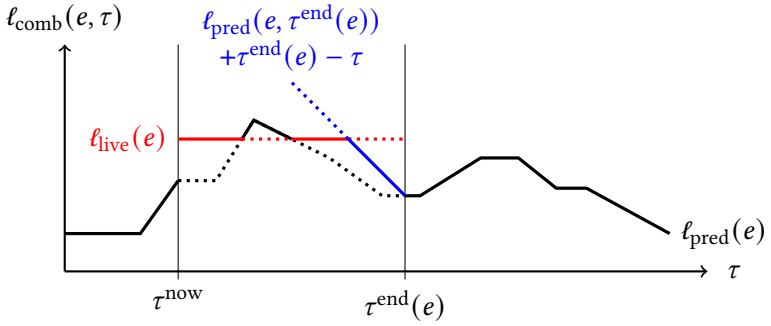


Figure 9.1: Combined travel time function $\ell_{\text{comb}}(e,\tau) = \max(\ell_{\text{pred}}(e,\tau), \min(\ell_{\text{live}}(e), \ell_{\text{pred}}(e,\tau^{\text{end}}(e)) + \tau^{\text{end}}(e) - \tau))$ with both predicted and live traffic information. The predicted traffic $\ell_{\text{pred}}(e)$ is indicated in black. The live travel time $\ell_{\text{live}}(e)$ with expected end $\tau^{\text{end}}(e)$ is depicted in red. The switch back to the predicted function is colored in blue. The solid line indicates the combined function $\ell_{\text{comb}}(e)$ for the current day. For later days, only ℓ_{pred} will be used. Dotted lines only serve the purpose of visualization.

Contribution. We introduce a time-dependent generalization of A* potentials and present two Lazy RPHAST extensions that realize a time-dependent potential function. Further, we show how to apply them to queries in a setting that combines live and predicted traffic. An extensive evaluation confirms the effectiveness of our potentials. Queries incorporating both predicted and current traffic can be answered within few tens of milliseconds. Live traffic updates can be integrated within a fraction of a minute. Our time-dependent potentials are up to an order of magnitude faster than CH-Potentials and about two orders of magnitude faster than Dijkstra's algorithm. To the best of our knowledge, this makes our approach the first to achieve interactive query performance while allowing fast updates in this setting.

Outline. This chapter is organized as follows. First, in Section 9.1, we reconsider the problem model and present a slightly refined version of it. Section 9.2 introduces time-dependent A* potentials and investigates theoretic properties necessary for correctness. Then, two Lazy RPHAST-based practically efficient realizations are presented in Section 9.3. A thorough practical evaluation is given in Section 9.4.

9.1 Model Refinement

We consider a variation of the three-phase combined traffic shortest path problem. As before, traffic predictions are FIFO PPLFs with an horizon H of one day. The difference compared to the model as we defined it in Section 3.3 is the way we defined the combined travel time functions. We assume that traffic predictions are conservative estimates and that live traffic will only be slower than the predicted traffic due to accidents and other traffic incidents, i.e. $\ell_{\text{pred}}(e,\tau^{\text{now}}) \leq \ell_{\text{live}}(e)$. Therefore, we define the combined travel time function $\ell_{\text{comb}}(e,\tau) =$

$\max(\ell_{\text{pred}}(e,\tau), \min(\ell_{\text{live}}(e), \ell_{\text{pred}}(e, \tau^{\text{end}}(e)) + \tau^{\text{end}}(e) - \tau))$. It follows that $\ell_{\text{pred}}(e,\tau) \leq \ell_{\text{comb}}(e,\tau)$. Figure 9.1 depicts an example of such a combined travel time function.

The modelling assumption that predicted traffic is a lower bound of real-time traffic is, of course, a severe restriction from a theoretical perspective. However, we argue that, from a practical perspective, it is only a minor limitation. Live traffic should account for unexpected traffic events which will almost exclusively only make traffic worse. If the live traffic is frequently better than the predicted traffic, the predictions should be adjusted. Thus, the assumption will hold for most of the time on realistic data. Further, the reason to solve routing problems to exactness is not that the computed travel times will perfectly match reality. Traffic data is always only an approximation of reality. Thus, a less flexible model which can be solved efficiently and exactly may sometimes be more practical than a more flexible one that can only be solved heuristically or inefficiently.

9.2 Time-Dependent A* Potentials

We now propose a time-dependent generalization $\pi_t : \mathcal{V} \to (T \to T^{\geq 0})$ of A* potentials, i.e. estimates are a function of the time. This allows us to obtain tighter estimates and enables faster queries. Analogue to classical potentials, there are properties of time-dependent potentials to consider for the correctness of A*:

- Strong First-In First-Out: $\pi_t(v,\tau) < \pi_t(v, \tau+\varepsilon) + \varepsilon$ for $v \in \mathcal{V}$, $\tau > \text{dist}(s, v, \tau^{\text{dep}})$ and $\varepsilon > 0$. This ensures that queue keys increase monotonically with the distance from s. This property has no time-independent equivalent because it holds trivially in this case.

- Feasibility: $\ell(uv,\tau) + \pi_t(v, \tau + \ell(uv,\tau)) - \pi_t(u,\tau) \geq 0$ for all edges $uv \in \mathcal{E}$ and times $\tau > \text{dist}(s, u, \tau^{\text{dep}})$. A* can be analyzed as an equivalent run of Dijkstra's algorithm with a modified weight function derived from the input weights and the potentials. With feasibility, these modified weights are non-negative, which implies correctness and polynomial running time. When $\pi_t(t,\tau) = 0$, feasibility also implies the lower bound property. However, feasibility is not strictly necessary to guarantee correctness.

- Lower bound: $\pi_t(v,\tau) \leq \text{dist}(v,t,\tau)$ for every vertex $v \in \mathcal{V}$ and time $\tau = \text{dist}(s, v, \tau^{\text{dep}})$. This ensures that the search has found the correct distance once the target vertex is settled. This is also sufficient for correctness. However, without feasibility, A* may settle vertices multiple times. In theory, this can lead to an exponential running time.

In the following, we formally justify these properties and provide correctness proofs for A* with time-dependent potentials adhering to them. Here, it is often more practical to use arrival time functions $\hat{\ell}$ instead of travel time functions ℓ. With arrival time functions, we can represent path lengths simply as the composition of the arrival time functions of the individual edges. We use $(f \circ g)(\tau) = f(g(\tau))$ to denote the function composition.

Similarly to the time-independent case, we can define a modified weight function ℓ_{π_t} such that running A* on the graph G with weights ℓ with a time-dependent potential π_t is equivalent to running Dijkstra's algorithm on G with modified weights ℓ_{π_t}. Consider a vertex u with an arrival time of τ. In A*, its queue key is $\tau' = \pi_t(u,\tau) + \tau = \hat{\pi}_t(u,\tau)$. For the equivalent run of Dijkstra's algorithm, we want the distances to equal these queue keys. Therefore, the reduced weight function of edge uv must compose the original function with the potential at the head vertex v, i.e. $\hat{\pi}_t(v, \hat{\ell}(uv,\tau)) = (\hat{\pi}_t(v) \circ \hat{\ell}(uv))(\tau)$. However, the input of ℓ is a time τ, not the A* queue key $\tau' = \hat{\pi}_t(u,\tau)$. To first go back from τ' to τ, we also need to compose this with the inverted potential function $\hat{\pi}_t(u)^{-1}$. Thus, the modified weight $\hat{\ell}_{\pi_t}$ are defined as $\hat{\ell}_{\pi_t}(uv,\tau') = \hat{\pi}_t(v, \hat{\ell}(uv, \hat{\pi}_t(u)^{-1}(\tau'))) = (\hat{\pi}_t(v) \circ \hat{\ell}(uv) \circ \hat{\pi}_t(u)^{-1})(\tau')$.

For $\hat{\pi}_t(u)^{-1}$ to be well-defined, we need $\tau_1 \neq \tau_2 \implies \hat{\pi}_t(\tau_1) \neq \hat{\pi}_t(\tau_2)$. Therefore, for potentials, we require the *strong First-In First-Out* property:

$$\pi_t(v,\tau) < \pi_t(v,\tau+\varepsilon) + \varepsilon, v \in \mathcal{V}, \varepsilon > 0$$

Note that potentials that only adhere to the regular FIFO property might also work in practice. However, in this case, the modified weights for a theoretically equivalent run of Dijkstra's algorithm are not well-defined anymore. This breaks the following correctness argument. Nevertheless, a more sophisticated analysis could probably work around this problem.

Shortest paths for the modified weights ℓ_{π_t} are the same as with the original weights ℓ. Consider a path $P = (s, v_1, \ldots, v_k, t)$. By definition, the arrival time function of a path is

$$\hat{\ell}_{\pi_t}(P) = \hat{\pi}_t(t) \circ \hat{\ell}(v_k t) \circ \hat{\pi}_t(v_k)^{-1} \circ \hat{\pi}_t(v_k) \circ \cdots \circ \hat{\pi}_t(v_1)^{-1} \circ \hat{\pi}_t(v_1) \circ \hat{\ell}(sv_1) \circ \hat{\pi}_t(s)^{-1}$$

Note that all the inner potential evaluations cancel out. Besides the initial and final potential evaluations, only the composition of the original weights remains. Now consider the modified weights of two different st paths. With a fixed departure, the initial inverted potential at s is the same for both paths. Thus, the original weights determine which path is shorter. The final potential evaluation at t cannot change the relative order due to the FIFO property. Therefore, shortest paths for ℓ_{π_t} are the same as for ℓ.

We can infer that if these modified weights do not have any negative travel times, then Dijkstra and thus A* will obtain correct results. This leads to the *feasibility* property for time-dependent potentials:

$$\hat{\ell}_{\pi_t}(uv,\tau') = \hat{\pi}_t(v, \hat{\ell}(uv, \hat{\pi}_t(u)^{-1}(\tau'))) \geq \tau'$$

To simplify correctness proofs for potentials, we reformulate this to a simpler equivalent property:

$$\begin{aligned} \hat{\pi}_t(v, \hat{\ell}(uv,\tau)) &\geq \hat{\pi}_t(u,\tau) \\ \pi_t(v, \ell(uv,\tau)+\tau) + \ell(uv,\tau) + \tau &\geq \pi_t(u,\tau) + \tau \\ \pi_t(v, \ell(uv,\tau)+\tau) + \ell(uv,\tau) - \pi_t(u,\tau) &\geq 0 \end{aligned}$$

This formulation also shows that the time-dependent feasibility is a generalization of the classical feasibility property $\ell(uv) + \pi_t(v) - \pi_t(u) \geq 0$.

The third property is the *lower bound* property:

$$\pi_t(v, \tau) \leq \text{dist}(v, t, \tau)$$

With a potential fulfilling this property, A* is guaranteed to have found the shortest path once the target vertex is settled, even if the potential is not feasible. This is because every vertex on the shortest path must have a queue key less or equal to the target when discovered with optimal distance. Suppose for contradiction the target vertex would be settled with distance d which is greater than the shortest distance. But because of the lower bound property, every vertex on the shortest path will have a queue key smaller than d as soon as it was discovered with its shortest distance. By induction, this must happen for all vertices on the shortest path. Thus, also the shortest path to t must have been found. This contradicts our assumption.

Even with negative travel times due to infeasible potentials, negative cycles are not possible. Consider a cycle C starting and ending at vertex v. The length of the cycle with the modified weights is $\ell_{\pi_t}(C, \tau') = \hat{\pi}_t(v, \hat{\ell}(C, \hat{\pi}_t(u)^{-1}(\tau')))$ because all inner potential functions cancel out. As $\hat{\pi}_t(v)$ has to adhere to the FIFO property, the length of C cannot be negative. We conclude that the distance at t is final as soon as t is settled. Thus, A* with lower bound potentials will obtain correct results. However, the running time may be exponential in the graph size.

Crucially, to guarantee correctness, potentials need not adhere to these properties for *all* points in time τ. Assume we are running A* to answer a query from s to t with departure τ^{dep}. Clearly, A* will never invoke the potential $\pi_t(v, \tau)$ of a vertex v with $\tau < \text{dist}(s, v, \tau^{\text{dep}})$. Therefore, it is sufficient to guarantee the strong FIFO property at vertex v and the feasibility property for edge vw for $\tau \geq \text{dist}(s, v, \tau^{\text{dep}})$. For the lower bound property it even suffices to guarantee it only at exactly $\tau = \text{dist}(s, v, \tau^{\text{dep}})$. When a vertex is discovered with a suboptimal distance, the queue key may be arbitrarily larger but never smaller because of the FIFO property. Therefore, the inductive argument still applies. All vertices on the shortest path will be traversed before the target is settled. Our practical potentials heavily rely on this and only compute data for the specific times necessary to answer a query correctly.

9.3 Lazy RPHAST-based Time-Dependent Potentials

In the following, we present two practical realizations of time-dependent A* potentials. Both are extensions of Lazy RPHAST. Lazy RPHAST/CH-Potentials is already a very efficient potential and obtains exact distances for scalar lower bound weights, i.e. the tightest possible estimates with a time-independent potential definition. To outperform CH-Potentials, on the one hand, we have to obtain significantly tighter estimates. On the other hand, we also must avoid the potential evaluation becoming too expensive. Therefore, we avoid costly operations on functions and work with scalar values as much as possible. As a result, even though our potentials are time-dependent, computed estimates during a single query usually will *not* change depending on the visit time of a vertex.

9.3.1 Multi-Metric Potentials

Let $(s, t, \tau^{\text{dep}})$ be a query and $\tau^{\max}$ an upper bound on the optimal arrival time at the target. Consider any $\tau' \leq \tau^{\text{dep}}$, $\tau^{\max} \leq \tau''$ and the weight function $l[\tau', \tau''](e) := \min_{\tau' \leq \tau \leq \tau''} \ell_{\text{pred}}(e, \tau)$. Clearly, $\text{dist}_{l[\tau', \tau'']}(v, t)$ provides lower bound estimates of distances to the target vertex during the time relevant for this query. If τ' and τ'' are close to τ^{dep} and $\tau^{\max}$ and, if the difference between τ' and τ'' is not too big, the estimates will be significantly tighter than global lower bound distances. The *Multi-Metric Potentials* (MMP) approach is based on this observation. Instead of using a single potential based on a global lower bound valid for the entire time, we process multiple lower bound weight functions for different time intervals. At query time, we then select an appropriate weight function. The upper bound $\tau^{\max}$ is computed with a time-independent CCH query on a scalar upper bound function $\overline{\ell}^{+}_{\text{comb}}$ computed during the update phase. Efficiently computing distances with respect to the selected weight function is done with Lazy RPHAST. Therefore, no time-dependent computations need to be performed to evaluate this potential function. MMP only depend on the departure time of the query but not of the potential evaluation time. Still, MMP will be significantly tighter than any time-independent potential can be.

Phase Details. The first step of the preprocessing for this potential is to perform the regular CCH preprocessing, i.e. compute an importance ordering and construct the unweighted augmented graph. Now let $\mathcal{I}$ be a set of time intervals. In our implementation, we cover the time between 6:00 and 22:00 with intervals of a length of one, two, four, and eight hours, starting every 30 minutes, and one interval covering the entire day. We do not maintain any additional intervals between 22:00 and 6:00 as most edge weights correspond to their respective free-flow travel time during this period. Thus, the lower bound weights would be equal to the full-day lower bounds. During preprocessing, for each interval $[\tau'_i, \tau''_i] \in \mathcal{I}$, we extract lower bound functions $l[\tau'_i, \tau''_i]$ and run the CCH customization algorithm to obtain $l[\tau'_i, \tau''_i]^+$. This can be parallelized trivially. Also, the customization can be parallelized internally. For further engineering details, we refer to Chapter 7.

During the update phase, we compute an additional lower bound weight function starting at τ^{now} with duration λ derived from the combined weights ℓ_{comb} and run the basic customization for it. We use $\lambda = 59$ minutes to reasonably cover the live traffic but keep the live interval shorter than any other interval. Further, we extract an upper bound weight function $\overline{\ell}_{\text{comb}}$ which is valid for the entire day for both the predicted and the live traffic, and perform the CCH basic customization to obtain $\overline{\ell}^{+}_{\text{comb}}$.

The query starts with a classical CCH query on the customized upper bound $\overline{\ell}^{+}_{\text{comb}}$ to obtain a pessimistic estimate of $\tau^{\max}$. We then select the smallest interval $[\tau'_i, \tau''_i]$ such that $[\tau^{\text{dep}}, \tau^{\max}] \subseteq [\tau'_i, \tau''_i]$. Running Lazy RPHAST on G^+ with the customized weight function $l[\tau'_i, \tau''_i]^+$ yields the desired potential function.

Correctness. For any given single query, the estimates obtained by MMP are time-independent. They return the exact shortest distances with respect to a lower bound weight function valid for the query. Constant potentials trivially adhere to the strong FIFO property. Also, as shown in Chapter 6, shortest distances for a lower bound are feasible potentials.

9.3.2 Interval-Minimum Potentials

Interval-Minimum Potentials (IMP) is a time-dependent adaptation of the Lazy RPHAST algorithm. While Lazy RPHAST has a single scalar weight for each edge, the Interval-Minimum Potential uses a time-dependent function. This allows for tighter estimates but introduces new challenges. First, we need an augmented graph with sufficiently accurate time-dependent lower bounds. We utilize the existing CATCHUp customization from the previous chapter because it is based on CCH. Second, storing the shortcut travel time functions ℓ^+ may consume a lot of memory. Further, the representation as a list of breakpoints makes the evaluation more expensive than looking up a scalar weight. Therefore, we resort to a different representation and store functions as piecewise constant values in buckets of equal duration. Third, evaluating these functions requires a time argument. While $\pi_t(v, \tau)$ includes the time argument τ for the time at v, Lazy RPHAST also needs a time for every recursive invocation. Therefore, we apply Lazy RPHAST a second time on global upper and lower bound weight functions $\overline{\ell}^+_{\text{comb}}$ and $\underline{\ell}^+_{\text{pred}}$ to quickly obtain arrival intervals for arbitrary vertices. We then use these intervals to evaluate the edge weights and obtain tight time-dependent lower bounds.

Phase Details. The first preprocessing step is the CCH preprocessing. For the second step, we need to obtain time-dependent travel times for the augmented graph G^+ based on the predicted traffic weights ℓ_{pred}. For this, we utilize the CATCHUp customization as described in Section 8.2. Recall that the CATCHUp customization maintains for each edge in $uv \in \mathcal{E}^+$ approximated *time-dependent* lower bound functions $\ell^+_{\text{lb}}(uv)$ (see Section 8.2.4). Before we drop these functions in the CATCHUp customization, we extract from them piecewise *constant* lower bound functions $\ell^+_{\text{pclb}}(uv, \tau) := \min\left\{\ell^+_{\text{lb}}(uv, \tau') \mid \gamma\lfloor\frac{\tau}{\gamma}\rfloor \le \tau' < \gamma(\lfloor\frac{\tau}{\gamma}\rfloor + 1)\right\}$ where γ is the length of each constant segment. This enables a compact representation. Functions can be represented with a fixed number of values per edge. We use 96 buckets of length $\gamma = 15$ minutes. Additionally, we store the scalar shortcut bounds $\underline{\mathfrak{b}}$ maintained throughout the CATCHUp customization as $\underline{\ell}^+_{\text{pred}}$.

In the update phase, we extract a combined traffic upper bound weight function $\overline{\ell}_{\text{comb}}$ for the entire day and run the CCH customization to obtain $\overline{\ell}^+_{\text{comb}}$.

The query consists of two instantiations of the Lazy RPHAST algorithm. The first one uses the scalar bounds $\underline{\ell}^+_{\text{pred}}$ and $\overline{\ell}^+_{\text{comb}}$ and computes an interval of possible arrival times at arbitrary vertices when departing from s at τ^{dep}. Since arrival intervals are distances from the source vertex, we have to apply Lazy RPHAST in reverse direction. This means we first run Dijkstra's algorithm from s on $G^{\uparrow}$, and then, we apply the recursive distance-memoizing DFS on $G^{\downarrow}$ for any vertex for which we want to obtain an arrival interval. We denote this

instance as *Arrival Interval Lazy RPAHST* (AILR). With these arrival intervals, we can now compute lower bounds to the target with the second Lazy RPHAST instantiation, which uses the time-dependent lower bounds ℓ^+_{pclb}. The first step is to run Dijkstra's algorithm from t on $G^{\downarrow}$. To relax an edge $uv \in \overleftarrow{\mathcal{E}}^{\downarrow}$, we first need to obtain an arrival interval $[\tau_{\min}, \tau_{\max}]$ at v using AILR. This allows us to determine for vu at the relevant time a tight lower bound $d := \min_{\tau \in [\tau_{\min}, \tau_{\max}]} \ell^+_{\text{pclb}}(vu, \tau)$. Then, we check if we can improve the lower bound from v to t, i.e. $\mathtt{D}^{\downarrow}[v] \leftarrow \min(\mathtt{D}^{\downarrow}[v], \mathtt{D}^{\downarrow}[u] + d)$. Having established preliminary backward distances for all vertices in the CH search space of t, we can now compute estimates with the recursive distance-memoizing DFS. To obtain a distance estimate for vertex u, we first recursively compute distance estimates $\mathtt{D}[v]$ for all upward neighbors v where $uv \in \mathcal{E}^{\uparrow}$. Then, we use AILR to obtain an arrival interval $[\tau_{\min}, \tau_{\max}]$ at u. Finally, after initializing $\mathtt{D}[u] \leftarrow \mathtt{D}^{\downarrow}[u]$, we relax all upward edges uv and set $\mathtt{D}[u] \leftarrow \min(\mathtt{D}[u], \mathtt{D}[v] + \min_{\tau \in [\tau_{\min}, \tau_{\max}]} \ell^+_{\text{pclb}}(uv, \tau))$. This yields the final estimate for u.

Choosing a good memory layout for the bucket weights is crucial for the performance. We store all edge weights of each bucket consecutively. Typically, only a few buckets per edge are relevant because the arrival intervals are relatively small. Also, all outgoing edges of each vertex are evaluated consecutively. Thus, having their weights for the same bucket close to each other increases cache hits.

Correctness. Estimates obtained by IMP are lower bounds of the actual time-dependent shortest distances. This directly follows from the correctness of the CATCHUp preprocessing and the Lazy RPHAST algorithm. Also, they do satisfy the strong FIFO property because, for any given single query, the estimates are constant. However, they are not feasible due to the piecewise constant approximation schema. We could not observe any practical negative consequences of this, though.

9.3.3 Optimizations

Perfect Customization. So far, we have described our potentials in terms of the CCH augmented graph G^+ which is valid for any length function. However, as described in Section 8.2.2, this graph contains many unnecessary edges. We can identify and remove some of these by utilizing the perfect customization on appropriate upper bounds.

During the update phase, we also run the perfect customization for the upper bound and obtain $\overline{\ell}^*_{\text{comb}}$. This allows us to remove unnecessary edges and accelerate the query phase. We remove edges uv where $\text{dist}^{<}_{\text{comb}}(u, v) \geq \underline{\ell}^+(uv) > \overline{\ell}^*_{\text{comb}}(uv) \geq \text{dist}_{\text{comb}}(u, v)$. This condition is the same for both MMP and IMP. For MMP, we could remove even more edges if we were to check the condition for every lower bound weight function individually. However, in this case, we could not reuse the same reduced augmented graph topology for all weight functions. As this would roughly double memory consumption, we only remove edges that can be removed for all weight functions. We parallelize the reduced graph constructions as described in Section 7.2.2.

Metric Switching. The tightness of MMP can be improved further by switching to weight functions for smaller intervals as the query progresses. When initializing the potential, we select, beside $[\tau'_i, \tau''_i] \supseteq [\tau^{\text{dep}}, \tau^{\max}]$, additional intervals $[\tau'_j, \tau''_j] \subset [\tau'_i, \tau''_i]$ with $\tau''_j \geq \tau^{\max}$. When evaluating the potential of a vertex at instant τ, we use the weight function with the shortest associated interval that still includes τ. However, running Lazy RPHAST separately for each of these weight functions would be too expensive. Therefore, we modify the DFS part of Lazy RPHAST to track, besides the distances $\texttt{D}[u]$, also the weight function $\texttt{W}[u]$ which was used. Then, an already memoized distance $\texttt{D}[v]$ for vertex v can be reused if the interval associated with $\texttt{W}[v]$ is a superset of the best interval for τ. We pass the τ parameter unmodified to recursive invocations. Thus, we maintain the invariant that if $\texttt{D}[v]$ is final and valid for τ, also all vertices in the CH search space of v will have a valid lower bound for an interval that includes τ. Therefore, we will only underestimate lower bounds. The lower bound property is preserved. The strong FIFO property is preserved, too. Potential functions are now piecewise constant with only increasing jumps. However, this breaks the feasibility property. We could not observe any practical negative consequences of this, though.

For IMP, we employ a similar approach. We use the potential evaluation time argument τ to tighten the arrival intervals and reduce the number of buckets to look up. When evaluating the potential of a vertex v at instant τ, we still obtain the arrival interval $[\tau_{\min}, \tau_{\max}]$ using the AILR. Typically, τ will be greater than $\tau_{\min}$. Therefore, we can avoid looking up unnecessary bucket entries by only evaluating $[\max(\tau_{\min}, \tau), \tau_{\max}]$. We track the respective bucket of τ in $\texttt{B}[v]$. When the potential is evaluated again with a possibly smaller τ, we check if the respective bucket is smaller than $\texttt{B}[v]$ to determine if the memoized distance can be reused. If not, we reevaluate outgoing edges with the additional buckets and update $\texttt{B}[v]$ accordingly. We pass the τ parameter unmodified to recursive invocations.

9.3.4 Compression

Both of our time-dependent potentials use many weight functions. This can lead to problematic memory consumption. However, since we only need lower bounds, we can merge weight functions. Consider two MMP intervals with weight functions l_1 and l_2. A combined function $l_{1\cup 2}(uv) = \min(l_1(uv), l_2(uv))$ is valid for both intervals, albeit less tight. We can merge IMP buckets analogously. Thus, we can reduce memory consumption by trading tightness. Both potentials can handle merged lower bound functions with a layer of indirection: Buckets and intervals are mapped to a weight function ID. The weight of an edge in a merged weight function is the minimum weight of this edge in all included functions.

We now discuss an efficient and well-parallelizable algorithm to iteratively merge weight functions until only k functions remain. In each step, we merge the pair of weight functions with the minimal sum of squared differences of all edge weights. Since comparing all pairs of weight functions is expensive, we track the minimum difference sum $\Delta_{\min}$ we have found so far and stop any comparison where the sum exceeds $\Delta_{\min}$. However, even when stopping a comparison, we store the preliminary sum and the edge ID up to which we have summed up

the differences. Then, we do not need to start from scratch should we continue to compare this particular pair of weight functions. Finally, we maintain all pairs of weights along with the (possibly preliminary) difference sums in a priority queue ordered by the difference sums. When merging two weight functions, all other associated queue entries are removed from the queue and new entries for comparisons between the new weight function and all other functions are inserted. To determine the next weight function pair to merge, unfinished weight function pairs are popped from the queue and processed in parallel. The minimum difference is tracked in an atomic variable.

9.4 Evaluation

Environment. Our benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has 8 cores clocked at 3.5 GHz and 8 × 64 KiB of L1, 8 × 1 MiB of L2, and 24.75 MiB of shared L3 cache. Hyperthreading was disabled and parallel experiments use 16 threads. We implemented our algorithms in Rust[1] and compiled them with `rustc 1.61.0-nightly (c84f39e6c 2022-03-20)` in the release profile with the `target-cpu=native` option.

Inputs. We evaluate our time-dependent potentials on OSM Germany/Ger19 and Eur20 as these are the networks where we have real-world live traffic snapshots. See Section 4.2.2 and Section 4.2.3 for a discussion of these data sets. For completeness, we also evaluate our time-dependent potentials for only predicted traffic on all other time-dependent instances and report the results in Appendix D.

Methodology. We evaluate our algorithms by sequentially solving batches of 100 k shortest path queries with three different query sets: First, there are *random* queries where source and target are drawn from all vertices uniformly at random. These are mostly long-range queries. Second are *1h* queries where we draw a source vertex uniformly at random, run Dijkstra's algorithm from it and pick the first node with a distance greater than one hour as the target. Third, we generate queries following the Dijkstra rank methodology [SS05] to investigate the performance with respect to query distance. For these *rank* queries, we pick a source uniformly at random and run Dijkstra's algorithm from it. We use every 2^i-th settled vertex as the target for a query of Dijkstra rank 2^i. For queries with only predicted traffic, we pick τ^{dep} uniformly at random. When using live traffic, we set $\tau^{\text{dep}} = \tau^{\text{now}}$. To evaluate the performance of the preprocessing and update phases, we run them 10 and 100 times, respectively. Preprocessing and update phases utilize all cores using 16 threads.

We compare our time-dependent potentials MMP and IMP against time-independent CH-Potentials algorithm realized on CCH. Therefore, we denote this approach as CCH-Potentials. All three potentials use the same CCH vertex order and augmented graph. CCH-Potentials

[1] Our code and experiment scripts are available at `https://github.com/kit-algo/tdpot`

Table 9.1: Query and preprocessing performance results of different potential functions on different graphs and live traffic scenarios. We report average running times, number of queue pops, relative increases of the result distance over the initial distance estimate and speedups over Dijkstra's algorithm for 100 k random queries. Additionally, we report preprocessing and update times and the memory consumption of precomputed auxiliary data.

	Graph	Live traffic	Running time [ms]	Queue [$\cdot 10^3$]	Length incr. [%]	Speedup	Prepro. [s]	Update [s]	Space [GB]
CCH Pot.	Ger	–	137.5	92.3	12.2	24.8	165.2	–	0.8
		10:21	236.5	158.3	18.9	14.7		–	0.8
		15:41	128.0	89.6	19.1	27.0		–	0.8
	Eur	–	102.6	65.2	4.2	58.0	249.7	–	1.0
		07:47	152.2	102.2	8.4	39.3		–	1.0
MMP	Ger	–	117.7	74.6	9.9	29.0	382.6	–	33.7
		10:21	170.0	110.0	13.0	20.4		15.2	34.0
		15:41	119.0	79.5	15.8	29.0		15.3	34.0
	Eur	–	95.3	58.6	3.5	62.5	581.5	–	56.2
		07:47	131.2	84.5	5.8	45.6		22.7	57.2
IMP	Ger	–	22.2	5.1	1.8	154.1	13 687.0	–	30.7
		10:21	29.1	7.6	2.6	119.2		13.5	31.2
		15:41	37.7	11.3	4.2	91.5		13.6	31.2
	Eur	–	11.5	1.8	0.4	518.0	1 799.9	–	52.1
		07:47	25.4	7.4	1.7	235.5		20.1	53.1

provide heuristic estimates based on a lower bound without any real-time or predicted traffic. Thus, no update phase is necessary to integrate real-time traffic updates. It is the only other speedup technique we are aware of that supports exact queries for our problem model. Dijkstra's algorithm without any acceleration is our baseline.

Experiments. In Table 9.1, we report performance results for our time-dependent potentials on random queries. We observe that IMP is the fastest approach by a significant margin, up to an order of magnitude faster than time-independent CCH-Potentials and roughly two orders of magnitude faster than Dijkstra's algorithm. The search space reduction is even greater, but this does not fully translate to running times due to the higher potential evaluation overhead of IMP. With only predicted traffic, IMP is only two to three times slower than CATCHUp. This shows that using A* to gain algorithmic flexibility comes at a cost, but the overhead compared to purely hierarchical techniques is manageable. In contrast, MMP is only slightly faster than CCH-Potentials. This is expected since random queries are mostly long-range for which MMP is not particularly well suited.

Preprocessing times are within a couple of minutes for CCH-Potentials and MMP. IMP prepro-

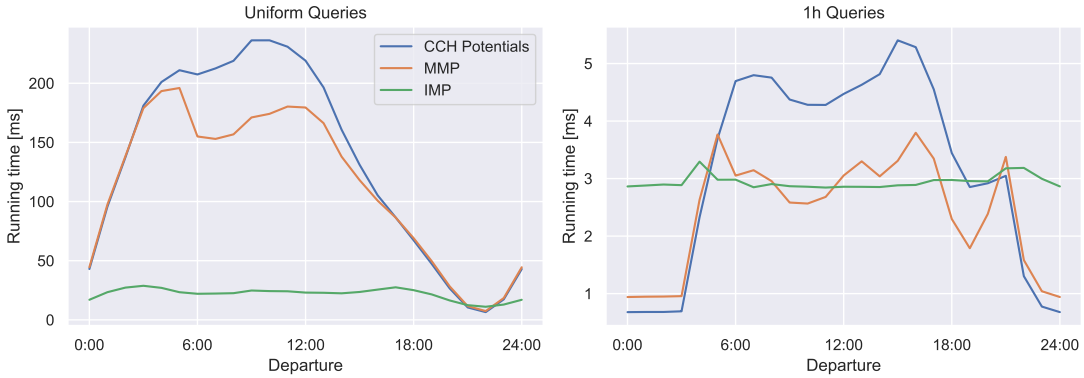


Figure 9.2: Average running time of 100 k uniform and 1h queries on OSM Germany with only predicted traffic. Each query has a departure time drawn uniformly at random. The resulting running times are grouped by the departure time hour.

cessing is significantly more expensive because of the time-dependent CATCHUp preprocessing. This is especially pronounced on OSM Germany where the time-dependent travel time functions fluctuate strongly. Still, running preprocessing algorithms on a daily basis is quite possible. This also underlines that frequently running a CATCHUp customization to include live traffic is not feasible. For both our approaches, real-time traffic updates are possible within a fraction of a minute. MMP is slightly slower because it uses a few more weight functions. Both our approaches are quite expensive in terms of memory consumption, but this can be mitigated through the use of compression (see Figure 9.4).

Introducing live traffic decreases the quality of the estimates and thus increases search space sizes and running times. For IMP, this increases running times by roughly a factor of two. Even with heavy rush hour traffic, IMP is still more than 90 times faster than Dijkstra's algorithm. Surprisingly, for CCH-Potentials and MMP, this scenario seems easier to handle than light midday traffic. This actually is an effect of the *predicted* traffic. It also has a strong influence on the performance of CCH-Potentials and MMP depending on the departure time.

We investigate this behavior with Figure 9.2 which depicts query performance by departure time over the course of the day. Clearly, the departure time has a significant influence both for short-range and long-range queries. For long-range queries, the peaks are shifted and smeared because of the travel time (4–5 hours on average on OSM Germany) covered by the query. This is the reason why the heavy afternoon traffic appears to be easier than the light midday traffic for MMP and CCH-Potentials. For IMP, the influence of the departure time is much smaller, which makes it consistently the fastest approach on long-range queries. For short-range queries, the overhead of IMP make it the slowest during the night. Moreover, MMP is roughly as fast as IMP for 1h queries during the daytime. Therefore, MMP may actually be a simple and effective approach for practical applications where short-range queries are more prominent.

Figure 9.3 depicts the performance by query distance. For short-range queries, IMP is slower than the other approaches because the potential is expensive to evaluate, but it scales much better to long-range queries because of its estimates are tighter. Also, the variance in running times is significantly smaller. Even for rank 2^{24}, most queries can be answered within a few tens of milliseconds. Nevertheless, MMP is actually faster on most ranks. Only at rank 2^{24}, MMP running times become as slow as the CCH-Potentials baseline. A jump in MMP running times can be observed from rank 2^{23} to 2^{24}. This is because the mean query distance jumps from five to six hours on rank 2^{23} to over eight hours on rank 2^{24}, which is longer than the longest covered interval. Thus, on rank 2^{24}, MMP fall back to classical CCH-Potentials on many queries. We also observe a few strong outliers. This happens because of blocked streets in the live traffic data. When the target vertex of a query is only reachable through a blocked road segment, A* will traverse large parts of the networks until the blocked road opens up. This affects all three potentials in the same way and demonstrates an inherent weakness of A*-based approaches: the performance always depends on the quality of the estimates. However, on realistic instances, the time-dependent preprocessing algorithms of purely hierarchical approaches are too expensive for frequent rerunning. This makes our approach the first to enable interactive query times across all distances in a setting with combined live and predicted traffic.

Finally, Figure 9.4 showcases the effects of reducing the number of weight functions. MMP appears to be very robust against compression. We can reduce the number of weight functions to 16 (a memory usage reduction of about a factor of 6) before the slowdowns become noticeable in the mean running time. However, MMP only achieves relatively small speedups

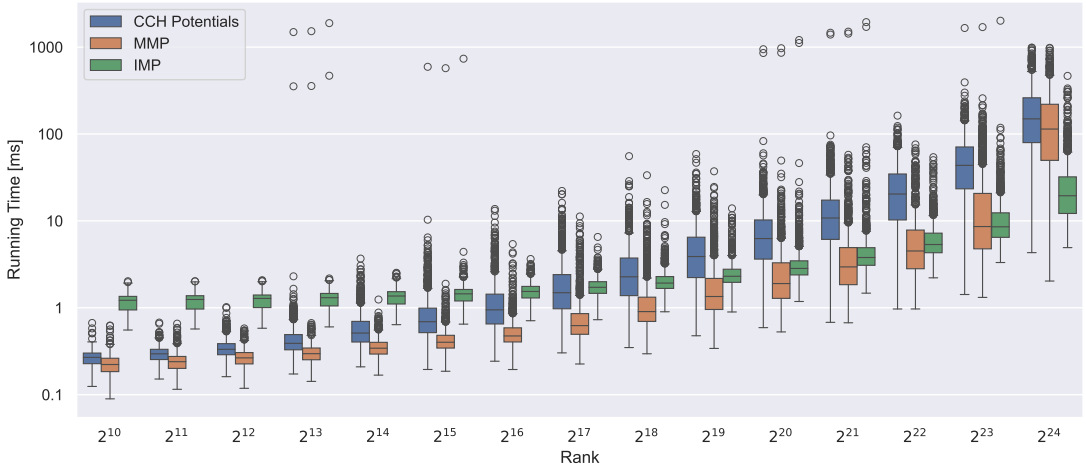


Figure 9.3: Box plot of running times for 1 000 queries per Dijkstra-rank on PTV Europe with live traffic and fixed departure at 07:47. The boxes cover the range between the first and third quartile. The band in the box indicates the median; the whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

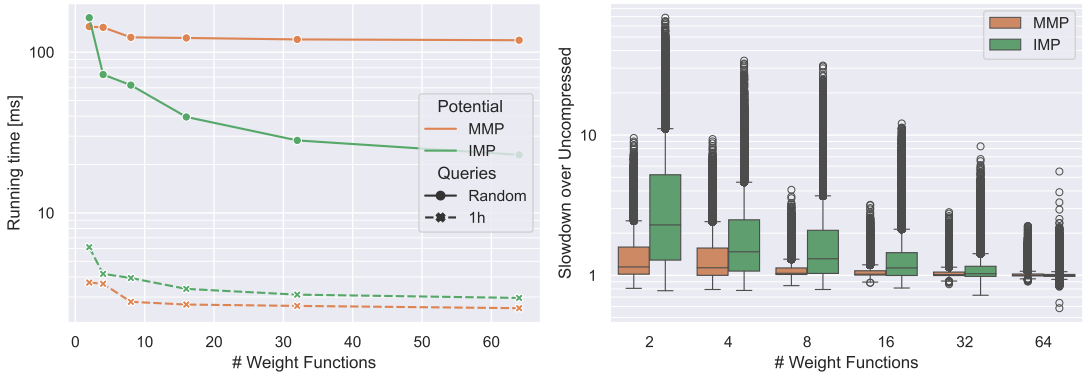


Figure 9.4: Left: Mean running times of 100 k queries on OSM Germany with only predicted traffic by number of remaining weight functions. Right: Boxplot of the per-query relative slowdown over the running time of the respective query with all weight functions.

compared to CCH-Potentials, i.e. rarely more than a factor of three. Therefore, its robustness is not particularly surprising. IMP, which achieves stronger speedups, is less robust against compression. Nevertheless, we can reduce the memory consumption by a factor of about three to 32 functions and still achieve very decent query times. With 32 functions, the absolute memory consumption decreases to less than 20 GB, which is at least manageable. Surprisingly, even with only four weight functions, IMP is still faster than MMP on long-range queries. This clearly shows the superiority of IMP for long-range queries. The compression algorithm itself takes less than a minute, depending on the final number of weight functions. Thus, its running time is dominated by the regular preprocessing. However, this is only the case because of the efficient parallelization.

Figure 9.5 depicts running times and parallel efficiency of the compression algorithm with a varying number of threads on PTV Europe when merging 96 buckets until only 16 buckets remain. Without parallelization, the compression takes 6 to 7 minutes. This is almost as long as the full MMP preprocessing. Luckily, with 16 threads, the running time can be reduced to 50 seconds which is a speedup of about 8. With fewer threads, the efficiency is even higher. Even though the parallelization does not scale perfectly, running times are still reduced significantly. As a result, compression times only make up a small fraction of the total preprocessing times.

Modelling Assumptions. Our approach of using time-dependent A* potentials for combined traffic is based on the assumption that live traffic will always be worse than predicted traffic. We analyze our traffic data to validate this assumption. With the PTV data, 58% of the live traffic speeds are in the range of the predicted speeds, and 36% are strictly worse than the predictions. Only 6% of the reported live traffic is better than the predictions. This appears to confirm our assumption. However, the Mapbox traffic behaves differently. Only 5% of the observed

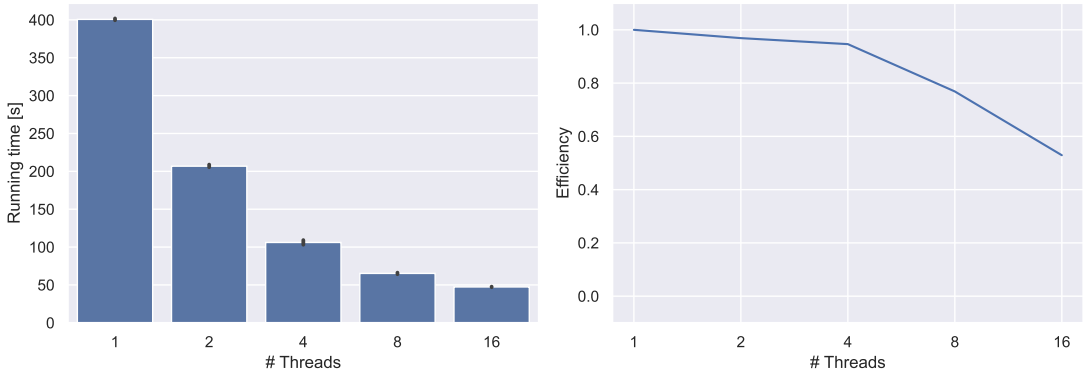


Figure 9.5: Left: Mean running times of compression of IMP buckets for PTV Europe from 96 to 16. The black lines (barely visible) indicates the standard deviation. Right: Parallel efficiency, i.e. speedup over single threaded running time divided by number of threads.

live speeds are strictly worse than the predictions, while up to 23% are strictly better. The remaining observations are within the predicted speed range. This result appears to contradict our assumption. The reason for this substantial difference between the data sets is that the Mapbox live traffic speeds are derived from raw GPS traces, while the PTV data is a traffic incident database. Therefore, whether our model can be applied depends on the type of traffic data one has to work with.

9.5 Conclusion

In this chapter, we proposed time-dependent A* potentials for efficient and exact routing in time-dependent road networks with both predicted and live traffic. We presented two realizations of time-dependent potentials with different trade-offs. Both allow fast live traffic updates within a fraction of a minute. IMP achieves query times two orders of magnitude faster than Dijkstra's algorithm and up to an order of magnitude faster than state-of-the-art time-independent potentials. To the best of our knowledge, this makes our approach the first to achieve interactive query performance while allowing fast updates in this setting. For future work, we would like to apply our time-dependent potentials to other extended scenarios in time-dependent routing. Further, studying ways to relax our modelling assumptions could be fruitful. For example, one could derive time-dependent potentials from a lower bound of the traffic predictions with some additional slack for live traffic faster than the predictions. Also, it would be very interesting to develop other time-dependent potential functions.

Part III

Extended Problem Settings

10 Fast Computation of Shortest Smooth Paths and Uniformly Bounded Stretch with Lazy RPHAST

So far, we have assumed that the data we use for our routing is an accurate representation of the reality and that a shortest path in our graph yields a "good" practical route. Unfortunately, this is not always the case. As traffic feeds are derived from live data, they are inherently noisy and incomplete. Simply exchanging free-flow for traffic-aware travel times and then solving the shortest path problem may, in fact, lead to problematic routes. For example, such routes may include undesired detours such as briefly using a parking area to bypass a jammed highway. In other cases, the shortest route may traverse a residential area while there is an alternative route around the area which is just a few seconds faster. It appears that the shortest path does not always correspond to the practically best route.

Therefore, in this chapter, we study an extended problem model, the *shortest smooth path problem* (SSPP) [DSS18]. To avoid undesired detours, a second weight function is taken into account. The first *volatile* weight function models the current traffic situation. The second *smooth* weight function models the free flow travel times and may include additional penalties, for example to avoid residential areas. The goal is to find the shortest path with respect to the volatile weights without too severe detours with respect to the smooth weights.

Related Work. The SSPP was initially introduced by Delling et al. in [DSS18]. The authors discuss the complexity of the problem and show some relations between SSPP and Knapsack but no definitive conclusions could be drawn in their work. The paper also includes two CRP-based algorithms for the SSPP. *Iterative Path Blocking* (IPB) is presented as an exact algorithm for the SSPP. However, it has two issues: First, it takes several seconds even on short-range queries. This makes it unsuitable for practical applications. Second, as we show in this chapter, it is, in fact, not exact. The authors also present a heuristic algorithm based on the via-node paradigm,

i.e. it finds solutions which are concatenations of two shortest paths. It is much faster but may miss promising paths because only via-paths are considered and the path quality is checked heuristically. We are not aware of any other works studying the SSPP.

In the SSPP, limiting the relative length of detours is formalized with the *uniformly bounded stretch* (UBS). The UBS is a path quality measure and quantifies how much longer detours on a path are than their respective fastest alternative. So far, it has been primarily studied in the context of alternative routes [ADGW13]. While quite useful, it is expensive to compute and requires evaluating all subpaths of a path. The authors of [ADGW13] state that it would be ideal to check the UBS in time proportional to the length of the path and a few shortest path queries, though they are not aware of any way to do that. To the best of our knowledge, this goal has not been achieved to this day.

Attribution. This chapter is based on a paper published at SEA 2022 [Zei22a]. A preliminary evaluation of the algorithms can be found in the Bachelor's thesis of Jakob Bussas [Bus21].

Contribution and Outline. In Section 10.2, we settle the complexity of the SSPP by proving that it is strongly NP-complete. Section 10.3 contains algorithmic results. First, we show that IPB as described in [DSS18] may not find optimal results. Second, we describe necessary adjustments to make it exact. Third, we present an alternative realization based on A* and CCH-Potentials. Fourth, we present an efficient algorithm to compute exact UBS values, typically with only a few shortest path queries and in time proportional to the path length as the authors of [ADGW13] had hoped for. Fifth, we present Iterative Path Fixing, a new SSPP heuristic. All our algorithms utilize Lazy RPHAST as a crucial ingredient to achieve fast running times. Section 10.4 contains a thorough evaluation of our algorithms. It clearly shows the effectiveness of our UBS algorithm and our CH-Potentials-based IPB realization, outperforming the state of the art by up to two orders of magnitude.

10.1 Smooth Paths

The *stretch* of a path is defined as $\text{stretch}_\ell(P) = \frac{\ell(P)}{\text{dist}_\ell(v_1, v_k)}$, i.e. the ratio between the path length and the shortest distance between its endpoints. The *uniformly bounded stretch* $\text{UBS}_\ell(P) = \max_{1 \le i < j \le k} \text{stretch}_\ell(P_{i,j})$ indicates the maximum stretch over all subpaths. Let $\text{opt}(s, t)$ denote some shortest path between s and t. We observe the following useful property of UBS:

Observation 10.1. *The UBS of a path* $P = (v_1, \dots, v_i, \dots, v_j \dots, v_k)$ *where* $P_{1,i} = \text{opt}(v_1, v_i)$ *and* $P_{j,k} = \text{opt}(v_j, v_k)$ *is equal to* $\text{UBS}(P_{i,j})$.

This is because the stretch of any subpath only decreases when appending optimal segments to the beginning or end.

In [DSS18], Delling et al. introduce the *shortest smooth path problem* (SSPP). A path P is ϵ-smooth with respect to a weight function ℓ when $\text{UBS}_\ell(P) < 1 + \epsilon$. Given a graph G, vertices

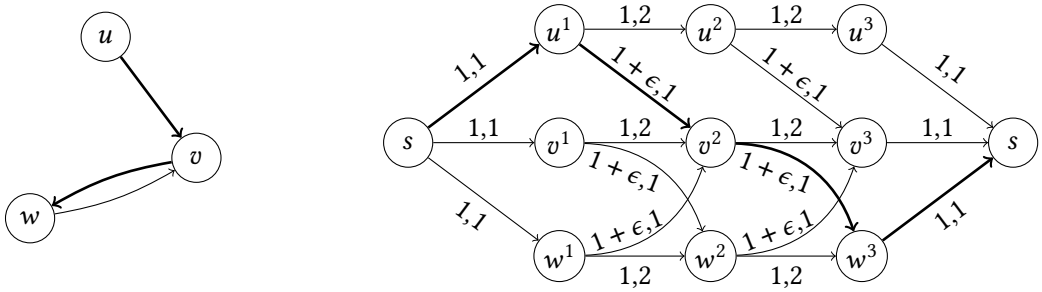


Figure 10.1: Illustration of our transformation from HamiltonPath to ShortestSmoothPath. The first edge weight is the smooth weight, the second the volatile weight. The thick edges indicate a Hamiltonian path and the corresponding shortest ϵ-smooth path.

s and s, a smooth weight function ℓ and a volatile weight function ℓ_v and a parameter $\epsilon > 0$, the shortest smooth path problem asks for the shortest path with respect to ℓ_v that is ϵ-smooth in ℓ.

Note that we consider the problem in a three-phase setup as we also did for Chapters 7 and 9. However, the algorithms discussed in this chapter only touch the query phase.

10.2 Complexity

In this section, we prove that SSPP is strongly NP-complete for any $\epsilon \in \mathbb{Q}^{>0}$. We define the decision variant of the problem as follows: An instance $(G, \ell, \ell_v, s, t, k)$ is a satisfiable instance of ϵ-ShortestSmoothPath-Dec if and only if there exists a path $P = (s, \ldots, t)$ in G with $\ell_v(P) \leq k$ and $\mathrm{UBS}_\ell(P) < 1 + \epsilon$.

Theorem 10.2. *ϵ-ShortestSmoothPath-Dec is strongly NP-complete for any $\epsilon \in \mathbb{Q}^{>0}$.*

Proof. A solution can be verified in polynomial time. Determining the path weight in ℓ_v takes running time linear in $\mathcal{O}(|P|)$. To check the UBS, shortest distances have to be computed for all $\mathcal{O}(|P|^2)$ subpaths. This shows that ShortestSmoothPath-Dec $\in$ NP.

To prove the hardness, we give a reduction from the strongly NP-complete HamiltonPath problem [GJ79]. The goal is to find a *Hamiltonian* path, i.e. a simple path which traverses every vertex exactly once. Let $G = (\mathcal{V}, \mathcal{E})$ be the HamiltonPath instance. To distinguish them from the vertices in the SSPP instance, we will denote the vertices in the HamiltonPath instance as *nodes*. We construct the vertices of our SSPP instance by copying each node n times (forming n layers) and creating two additional vertices s and s. Edges only connect successive layers. There are edges between vertices corresponding to the same node and edges corresponding to edges from the HamiltonPath instance. Any $(s, \ldots, t)$ path has exactly $n + 1$ edges and has to traverse all layers. We will choose the edge weights in such a way that the shortest ϵ-smooth path between s and s has to use a different node in each layer. Paths using the same node in different layers will always be non-ϵ-smooth in ℓ or too long in ℓ_v.

Formally, we construct the graph $G' = (\mathcal{V}', \mathcal{E}')$ for our SSPP instance as follows: We set the vertices $\mathcal{V}' = \{v^i \mid v \in \mathcal{V}, i \in [1, n]\} \cup \{s, t\}$. The edge set $\mathcal{E}'$ is the union of three groups of edges $\mathcal{E}_{\text{orig}}$, $\mathcal{E}_{\text{self}}$ and $\mathcal{E}_{\text{terminal}}$ where $\mathcal{E}_{\text{orig}} = \{(u^i, v^{i+1}) \mid uv \in \mathcal{E}, 1 \leq i < n\}$ are the edges between the layers corresponding to edges in the HamiltonPath instance, $\mathcal{E}_{\text{self}} = \{(v^i, v^{i+1}) \mid v \in \mathcal{V}, 1 \leq i < n\}$ are the additional edges between the same nodes in successive layers and $\mathcal{E}_{\text{terminal}} = \{(s, v^1) \mid v \in \mathcal{V}\} \cup \{(v^n, t) \mid v \in \mathcal{V}\}$ are the edges connecting the terminals with the first and last layer. In both weight functions, all edges $e_{\text{term}} \in \mathcal{E}_{\text{terminal}}$ get the weight $\ell(e_{\text{term}}) = \ell_{\text{v}}(e_{\text{term}}) = 1$. The edges in $e_{\text{self}} \in \mathcal{E}_{\text{self}}$ get a smooth weight $\ell(e_{\text{self}}) = 1$ and a volatile weight $\ell_{\text{v}}(e_{\text{self}}) = 2$. The edges in $e_{\text{orig}} \in \mathcal{E}_{\text{orig}}$ get a smooth weight $\ell(e_{\text{orig}}) = 1 + \epsilon$ and a volatile weight $\ell_{\text{v}}(e_{\text{orig}}) = 1$. Setting $k = n + 1$ completes our ShortestSmoothPath-Dec instance. This transformation has quadratic running time. See Figure 10.1 for an illustrated example of the construction. For the sake of readability, we use non-integer weights of $1 + \epsilon$ in this proof. The weights can be turned into integers by multiplying them with the denominator of ϵ.

Now, assume that the HamiltonPath instance admits a Hamiltonian path $P = (v_1, \ldots, v_n)$. Then, $P' = (s, v_1^1, \ldots, v_n^n, t)$ is a solution to the SSPP problem. The path uses two edges from $\mathcal{E}_{\text{terminal}}$ and $n - 1$ edges from $\mathcal{E}_{\text{orig}}$. Thus, $\ell_{\text{v}}(P') = n + 1 = k$. Also, its $\text{UBS}_\ell(P')$ must be smaller than $1 + \epsilon$. Due to Observation 10.1, it is sufficient to show that the UBS is small enough for $P'' = (v_1^1, \ldots, v_n^n)$. For any subpath $P''_{i,j} = (u^i, \ldots v^j)$ for $i < j$, u must not be equal to v because P is a Hamiltonian path. As all edges are from $\mathcal{E}_{\text{orig}}$, $\ell(P''_{i,j}) = (j - i) \cdot (1 + \epsilon)$. The shortest path (with respect to ℓ) between u^i and v^j has to use at least one $\mathcal{E}_{\text{orig}}$ edge because $u \neq v$. Thus, $\text{dist}_\ell(P''_{i,j}) \geq (j - i - 1) + (1 + \epsilon)$. This yields

$$\text{UBS}_\ell(P''_{i,j}) = \frac{\ell(P''_{i,j})}{\text{dist}_\ell(P''_{i,j})} \leq \frac{(j-i)\cdot(1+\epsilon)}{j-i+\epsilon} < \frac{(j-i)\cdot(1+\epsilon)}{j-i} = 1 + \epsilon$$

which proves that P' is a valid solution for the SSPP instance.

Conversely, suppose that our SSPP instance has an ϵ-smooth path $P' = (s, v_1^1, \ldots, v_n^n, t)$ of weight $\ell_{\text{v}}(P') = n + 1$. Such a path cannot contain edges from $\mathcal{E}_{\text{self}}$ because their volatile weight is 2. We now show that no two vertices in the path can correspond to the same node and thus that $P = (v_1, \ldots, v_n)$ is indeed a Hamiltonian path in G. Suppose for contradiction that $P'_{i,j} = (v^i, \ldots, v^j)$ was a subpath of P'. The length $\ell(P'_{i,j})$ is $(i - j) \cdot (1 + \epsilon)$. Since start and end vertex correspond to the same node, the shortest ℓ path between these vertices is made up of edges from $\mathcal{E}_{\text{self}}$ and has distance $\text{dist}_\ell(v^i, v^j) = i - j$. Thus, $\text{UBS}_\ell(P'_{i,j}) = (1 + \epsilon)$ which means that this subpath must not be part of a solution for the SSPP instance. This is a contradiction. Thus, the SSPP solution induces a valid solution for the HamiltonPath instance. □

10.3 Algorithms

In [DSS18], the *Iterative Path Blocking* (IPB) algorithm is proposed to solve the SSPP optimally. The algorithm repeats two steps until a valid path is found. It maintains a set of forbidden paths $\mathcal{F}$, which is initially empty. In the first step, a shortest path with respect to ℓ_{v} is computed while

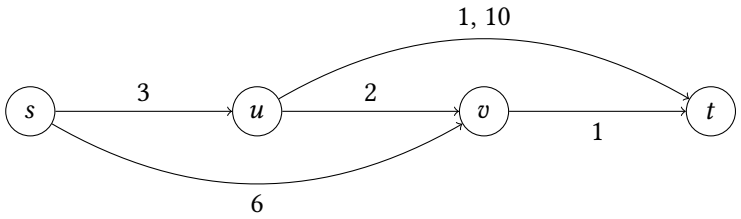


Figure 10.2: Example graph where for $\epsilon = 1$ the shortest ϵ-smooth path (s, v, t) is not prefix-optimal. For all edges except ut, the smooth and the volatile weight function are equal. For ut, the smooth weight is 1 and the volatile weight 10.

avoiding any forbidden paths. In the second step, the obtained path is checked for subpaths violating the UBS constraint. Any violating subpaths are blocked, i.e. added to the list of forbidden paths. Then, the algorithm continues with the next iteration. If no violating subpath is found, the final path is returned.

This framework can be implemented with different concrete algorithms for both steps. The implementation described in [DSS18] is based on CRP [DGPW17]. Here, we propose optimized implementations for both steps based on Lazy RPHAST and CCH-Potentials.

10.3.1 Avoiding Blocked Paths

The authors of [DSS18] describe their approach to the first phase as a variant of Dijkstra's algorithm. When relaxing an edge uv where v is the endpoint of a forbidden path, they backtrack the parent pointers of v, comparing the reconstructed path to the forbidden path. Should the paths match, the search is pruned at v. This algorithm correctly avoids forbidden paths. However, it also avoids some additional paths because Dijkstra's algorithm by construction only finds prefix-optimal paths. But optimal shortest smooth paths may not be prefix-optimal with respect to the volatile weight function ℓ_v. See Figure 10.2 for an example. To the best of our understanding, IPB as described in [DSS18] will not find the shortest smooth path in this example. The algorithm will find the path (s, u, v, t) in the first iteration. This path is not 1-smooth because (u, v, t) has stretch 3 and (u, v, t) will be added to the forbidden path set. With (u, v, t) forbidden, the algorithm will find the path (s, u, t) in the next iteration and return it as the final result. However, the shortest 1-smooth path is (s, v, t). It was missed because the prefix (s, v) is not optimal in ℓ_v and was therefore pruned at v by (s, u, v). We will refer to this variant from now on by *heuristic iterative path blocking* (IPB-H). IPB-H will still find an ϵ-smooth path though it may not necessarily be the shortest. In the following, we describe necessary adjustments to turn IPB into an exact algorithm. We will denote this variant as *exact iterative path blocking* (IPB-E).

To find shortest smooth path with Dijkstra's algorithm, we need to adjust the notion of optimality used to compare labels. It might be necessary to keep a label with suboptimal distance from the start as in the example from Figure 10.2 where the label for (s, v) needs to be

kept at v despite being longer than (s, u, v). This leads to a *label-correcting* variation of Dijkstra's algorithm with possibly multiple labels per vertex. A *label* l at a vertex v consists of a distance $d(l)$ from the source, a set of active forbidden paths $\mathcal{A}(l)$, and a pointer to the parent vertex and label for efficient reconstruction of the labels' path $P(l) = (s, \dots, v)$. The active forbidden path set $\mathcal{A}(l)$ contains all forbidden paths which have a prefix which is a suffix of $P(l)$. A label l can be discarded when v has another label l' with $d(l') \leq d(l)$ and $\mathcal{A}(l') \subseteq \mathcal{A}(l)$.

The search is initialized with a single label at s with distance zero and an empty set of active forbidden paths. When a vertex u with a label l_u is popped from the queue and an edge uv is relaxed, we create a new label l_v as follows: We set the distance $d(l_v) = d(l_u) + \ell_v(uv)$ and the parent label to l_u. We also need to keep track of traversed forbidden paths. If uv is the first edge of a forbidden path $F = (u, v, \dots)$, the path F needs to be added to $\mathcal{A}(l_v)$. For any active forbidden path $F = (\dots, u, w, \dots) \in \mathcal{A}(l_u)$, we need to check if $w = v$, i.e. uv lies on F. If this is the case, F is contained in $\mathcal{A}(l_v)$, or, if uv is the last edge of F, the label l_v must be dropped. If uv is not on F, the forbidden path is not in $\mathcal{A}(l_v)$.

An efficient implementation of this algorithm requires careful engineering. For each edge, we keep track of the forbidden paths it lies on. Labels use a bitset to store the active forbidden paths. This allows for efficient subset checks with bit-wise operations. The bitset size is determined individually for each vertex by the number of blocked paths the respective vertex lies on. In our implementation, we use at least one 128-bit integer which suffices for most queries. Should the number of blocked paths for a vertex exceed 128, we switch to using a dynamically sized array of integers for that vertex. Additionally, each vertex maintains its own queue of labels ordered by distance from s. When the vertex is popped from the queue, it pops the next label from its queue and propagates only this label. If there are any remaining labels in the queue, the vertex is reinserted into the global queue. Finally, we utilize A* with CCH-Potentials on the volatile weight function to guide the search towards the target. As our experiments show, disallowing non-ϵ-smooth paths increases distances only very slightly. Thus, the heuristic is close to perfect and A* very effective for this problem.

Time-Dependent Volatile Weights

So far, we have described our smooth path algorithms for dynamic real-time traffic. However, time-dependent traffic predictions are also a critical aspect of traffic data. The authors of [DSS18] also mention time-dependent smooth paths as possible future work. Fortunately, our A*-based algorithms can be easily adapted to time-dependent and even combined dynamic and time-dependent smooth paths. We utilize IMP from the previous chapter (see Section 9.3.2). There is only one necessary adjustment. Recall that IMP use Lazy RPHAST on $\overline{\ell}^{+}_{\text{comb}}$ (and $\underline{\ell}^{+}_{\text{pred}}$) to obtain an arrival time interval at each vertex for the evaluation of the time-dependent lower bounds. When searching for smooth paths, the distance in $\overline{\ell}_{\text{comb}}$ may not necessarily be a valid upper bound anymore. The shortest path for that length function may not be smooth and thus forbidden. Fortunately, we can obtain a valid upper bound differently. The shortest path for the smooth weight function ℓ will always be sufficiently smooth. Thus, if we take this path

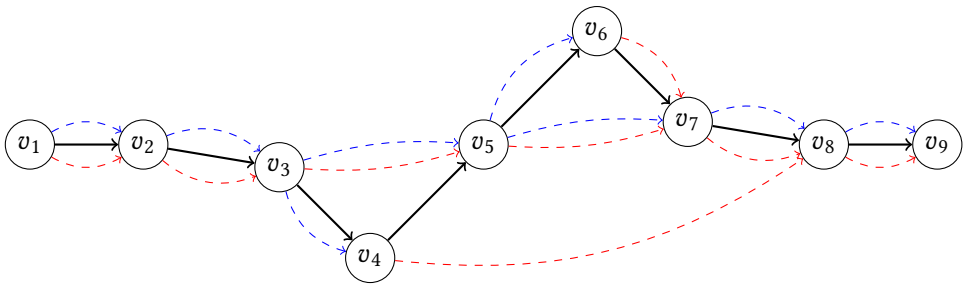


Figure 10.3: Example path (solid, black) with shortest path tree from v_1 to all vertices on the path (dashed, blue) and reverse shortest path tree from all vertices on the path to v_9 (dashed, red).

and evaluate its length in terms of $\overline{\ell}_{\text{comb}}$, we get a valid upper bound on the volatile travel time. Therefore, for the upper bound, we slightly modify CCH and Lazy RPHAST to work on *two* weight functions. The first one, derived from ℓ, is used to decide which path/triangle/edge is the better way, and the second one, derived from $\overline{\ell}_{\text{comb}}$, is used to output the final path length. This is everything necessary to adjust IMP to IPB.

10.3.2 Efficient UBS Computation

According to Delling et al. [DSS18], the UBS computation is one of the bottlenecks of the IPB approach. They employ a many-to-many algorithm. Here, we introduce an algorithm which can compute exact UBS values of typical paths with only a few shortest path queries. We also present a worst-case example where each subpath has a distinct stretch value. This suggests that achieving subquadratic worst-case running time may not be possible.

Consider a path $P = (s = v_1, \ldots, t = v_k)$ as depicted in Figure 10.3. Our algorithm works iteratively. We start with the full path and successively remove prefixes and suffixes until the path is empty or only a shortest path remains. First, we compute shortest distances from s to all vertices on the path. This can be done with a single run of Dijkstra's algorithm which can terminate once all v_i have been settled. Beside the shortest distances, this yields a shortest path tree. We also run Dijkstra's algorithm from s on the reversed graph which yields a backward shortest path tree to s. Now we find the greatest index i such that $P_{1,i}$ is a prefix of all shortest paths $\text{opt}(s, v_l)$ where $i < l \leq k$, i.e. the first branching vertex in the forward shortest path tree. In the worst case this may be s. In the example in Figure 10.3 this is v_3. We analogously obtain the first branching vertex in the reverse shortest path tree to v_k (v_8 in our example). Stated formally, this is the smallest index j such that $P_{j,k}$ is a suffix of all shortest paths $\text{opt}(v_l, t)$ where $1 \leq l \leq j$. By Observation 10.1, subpaths starting from vertices in the segment $P_{1,i-1}$ and subpaths ending at vertices from $P_{j+1,k}$ are not relevant to the UBS computation. We exploit this and only check paths starting from v_i or ending at v_j in the current iteration.

We check the stretch of all subpaths $P_{i,l}$ where $i < l \leq j$ with a linear sweep over the v_l. Since $P_{1,i}$ is a prefix of all shortest paths from s, we can compute the distance $\text{dist}(v_i, v_l)$ as

$\text{dist}(s, v_l) - \text{dist}(s, v_i)$. Thus, each stretch can be checked in constant time with the distances computed by Dijkstra's algorithm. When we are only interested in violating subpaths (rather than computing the exact UBS value of P), the sweep can be stopped after the first (i.e. shortest) violating segment has been found. Forbidding the shortest violating segment starting at v_i is sufficient because it is contained in all longer segments. Checking the stretches of the subpaths $P_{l,j}$ where $i \leq l < j$ works analogously.

Having checked all these stretches, we continue with the next iteration by applying the whole algorithm to the subpath $P_{i+1,j-1}$. We can stop when $i + 1 \geq j - 1$ or when the entire considered path is a shortest path between its endpoints.

This algorithm can be adopted to efficiently compute other path quality measures such as the *local optimality* [ADGW13].

Worst-Case Running Time

This algorithm performs great when long segments are shortest paths, which will often be the case when searching shortest smooth paths. But in the worst case, it still has to check $\Theta(n^2)$ subpath stretches. Consider a complete graph with unit weights and the path $P = (v_1, \ldots, v_n)$. In this graph, the shortest path between any two vertices is always the direct edge and the distance is exactly one. Thus, the shortest path tree from any vertex is a star with the direct edges and our algorithm can only advance by a single vertex in each iteration. This results in a worst case running time of n runs of Dijkstra's algorithm.

We suspect that it is not possible to compute the UBS asymptotically faster. Consider the same graph as before but with weights of unique powers of two for the edges of the path. Now any subpath has a unique length. As all subpaths of three or more vertices still have a shortest distance of one between their endpoints, there are $\Theta(n^2)$ unique stretch values. Thus, computing the UBS of the whole path without checking all $\Theta(n^2)$ stretch values should be difficult if not impossible.

Lazy RPHAST with Path Unpacking

While this algorithm typically needs few stretch checks, running Dijkstra's algorithm a couple of times is still prohibitively slow on large road networks. Luckily, we can speed these computations up drastically by employing Lazy RPHAST, which we already used as an A* heuristic in the shortest path search phase. Recall that Lazy RPHAST allows us to select one target vertex and then to compute shortest distances quickly from many vertices to this target. For the efficient UBS computation, we use two instantiations of this algorithm. In each iteration, we select both endpoints of the considered path and compute distances from and to the endpoints for all vertices on the path. However, we also need the shortest path trees. We therefore extend Lazy RPHAST to also compute shortest path trees.

Dijkstra's algorithm on $G^{\downarrow}$ yields initial parent pointers. We adjust Algorithm 7.3 to continue to maintain these parent pointers during edge relaxation. Thus, after having called

Algorithm 10.1: Path unpacking for Lazy RPHAST.

Data: $\texttt{P}[u]$: parent vertex on the shortest path from u to s, as computed by Dijkstra's algorithm on $G^{\downarrow}$ and an extended Algorithm 7.3.
Data: $\texttt{U}[u]$: whether the path from u to s has been fully unpacked.

```
1  Function Unpack(u):
2      if ¬U[u] ∧ u ≠ t then
3          ComputeAndMemoizeDist(u)
4          Unpack(P[u])
5          if (u, P[u]) is a shortcut for (u, v, P[u]) then
6              P[v] ← P[u]
7              Unpack(v)
8              P[u] ← v
9              Unpack(u)
10         U[u] ← true
```

`ComputeAndMemoizeDist`, we have the shortest path through the CH search space in G^{+}. Algorithm 10.1 depicts the routine to efficiently unpack shortcuts on this path and retrieve shortest path trees in the original graph. We use a bitvector `U` (using a clearlist for fast reinitialization) to mark vertices for which the shortest path has already been fully unpacked which is checked before any actual work is performed. Then, we have to call `ComputeAndMemoizeDist` to ensure that the path through the CH search space has been obtained for u. For vertices encountered through recursive shortcut unpacking this might have not happened before. In the next step, we can now recursively unpack the full path up to the parent $\texttt{P}[u]$ of our current vertex u. Now, all that remains is to unpack the edge $(u, \texttt{P}[u])$ if it is a shortcut. If so, the middle vertex v is set in `P` as the vertex between u and $\texttt{P}[u]$ and `Unpack` is invoked recursively first for v and then again for u to unpack the edges $(v, \texttt{P}[v])$ and (u, v).

10.3.3 Iterative Path Fixing

With an efficient algorithm to find UBS-violating segments we can introduce another natural heuristic to find short smooth paths: Find the shortest path with respect to ℓ_{v} and replace each UBS violating subpath $(v_i, \ldots, v_j)$ with $\text{opt}_{\ell}(v_i, v_j)$. The result may still contain UBS violating subpaths. In this case, we iteratively continue to replace violating segments. When a path contains overlapping violating subpaths, we replace the first, ignore following overlapping subpaths and continue with the next non-overlapping segment. We denote this algorithm as *iterative path fixing* (IPF).

10.4 Evaluation

Environment. Our benchmark machine runs openSUSE Leap 15.3 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.5 GHz and 8 × 64 KiB of L1, 8 × 1 MiB of L2, and 24.75 MiB of shared L3 cache. All running times are sequential. We implement our algorithms in Rust[1] and compile them with `rustc 1.58.0-nightly (b426445c6 2021-11-24)` in the release profile with the `target-cpu=native` option.

Inputs. We use our main benchmark instances for time-independent route planning: DIMACS Europe and OSM Germany. See Section 4.2.1 for a discussion of these networks. For comparison, we also include an OSM Europe graph with 173.8 million vertices and 348 million edges. This graph was used in the evaluation of [DSS18]. For our OSM Germany instance, we use the real-world traffic data from Mapbox discussed in Section 4.2.3. For both Europe instances, we do not have any real world traffic data. Thus, we resort to the approach suggested in [DSS18] and generate synthetic live traffic: For each road where the average speed is greater than 30 kph, we reduce the speed to 5 kph with a probability of 0.5%. To evaluate our algorithms for time-dependent and combined dynamic and time-dependent smooth paths, we also use OSM Germany with the Mapbox predictions and real-time traffic and the Eur20 instance from PTV.

Methodology. We evaluate our algorithms by performing batches of point-to-point shortest ϵ-smooth path queries. As the distance between source and target has a significant influence on the performance, we generate different query batches. For each batch, we pick 1000 source vertices uniformly at random. We then run Dijkstra's algorithm from each source vertex on the graph with the smooth weight function. Following the Dijkstra rank methodology, we store every 2^ith settled vertex [SS05]. This allows evaluating the performance development against varying path lengths. In [DSS18], 1-hour queries were performed. For comparison, we also generate an *1h* batch by picking the first settled vertex with a distance greater than one hour. In addition, we generate a *4h* batch for medium-range queries with the same method and a *random* batch for long range queries where the target is picked uniformly at random.

Preliminary experiments showed that some queries take prohibitively long to answer. Since we are solving an NP-hard problem, this is not very surprising. We abort queries if the algorithm has not found a path with UBS $< (1 + \epsilon)$ after 10 seconds. We report these queries as failed but, nevertheless, do include their running times in our measurements.

Experiments. We start by evaluating different UBS algorithms in isolation. The paths checked by the UBS algorithms are the paths we find while using IPB-H to find shortest smooth paths with $\epsilon = 0.2$ on the rank queries. We limit the time per rank and UBS algorithm to one hour. Thus,

[1] The code for this chapter, all implemented algorithms, scripts to perform experiments and to aggregate the results is available at `https://github.com/kit-algo/traffic_aware`

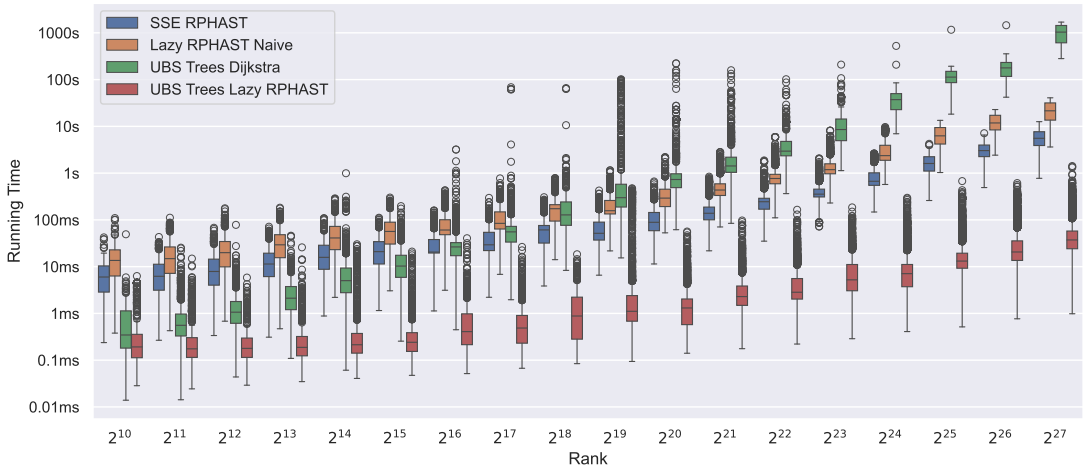


Figure 10.4: Running times of different UBS checking algorithms for paths encountered by IPB-H when answering queries of different ranks with $\epsilon = 0.2$ on OSM Europe. The boxes cover the range between the first and third quartile. The band in the box indicates the median, the whiskers cover 1.5 times the interquartile range. All other running times are indicated as outliers.

slow algorithms may not get to check all paths. Our baseline is computing all distances between pairs of path vertices at once with *SSE RPHAST* [DGW11], which to the best of our knowledge is the fastest known many-to-many algorithm. The second algorithm, denoted as *Lazy RPHAST Naive*, uses Lazy RPHAST to compute distances between all pairs of path vertices. The third one is *UBS Trees Dijkstra* which is the non-accelerated, i.e. Dijkstra-based, implementation of the UBS algorithm introduced in Section 10.3.2. *UBS Trees Lazy RPHAST* denotes the accelerated variant of this algorithm utilizing Lazy RPHAST as described in Section 10.3.2.

Figure 10.4 depicts the results of this experiment. We observe that SSE RPHAST is consistently faster than the naive Lazy RPHAST variant by a roughly constant factor. SSE RPHAST was designed as a many-to-many algorithm and is thus more efficient than naively applying a many-to-one algorithm $|P|$ times. The non-accelerated UBS Trees algorithm is very fast for short paths but quickly becomes prohibitively slow for longer paths. Running Dijkstra's algorithm will traverse a large part of the network if source and target are sufficiently far apart from each other. Doing this multiple times is not feasible. However, the accelerated variant beats SSE RPHAST by about two orders of magnitude across all path lengths.

UBS Trees running times have significantly greater variance than the many-to-many algorithms. This is because the amount of work which UBS Trees can avoid varies strongly between different paths. In contrast, the many-to-many-based algorithms will always check $\Theta(|P|^2)$ subpath distances. Note that the UBS Trees Dijkstra outliers disappear because we limit the time per rank and algorithm. If we checked all paths, the outliers would be present too, but the experiment would take prohibitively long.

Table 10.1: Average performance of our implementations of IPB-E, IPB-H and IPF for different query sets on all time-independent instances with $\epsilon = 0.2$. The Increase column denotes the length increase with respect to ℓ_v of the obtained path over opt_{ℓ_v} and includes only successful queries. We also include the length increase of the shortest path in the smooth weight function opt_ℓ as a trivial upper bound on the length increase. The running time also includes the time of queries aborted after 10 seconds.

		Increase [%]				Running time [ms]			Failed [%]		
	Instance	IPB-E	IPB-H	IPF	opt_ℓ	IPB-E	IPB-H	IPF	IPB-E	IPB-H	IPF
1h	DIMACS Eur Syn	0.8	2.5	4.1	6.3	718.0	168.4	1.5	6.4	1.6	0.0
	OSM Eur Syn	0.2	0.3	0.3	0.8	59.8	22.4	2.7	0.4	0.2	0.0
	OSM Ger 10:21	0.1	0.3	0.4	0.7	261.5	9.2	1.8	2.2	0.0	0.0
	OSM Ger 15:41	0.2	1.5	2.2	3.7	1 373.7	219.1	4.7	12.7	1.2	0.0
4h	DIMACS Eur Syn	0.8	3.4	5.1	6.5	3 513.6	435.4	6.1	33.1	3.1	0.0
	OSM Eur Syn	0.2	0.3	0.3	0.7	331.4	73.2	8.4	2.1	0.6	0.0
	OSM Ger 10:21	0.1	0.4	0.5	1.6	1 449.3	93.1	9.8	13.1	0.0	0.0
	OSM Ger 15:41	0.2	2.1	4.2	10.1	6 597.1	2 568.1	89.2	63.4	15.8	0.0
Random	DIMACS Eur Syn	0.8	2.8	5.3	7.4	6 700.6	2 436.7	30.6	64.2	17.1	0.0
	OSM Eur Syn	0.2	0.4	0.4	0.8	4 758.3	654.2	140.3	38.9	3.1	0.0
	OSM Ger 10:21	0.1	0.4	0.5	1.5	1 366.0	111.6	9.6	12.0	0.0	0.0
	OSM Ger 15:41	0.2	2.1	4.1	8.8	5 771.1	2 419.8	84.5	56.0	16.3	0.0

Next, we evaluate the performance of our query algorithms for the dynamic case on realistic queries and instances. Table 10.1 depicts the results. Both the query set and the instance have a strong influence on the running time. Note that random queries on OSM Germany are on average shorter than four hours which is the reason why the running times on OSM Germany for random queries are faster than for 4h queries. The length increase of the solutions primarily depends on the instance and less on the query set. The synthetic traffic affects DIMACS Europe more strongly than OSM Europe. We suspect that this is because OSM is modeled in much greater detail and contains more shorter edges. In terms of running time, IPB-H is significantly faster than IPB-E and IPF is significantly faster still, which is roughly what we expected. Conversely, the heuristics find somewhat longer paths than the exact IPB-E algorithm and IPF appears to find worse paths than IPB-H. However, one has to be careful interpreting these numbers as a non-negligible amount of queries did not terminate with IPB-E and IPB-H. Because the length increase numbers are averages over different sets, it is not immediately clear if the differences appear because the heuristics find worse paths or because the heuristics find long solutions where the exact algorithm did not finish within 10 seconds. For running times, the averages are also difficult to interpret. They are heavily skewed by outliers and there is no reason to assume a normal distribution. In fact, *median* running times for 1h queries of all algorithms on all instances are all below 2 ms. Clearly, drawing statistically sound conclusions from this

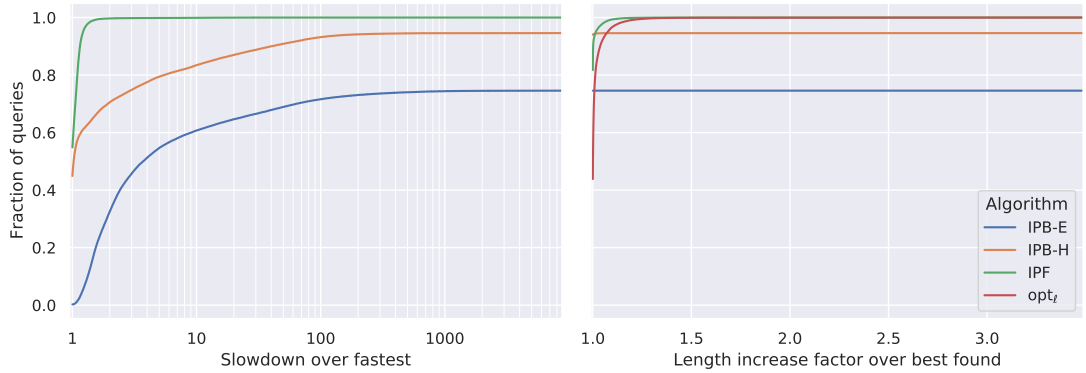


Figure 10.5: Relative performance profiles of our algorithms on all queries from Table 10.1.

experiment requires a closer look.

Figure 10.5 depicts *performance profiles* [DM02] for running times and obtained path lengths on all queries from Table 10.1 combined. Investigating queries across all instances combined is reasonable because we study the relative performance of the different algorithms *per query*. Let $\mathcal{A}$ be the set of algorithms, $\mathcal{Q}$ the set of queries and $\text{obj}(a, q)$ denote the considered measurement from the computation of $a \in \mathcal{A}$ to answer $q \in \mathcal{Q}$. In our case, this is either the running time or the length with respect to ℓ_v of the computed path. The performance ratio $r(a, q) = \frac{\text{obj}(a,q)}{\min\{\text{obj}(a',q) | a' \in \mathcal{A}\}}$ indicates by what factor a deviates from the best solution or the fastest running time for the query q. The performance profile $\rho_a : [1, \infty) \to [0, 1], k \mapsto \frac{|\{q \in \mathcal{Q} | r(a,q) \leq k\}|}{|\mathcal{Q}|}$ of a is the fraction of queries for which a is within a factor of k of the best measurement. For computations that were aborted after 10 seconds, $\text{obj}(a, q) = \infty$. We also include the same performance profiles separated per instance and query set in Appendix E. However, discussing the results in such detail is beyond the scope of this chapter.

The running time performance profile in Figure 10.5 allows for some more nuanced observations: IPF is the fastest algorithm on about 60% of the queries and almost never more than 10 times slower than the fastest one. Surprisingly, IPB-H is also sometimes the fastest to answer a query (in 43% of the queries) but it may also be up to 300 times slower than the fastest algorithm. However, for 83% of all queries it stays within a factor of 10. The exact algorithm is never the fastest but still within a factor of 10 for 61% of the queries. However, it may be several thousand times slower than the fastest algorithm, even with the running time limited to 10 seconds.

The path length performance profile also yields useful insights. Since IPB-E is an exact algorithm, its performance profile contains only a single data point, i.e. for all queries which terminated successfully IPB-E finds the shortest path. The line for IPB-H is almost constant. This means, there are only few queries where it does not find the best solution. Even when it does not find the best solution, it is close to the best one, i.e. the maximum length increase factor over the best solution is 1.36 and all other values are below 1.1. It is quite possible that

Table 10.2: Average performance of our implementations of IPB-E, IPB-H and IPF for different query sets on main time-dependent instances with $\epsilon = 0.2$. The Increase column denotes the length increase with respect to ℓ_v of the obtained path over opt_{ℓ_v} and includes only successful queries. We also include the length increase of the shortest path in the smooth weight function opt_ℓ as a trivial upper bound on the length increase. The running time also includes the time of queries aborted after 10 seconds.

		Live	Increase [%]				Running time [ms]			Failed [%]		
		Traffic	IPB-E	IPB-H	IPF	opt_ℓ	IPB-E	IPB-H	IPF	IPB-E	IPB-H	IPF
1h	Ger	–	0.01	0.13	114.03	233.81	86.4	36.5	5.9	0.00	0.00	0.0
		10:21	0.01	0.18	0.23	1.18	106.1	18.6	8.3	0.00	0.00	0.0
		15:41	0.02	0.78	1.13	2.88	1 032.1	166.1	14.4	0.08	0.01	0.0
	Eur	–	0.00	0.05	0.06	0.64	6.3	2.8	2.9	0.00	0.00	0.0
		07:47	0.03	4.42	26.35	48.88	433.3	248.5	19.6	0.03	0.02	0.0
4h	Ger	–	0.01	0.05	0.06	0.57	480.1	53.7	43.9	0.02	0.00	0.0
		10:21	0.01	0.07	0.08	0.72	594.9	68.5	49.1	0.02	0.00	0.0
		15:41	0.02	0.28	0.34	1.26	2 158.8	297.3	78.7	0.17	0.01	0.0
	Eur	–	0.01	0.03	0.03	0.46	44.6	9.0	8.6	0.00	0.00	0.0
		07:47	0.03	0.75	9.55	16.62	901.8	420.8	47.8	0.08	0.03	0.0
Random	Ger	–	0.01	0.06	0.08	0.59	727.6	105.9	55.5	0.04	0.00	0.0
		10:21	0.01	0.10	0.12	0.90	980.6	132.9	87.7	0.06	0.00	0.0
		15:41	0.02	0.34	0.43	1.53	2 363.5	406.9	105.4	0.17	0.01	0.0
	Eur	–	0.01	0.03	0.03	0.40	191.9	39.6	27.3	0.01	0.00	0.0
		07:47	0.02	0.14	3.82	8.45	1 282.7	480.2	172.3	0.08	0.03	0.0

IPB-H found the optimal solution even for some queries where IPB-E did not terminate. The qualitative performance of IPF varies more strongly. It also finds the best solution on 81% of the queries. More than 99% of the obtained solutions are within a factor of 1.2 to the best found. In the worst case IPF found a path 1.96 times the length of the best one found by another algorithm.

In combination with the averages reported in Table 10.1 we can now draw solid conclusions on the performance of the algorithms. IPF is the algorithm with the most stable running time. Even though it is not always the fastest, it is never much slower than any other algorithm. It is the only algorithm able to answer all queries in less than 10 seconds. In fact, it usually needs only a few milliseconds and only up to several hundreds of milliseconds for extreme cases. It sometimes pays for this with worse solution quality but is still very close to the best found for the vast majority of queries. This makes it an algorithm suitable for practical applications. IPB-H is also a very effective heuristic. It is drastically faster than the exact algorithm and sometimes even faster than IPF. Its performance in terms of quality is much more stable than IPF and often IPB-H will find the best path or something very close to it. The difference in average length increase between IPB-E and IPB-H was not because IPB-H finds much worse

Table 10.3: Average performance of our implementations of IPB-E, IPB-H and IPF for different values of ϵ with 1h queries on OSM Europe with synthetic live traffic. The Increase column denotes the length increase with respect to ℓ_v of the shortest smooth path over the shortest ℓ_v path. It includes only values from successful queries. All other columns indicate average values over all queries, including the ones terminated after 10 seconds.

		Increase	Iterations	Blocked	Running time [ms]			Failed
ϵ		[%]		paths	A*	UBS	Total	[%]
	IPB-E	0.43	137.90	676.2	307.6	22.7	335.9	2.4
0.01	IPB-H	0.56	22.38	24.9	52.8	21.0	74.0	0.6
	IPF	0.61	1.73	–	–	–	2.3	0.0
	IPB-E	0.34	68.10	351.7	132.5	14.8	150.3	0.9
0.05	IPB-H	0.39	32.78	39.8	19.6	38.7	58.6	0.5
	IPF	0.41	1.54	–	–	–	2.3	0.0
	IPB-E	0.27	47.35	256.4	103.3	12.7	118.3	0.8
0.10	IPB-H	0.33	27.10	27.1	3.5	28.9	32.7	0.3
	IPF	0.34	1.45	–	–	–	2.7	0.0
	IPB-E	0.23	24.92	141.7	51.1	7.5	59.7	0.4
0.20	IPB-H	0.26	19.33	19.0	2.6	19.6	22.4	0.2
	IPF	0.28	1.36	–	–	–	2.1	0.0
	IPB-E	0.16	13.64	80.0	41.1	3.8	45.6	0.1
0.50	IPB-H	0.17	19.54	18.9	2.5	19.4	22.1	0.2
	IPF	0.19	1.26	–	–	–	2.0	0.0
	IPB-E	0.11	10.51	55.5	28.1	4.4	33.4	0.2
1.00	IPB-H	0.12	15.13	14.3	2.4	9.6	12.2	0.1
	IPF	0.14	1.19	–	–	–	2.5	0.0

paths but because it is able to answer queries which IPB-E cannot answer. However, it still fails to answer about 5% of all queries in less than 10 seconds. The running time of IPB-E varies even more strongly. On the one hand, many easy queries can be answered in a few milliseconds, but on the other hand, 25% of all queries cannot be answered in less than 10 seconds. The feasibility of solving the problem to exactness with IPB-E strongly depends on the distance of queries and on the smoothness of ℓ_v.

In Table 10.2, we report results for instances with time-dependent and combined dynamic time-dependent volatile weights. The general trends are similar to the ones we observed in Table 10.1 and Figure 10.5. IPB-E finds the shortest paths but is the slowest algorithm. IPF is the fastest algorithm and can answer most queries even in interactive running time. However, there are a few examples where the heuristic makes bad decisions and finds paths which are significantly too long. IPB-H represents good trade-off in between. In contrast to the purely

dynamic case, we observe that length increases generally smaller except for a few outliers with IPF. This coincides with fewer failed queries and sometimes even faster average running times. Apparently, the time-dependent weight functions are smoother which makes the problem easier. We report the performance profiles for this experiment in Appendix E.

For our final experiment, we evaluate the performance of our algorithms with different choices for ϵ with 1000 time-independent queries of 1h range on OSM Europe. Table 10.3 depicts the results. This experiment was also performed in [DSS18] but with only 100 queries. Given the observation from the previous experiment, it should be clear that reported averages allow only for very rough comparisons. However, it is the only data available to compare against related work. Also note that due to the presence of heavy outliers, performing too few queries can distort the numbers drastically. For example, when we ran the same experiment with only 100 queries, the average running times of IPB-H were an order of magnitude faster.

We observe similar trends as the authors of [DSS18]. The smaller the choice of ϵ, the harder the problem becomes. Consequently, the length increase, the number of iterations, the number of blocked paths and the running time increase. However, for our implementation of IPB-H, we measure slightly bigger path increases and slightly more iterations. Our implementation of IPB-H achieves running times two orders of magnitude faster than the CRP-based IPB-H implementation in [DSS18]. One reason for this is our UBS algorithm which only needs a couple of milliseconds for all values of ϵ. In [DSS18], the UBS checking phase takes between 1.3 and 1.9 *seconds*. The CH-Potentials-based shortest path phase is also very efficient across the entire range of ϵ values. Even with many blocked paths, the path lengths increase only little and the CH-Potentials heuristic remains tight and yields good speedups. Our exact IPB-E implementation is still an order of magnitude faster than the IPB-H implementation in [DSS18].

10.5 Conclusion

In this chapter, we studied the shortest smooth path problem and proved its NP-completeness. We introduced a new algorithm for practically efficient UBS computation. This algorithm can compute the exact UBS of typically occurring paths with very few shortest path computations. It outperforms state-of-the-art exact UBS algorithms by around two orders of magnitude and makes computing exact UBS values feasible in practice. Also, it can be used for other path quality measures such as local optimality.

We adapted the existing IPB-H algorithm and realized it with our new UBS algorithm and A* with CH-Potentials. This realization of IPB-H outperforms the original implementation by two orders of magnitude. Also, we present necessary modifications to make the algorithm exact. IPB-E is still about an order of magnitude faster than the CRP-based heuristic implementation. As IPB-H and IPB-E are not always able to find solutions in reasonable time, we introduce another heuristic, IPF. It can consistently find smooth paths even for random queries on massive continental-sized instances in a few tenths of milliseconds. Finally, we demonstrated that with IMP, smooth paths can even be computed efficiently in a time-dependent or combined dynamic time-dependent scenario.

11 Efficient Route Planning with Temporary Driving Bans, Road Closures, and Rated Parking Areas

Many European countries impose temporary driving bans for heavy vehicles. Driving may be restricted during the night, on weekends, and on public holidays. Such bans may apply to the whole road network of a country or parts of it. When routing a heavy vehicle from a source to a destination, it is crucial to take these temporary driving bans into account. But it is not only about heavy vehicles. Temporary closures of bridges, tunnels, border crossings, mountain pass roads, or certain inner-city areas as well as closures due to roadworks may affect all road users alike. In case of road space rationing in cities, the driving restriction may depend on the license plate number. To sum up, temporary driving restrictions exist in different forms, and the closing and re-opening times of a road segment must be considered in the route planning.

As a consequence of temporary driving restrictions, waiting times may be inevitable and even last for hours. During such waiting hours, the vehicle must be parked properly, and thus a suitable parking area has to be found. The driving time of the detour from and to such a parking area should also be incorporated in the route planning. Unfortunately, the underlying shortest (here: quickest) path problem becomes NP-hard if waiting is only allowed at dedicated locations as we showed in Section 3.2.1. This is because in this case, the *FIFO* property is not satisfied, that is, the property that a driver cannot arrive earlier by departing later. Thus, our first research question for this chapter is how we can consider dedicated waiting locations without making the underlying problem NP-hard. It is our aim to obtain a feasible running time even for long-distance routes.

In practice, we often find that small parking areas without any facilities like public toilets or restaurants cause the least detour. So an algorithm that looks for the shortest route, that is, a route with the shortest driving time, would select small parking areas in these cases, provided that waiting is necessary. But the longer the waiting time is, the more vital a secure and pleasant

place for waiting becomes. So it may be important for the driver that nearby facilities of the parking area and their quality are somehow taken into account as well. How to do this is our second research question.

In our setting, a single-criterion objective is not practical. A driver may not always be in favor of the shortest route if that means to spend a very long time waiting and to arrive at the destination considerably later than on the quickest route, that is, a route with the earliest arrival at the destination. Conversely, a driver may not always be interested in a quickest route if that route means to take an unjustified long detour around temporarily closed road segments that could be avoided by waiting in a comfortable place. In other words, an early arrival at the destination (and thus low opportunity costs), little driving time (and thus low fuel costs), and pleasant waiting conditions (and thus high driver satisfaction) are competing criteria. Solutions can differ significantly with regards to these criteria. How to deal with this and find reasonable routes is the third research question.

In this chapter, we answer these questions as follows:

1. We present a model in which waiting is allowed at any vertex and any edge at any time in the road graph but waiting on edges and waiting on those vertices that do not correspond to parking areas is penalized. This is done by assigning a cost to time spent waiting there. Since driving comes at a price, too, we also assign a cost per time unit spent driving. As we will show, we can find a route with least costs in polynomial time if both cost parameters are set to the same value.

2. We assume that the nearby facilities of a parking area and their quality can be expressed by some single rating number. To take account of this, we assign a waiting cost to every corresponding vertex as well. This cost is lower than the cost of waiting anywhere else in the road graph, and it is even lower the higher the rating of the parking area is.

3. We return routes that are Pareto-optimal with regards to arrival time at the destination on the one hand and total costs on the other. Despite the potentially larger output, our algorithm still runs in polynomial time under the same condition as before.

Related Work. Variants of our problem have been studied before. In [DV00], a related problem is discussed where vertices (not edges) have time windows and waiting is associated with a cost. In [PG13], an overview over exact approaches to solving shortest path problems with resource constraints is given. Time windows on vertices are a specific kind of constraint in this framework. More specialized models for truck routing have been proposed. The authors of [TWB18] study the problem of planning a single break, considering driving restrictions and provisions on driver breaks. They aim to find only the route with the earliest arrival.

Attribution. This chapter is based on joint work with Alexander Kleff, Frank Schulz and Jakob Wagenblatt. The results have been published at SEA 2020 [KSWZ20]. A preliminary study on the topic can be found in the Bachelor's thesis of Jakob Wagenblatt [Wag19].

Contribution. We present a novel model that helps answer our three research questions in the context of temporary driving restrictions and dedicated waiting locations. To the best of our knowledge, this is the first unifying approach that gives answers to all three research questions. Our theoretical analysis reveals that our model can be solved to optimality in polynomial time, given certain restrictions on the parameterization. The experimental evaluation of our implementation demonstrates a practical running time. Many queries within continental-sized road networks can be answered within milliseconds. Except some pathological cases, even complex queries with four or more Pareto-optimal solutions are solved in less than a second.

Outline. In Section 11.1, we give a formal definition of the routing problem at hand. In Section 11.2, we present an exact algorithm for this problem. In Section 11.3, we analyze the complexity of the problem and show that our algorithm runs in polynomial time if the costs for driving are the same as for waiting anywhere else than at a dedicated waiting location. In Section 11.4, we describe techniques to speed up the computation. In Section 11.5, we present the main results of our experiments. Finally, we conclude in Section 11.6.

11.1 Problem

A problem instance comprises a *road graph with ban intervals on edges, driving costs and location-dependent waiting costs* (or *road graph with ban intervals and costs* for short) as well as a set of *queries*. We consider this problem in a two-phase setup where the graph is given during preprocessing and the queries are provided online in the query phase. The road graph is characterized by the following attributes:

- A set $\mathcal{V}$ of n vertices and a set $\mathcal{E}$ of m directed edges.
- A mapping $\mathcal{B}$ that maps each edge $e \in \mathcal{E}$ to a set of disjoint time intervals, where the edge is considered to be *closed* during each interval. Precisely, for any *ban interval* $[\tau^{\text{closed}}, \tau^{\text{open}}) \in \mathcal{B}(e)$ of an edge e, τ^{open} denotes the first point in time after τ^{closed} where the edge is open again. Here and in the following, all points in time are integers and the length of an interval is denoted by $|[\tau^{\text{closed}}, \tau^{\text{open}})|$ and equals $\tau^{\text{open}} - \tau^{\text{closed}} > 0$. During such a time span, a vehicle on the corresponding road segment must not move, i.e. it cannot enter the edge or when already on it, it must remain at its current location even if that would be dangerous or illegal. We denote the total number of ban intervals as b.
- An edge length function $\ell : \mathcal{E} \to \mathbb{N}$ that maps each edge $uv \in \mathcal{E}$ to the time $\ell(uv)$ that it takes to drive from u to v, provided the edge is open.
- A mapping p that maps each vertex to a rating in $\{0, 1, \ldots, r\}$ with $r \leq n$. Rating 0 means *unrated*, that is, it is assumed that it is highly difficult, dangerous, and not allowed to park the vehicle there. In contrast to an unrated location, we call a vertex v with $p(v) > 0$ a *parking location*.

- A parameter set of abstract costs, consisting of $\delta \in \mathbb{Q}^{\geq 0}$, the cost per unit of driving time, and $\omega_i \in \mathbb{Q}^{\geq 0}$ for all i from 0 to r, the cost per time unit of waiting on a vertex with rating i. Edges are always unrated so waiting there costs ω_0 per time unit. W.l.o.g. $\omega_i < \omega_{i-1}$ holds for all i between 1 and r, that is, we assume that waiting on vertices with a higher rating costs less than waiting on those with a lower rating.

A *uv-route* is a triple (P, A, D) of three sequences of the same length $k = |P| = |A| = |D|$. Here, P is the sequence of vertices along the route. It describes a (not necessarily simple) *path* in the graph that starts at u and ends in v, that is, $e_i = (P[i], P[i+1]) \in \mathcal{E}$ for all $1 \leq i < k$ and $P[1] = u$ and $P[k] = v$. The other two sequences A and D denote the *arrival times* and the *departure times* from the respective vertices, where $A[i] \leq D[i]$ for all $1 \leq i \leq k$ and $A[i+1] - D[i] \geq \ell(e_i)$ for all $1 \leq i < k$ holds.

A query comprises a *source* $s \in \mathcal{V}$ and a *destination* $t \in \mathcal{V}$ as well as a *planning horizon* H. The latter is defined as the time interval between an *earliest departure time* τ^{dep} from s and a *latest arrival time* $\tau^{\max}$ at t. Waiting costs arise as soon as the planning horizon opens. For a given query, we look for *feasible st*-routes. A route is feasible with respect to the planning horizon if $A[1] = \tau^{\text{dep}}$ and $D[k] \leq \tau^{\max}$. In addition, ban intervals must be taken account of. Let $I_i = [D[i], A[i+1])$ be the time interval in which the edge $e_i = (P[i], P[i+1])$ of the route's path is traversed. A route is feasible with respect to the ban intervals if $\sum_{B \in \mathcal{B}(e_i)} |I_i \cap B| \leq |I_i| - \ell(e_i)$ for all $1 \leq i < k$. Here, $\sum_{B \in \mathcal{B}(e_i)} |I_i \cap B|$ is the time during which the edge between $P[i]$ and $P[i+1]$ is closed while the edge is being traversed. Thus, the time spent on each edge must be greater or equal than the driving time plus the time where moving on the edge was not allowed.

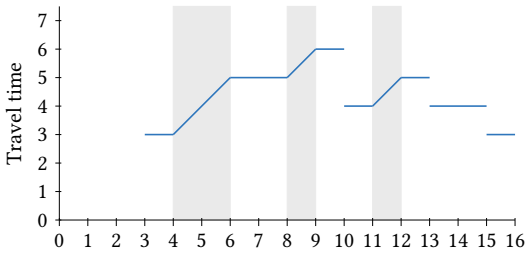
Let *travel time* include driving time and waiting time. The *travel time costs* of a route are the sum of the waiting time costs and the driving time costs. So given a route of length k, the travel time costs are

$$\sum_{i=1}^{k} \omega_{p(P[i])} \cdot (D[i] - A[i]) + \sum_{i=1}^{k-1} (\omega_0 \cdot (A[i+1] - D[i] - \ell(e_i)) + \delta \cdot \ell(e_i)),$$

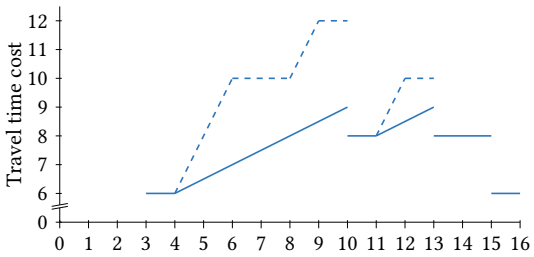
where we use $e_i = (P[i], P[i+1])$. We say an *st*-route is *Pareto-optimal* (or simply *optimal*) if it is feasible and if its travel time costs are less or its arrival time at t is earlier or equality holds in both cases compared to any other feasible *st*-route. For a query, the objective is to find a maximal set of (Pareto-)optimal *st*-routes such that no two routes in the set have both the same arrival time at t and the same travel time costs.

11.2 Algorithm

The algorithm maintains a priority queue. Each entry of the queue consists of a vertex and a point in time within the planning horizon as key. We say a vertex is *visited* at a certain point in time whenever we remove the top entry from the queue, that is, an entry with the earliest time among the entries in the queue. At every vertex $v \in \mathcal{V}$, we store a time-dependent function



(a) Travel time function f_e^{tt} of edge e with ban intervals (grey) and driving time $\ell(e) = 3$. The latest departure to be at v at time τ is $\tau - f_e^{\mathrm{tt}}(\tau)$.



(b) Cost profile of vertex v after linking, that is, after considering travel time (dashed) and waiting time at v (solid).

Figure 11.1: Computing the cost profile of a vertex v. Let v be adjacent to the source s via an edge $e = sv$ with three ban intervals and a driving time $\ell(e)$ of 3. The corresponding travel time function is given in Figure 11.1a. It is infinite between $0 = \tau^{\mathrm{dep}}$ and $3 = \ell(e)$. In Figure 11.1b, we see the cost profile f_v^{c} after considering the travel time along the edge (dashed) and after considering waiting at v (solid). Here, the assumed cost parameters are $\omega_{p(s)} = 0$, $\omega_{p(v)} = 0.5$, and $\delta = \omega_0 = 2$, where $\omega_{p(s)} = 0$ implies that the cost profile f_s^{c} at the source is 0 over the whole planning horizon.

$f_v^{\mathrm{c}} : H \to \mathbb{Q}^{\geq 0} \cup \{\infty\}$. It maps a point in time τ within the planning horizon H to an upper bound on the minimum travel time cost over all sv-routes that end in v at time τ. We call this function *cost profile* of v or, more general, *label* of v. The algorithm works in a *label correcting* manner in the sense that a vertex may be visited multiple times, albeit at different times within the planning horizon.

Before we describe the phases of the algorithm in greater detail, we introduce an auxiliary time-dependent function f_e^{tt} for every edge $e \in \mathcal{E}$. It maps a time τ at the head v of an edge $e = uv$ to the *shortest travel time* that it takes to traverse the edge from u to v completely and be at v at time τ, possibly including waiting time. That is, for a time τ at v, $f_e^{\mathrm{tt}}(\tau)$ is the minimum duration d such that $d - \sum_{B \in \mathcal{B}(e)} |[\tau - d, \tau) \cap B| \geq \ell(e)$ holds if such a d exists, and ∞ otherwise. In other words, $\tau - f_e^{\mathrm{tt}}(\tau)$ is the latest departure time from u in order not to arrive at v later than at time τ. An example is given in Figure 11.1a.

In the initialization phase of the algorithm, we set $f_s^{\mathrm{c}}(\tau) = \omega_{p(s)} \cdot (\tau - \tau^{\mathrm{dep}})$ for all $\tau \in H$. For every other $v \in \mathcal{V} \setminus \{s\}$, we set $f_v^{\mathrm{c}}(\tau) = \infty$ for all $\tau \in H$. Furthermore, we insert the source s with key τ^{dep} into the priority queue.

As long as the queue is not empty, we are in the main loop of the algorithm. In every iteration of the main loop, we remove the top entry from the queue. Let us suppose we visit a vertex u at time $\tau^{\mathrm{visit}} \geq \tau^{\mathrm{dep}}$. Then, we check for every edge $e = uv$ going out of u whether we can improve the cost profile f_v^{c} of v. We do so in three steps. In the first step, we consider the travel time along the edge and set

$$\tilde{f}_v^{\mathrm{c}}(\tau) = f_u^{\mathrm{c}}(\tau - f_{uv}^{\mathrm{tt}}(\tau)) + \delta \cdot \ell(uv) + \omega_0 \cdot (f_{uv}^{\mathrm{tt}}(\tau) - \ell(uv)) \tag{11.1}$$

for all τ with $\tau^{\text{visit}} + f_{uv}^{\text{tt}}(\tau) \leq \tau \leq \tau^{\max}$. For all other $\tau \in H$ we set $\tilde{f}_v^{\text{c}}(\tau) = \infty$. In the second step, we consider waiting at v at cost $\omega_{p(v)}$ per time unit and set

$$\tilde{f}_v^{\text{c}}(\tau) = \min\{\tilde{f}_v^{\text{c}}(\tau') + \omega_{p(v)} \cdot (\tau - \tau') \mid \tau^{\text{dep}} \leq \tau' \leq \tau\} \tag{11.2}$$

for all $\tau \in H$. An example of the first two steps is illustrated in Figure 11.1b. Finally, in the third step, we compare $\tilde{f}_v^{\text{c}}$ and f_v^{c}. Let τ^* be the earliest point in time such that $\tilde{f}_v^{\text{c}}(\tau^*)$ is less than $f_v^{\text{c}}(\tau^*)$ if such a time τ^* exists. Only if it exists, we set $f_v^{\text{c}}(\tau)$ to the minimum of $f_v^{\text{c}}(\tau)$ and $\tilde{f}_v^{\text{c}}(\tau)$ for all $\tau^* \leq \tau \leq \tau^{\max}$. Furthermore, we insert vertex v with key τ^* into the priority queue or decrease the key if v is already contained.

When the priority queue is empty, we enter the finalization phase of the algorithm. We say a time-cost-pair $(\tau, f_t^{\text{c}}(\tau))$ with $\tau \in H$ and $f_t^{\text{c}}(\tau) < \infty$ is Pareto-optimal if there is no time τ' with $\tau^{\text{dep}} \leq \tau' < \tau$ and $f_t^{\text{c}}(\tau') \leq f_t^{\text{c}}(\tau)$. In the finalization phase, we extract an st-route for every Pareto-optimal time-cost-pair. So let such a time-cost-pair $(\tau, f_t^{\text{c}}(\tau))$ be given. In order to find a corresponding route (P, A, D), we initially push t and τ and τ to the front of the (empty) sequences P and A and D, respectively. The following is done iteratively until we reach the source, that is, $P[1] = s$ holds. First, we look for an incoming edge $e = (u, P[1])$ of $P[1]$ and a departure time τ from u with

$$f_u^{\text{c}}(\tau) + \delta \cdot \ell(e) + \omega_0 \cdot (f_e^{\text{tt}}(A[1]) - \ell(e)) = f_{P[1]}^{\text{c}}(A[1])$$

which must exist. We push u and τ to the front of P and D, respectively. Then, we push the earliest time $t \leq D[1]$ such that

$$f_{P[1]}^{\text{c}}(\tau) + \omega_{p(P[1])} \cdot (D[1] - \tau) = f_{P[1]}^{\text{c}}(D[1])$$

holds to the front of the arrival time sequence A, and continue with the next iteration. This concludes the description of the finalization phase and thus the whole algorithm.

For the correctness of the algorithm it is important that the upper bound $f_v^{\text{c}}(\tau)$ on the minimum travel time cost is tight for all $\tau \leq \tau^{\text{visit}}$ and all $v \in \mathcal{V}$ whenever we visit a vertex at time τ^{visit}. After the main loop, it is tight for every $\tau \in H$ and all $v \in \mathcal{V}$, especially for t. This can be proven by induction on the time of visit. The time of visiting a vertex increases monotonically because whenever a vertex is inserted into the queue or its key is decreased, the (new) value of that key can only be later than the current time of visit.

11.3 Analysis

In this section, we first show the intractability of the general problem. Then, we restrict the problem by requiring the driving cost δ to be equal to the unrated waiting cost ω_0, and prove that our algorithm solves the restricted problem in polynomial time.

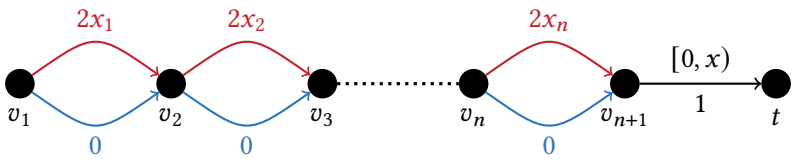


Figure 11.2: Transformation of a PARTITION instance consisting of n numbers x_i into a road graph with temporary driving bans. The last edge is closed before $x = \sum_{i=1}^{n} x_i$. All vertices have rating 0. The graph has parallel edges and edges with driving time 0 for the sake of convenience. This can be avoided by replacing each lower blue edge by two edges with driving time 1, adding 2 to the driving time of each upper red edge, and adding $2n$ to x.

11.3.1 Intractability of the General Problem

The first two theorems show the intractability of the general problem if $\delta \neq \omega_0$. Parking locations are not used in the proofs, so already the simplified problem without parking locations is intractable if $\delta \neq \omega_0$.

Theorem 11.1. *If $\delta < \omega_0$ then it is NP-complete to decide whether there is a feasible route with travel time costs less than or equal to a given threshold k.*

Proof. It can be verified in polynomial time if a route is feasible and has travel time costs less than or equal to k, so the problem is in NP. To show the NP-completeness we reduce from the weakly NP-complete [GJ79] PARTITION problem: Given a set of n numbers x_i, we construct in polynomial time a road graph with time windows and driving times as shown in Figure 11.2. The only time window is on the last edge which is closed up to time $x = \sum_{i=1}^{n} x_i$. Let $\delta < \omega_0$ and set the threshold for the travel time cost to $k = \delta(x + 1)$.

If there is a partition of the x_i into two subsets $\mathcal{S}_1$ and $\mathcal{S}_2$ with the same sum $x/2$ then there is a route with travel time costs k: For $v_i v_{i+1}$ select the upper red edge with driving time $2x_i$ if $x_i \in \mathcal{S}_1$, and the lower blue edge with driving time 0 if $x_i \in \mathcal{S}_2$. Without waiting this route arrives exactly at v_{n+1} at time x and hence arrives at time $x + 1$ at the destination t with travel time cost $\delta(x + 1) = k$.

On the other hand, if there is a route with travel time cost $c \leq k$, there is a valid partition of the x_i: Since the last edge is traversable not earlier than time x and $\delta < \omega_0$, any waiting time on the route implies $c > \delta(x + 1) = k$, so there can be no waiting time included in the route, $c = \delta(x + 1)$, and the driving time of the route is $x + 1$. Set $\mathcal{S}_1$ to the x_i of all upper red edges of the route and $\mathcal{S}_2$ to the remaining x_i. The last edge has driving time 1, so the driving time from v_1 to v_{n+1} equals x and consists solely of the driving times of the upper red edges in the route. We conclude that $\sum_{\mathcal{S}_1} 2x_i = x$, which implies $\sum_{\mathcal{S}_1} x_i = (\sum_{i=1}^{n} x_i)/2 = \sum_{\mathcal{S}_2} x_i$. □

Theorem 11.2. *If $\delta > \omega_0$ then the number of Pareto-optimal routes can be exponential in the number of vertices.*

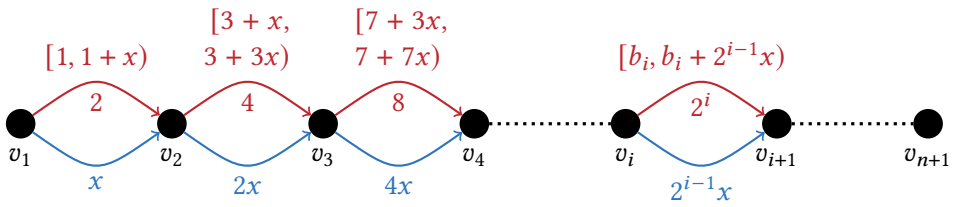


Figure 11.3: A graph with exponentially many Pareto-optimal routes if $\delta > \omega_0$. The ban interval of an upper red edge (v_i, v_{i+1}) begins at $\tau^{\text{closed}}{}_i = (2^i - 1) + (2^{i-1} - 1)x$. There is no parking location in this graph, so all vertices have rating 0. The graph has parallel edges for the sake of convenience, they can be avoided by replacing each lower blue edge by two edges and splitting the driving time.

Proof. Given $\delta > \omega_0$, let $x = \left\lceil \frac{2\delta}{\delta - \omega_0} \right\rceil + 1$ be a possibly large constant. Consider the graph shown in Figure 11.3 with $n + 1$ vertices and $2n$ edges. The graph has no parking locations, so waiting costs are independent of the location. Given a path, a route with that path which arrives as soon as possible has also minimal travel time cost.

In order to calculate the earliest possible arrival time of a route with a given path we assign every v_1v_{k+1}-path a number $y \in \mathbb{Z}^{\geq 0}$ with $0 \leq y < 2^k$: In the binary representation of y beginning with the least significant digit the i-th digit is 1 if and only if in the path the vertices v_i and v_{i+1} are connected by the upper red edge. Let $\overline{y}$ be the ones' complement of y, so $y + \overline{y} = 2^k - 1$. With this representation of y and assuming that for upper red edges always a waiting time for the whole duration of the ban interval is required, the earliest possible arrival time at v_{k+1} is

$$\text{arrival}(y) = \overline{y}x + y(2 + x) = (\overline{y} + y)x + 2y = (2^k - 1)x + 2y \tag{11.3}$$

By induction we show that whenever an upper red edge is used in the path, waiting the whole ban interval is required: For $k = 1$ the path consists of one edge and the departure from v_1 is at time 0. If $y = 0$ the edge is the lower blue edge with driving time x and the arrival at v_2 is at time x. If $y = 1$ the edge is the upper red edge with driving time 2, and the ban interval starts at time 1 during traversal of the edge. A waiting time of x is required, and the arrival at v_2 is at time $2 + x$. Assume now that the proposition holds for v_1v_{k+1}-paths with k edges and consider a v_1v_{k+2}-path with $k + 1$ edges. If the last edge is the lower blue edge there is nothing to show, so further assume the last edge connecting v_{k+1} and v_{k+2} is the upper red edge. According to Equation 11.3 the arrival at v_{k+1} is between $(2^k - 1)x$ and $(2^k - 1)x + 2^{k+1} - 2$. The driving time for (v_{k+1}, v_{k+2}) is 2^{k+1} and the ban interval begins at $\tau^{\text{closed}}{}_{k+1} = (2^{k+1} - 1) + (2^k - 1)x$. Hence, in all cases during the traversal of the edge the ban interval begins.

The travel time cost of the route can be calculated as follows. A lower blue edge v_iv_{i+1} has only driving cost of $2^i\delta x$. An upper red edge v_iv_{i+1} has driving cost $2^{i+1}\delta$ and waiting cost $2^i\omega_0 x$:

$$\begin{aligned}\text{cost}(y) &= \overline{y}\delta x + 2y\delta + y\omega_0 x \\ &= (2^k - 1 - y)\delta x + 2y\delta + y\omega_0 x \\ &= (2^k - 1)\delta x - y\delta x + 2y\delta + y\omega_0 x \\ &= (2^k - 1)\delta x + y(2\delta - x(\delta - \omega_0))\end{aligned} \tag{11.4}$$

Let y_1 and y_2 with $y_1 < y_2$ be the assigned numbers of two arbitrary $v_1 v_{n+1}$-paths and consider corresponding routes with the earliest possible arrival time and travel time costs as calculated above. From Equation 11.3 it follows directly that $\text{arrival}(y_1) < \text{arrival}(y_2)$. By definition of x, we have $2\delta - x(\delta - \omega_0) < 0$ and together with Equation 11.4 it follows that $\text{cost}(y_1) > \text{cost}(y_2)$. This means that each of the 2^n routes is Pareto-optimal because compared to any other route either the arrival time is earlier or the travel time cost is lower. □

11.3.2 Tractable Problem Variant

For the remaining analysis we assume $\delta = \omega_0$. In the setting without parking locations, there is only one optimal solution, since the quickest solution has also the least cost. Hence, this setting is a single-criterion shortest path problem with time-dependent edge weights that fulfill the *FIFO* property and can be solved in polynomial time with a time-dependent variant of Dijkstra's algorithm [Dre69], and also our algorithm reduces to such a time-dependent Dijkstra variant and has polynomial running time. Now we turn to the setting $\delta = \omega_0$ with parking locations and show that it is still tractable.

Cost profiles are piecewise linear functions. An important aspect of our polynomial time proof is to count the non-differentiable points of the profiles. The running time of each profile operation of our algorithm is linear in the number of non-differentiable points of the involved profiles. These points are either *convex*, *concave*, or *discontinuous*, meaning an environment around such a point exists in which the profile is convex or concave or discontinuous, respectively. In a discontinuous point, a profile is always jumping down.

The non-differentiable points in the cost profiles are induced by the travel time functions. In our example of Figure 11.1a, the convex points are $\{4, 8, 11\}$, the concave points are $\{6, 9, 12\}$, and the discontinuous points are $\{10, 13, 15\}$. For a travel time function f_e^{tt} of an edge e, we can assign a convex point τ to the beginning of a ban interval in τ, a concave point τ to the end of a ban interval in τ, and a discontinuous point τ to the end of a ban interval in $\tau - f_e^{\text{tt}}(\tau)$. From this initial assignment, we can derive a ban interval assignment of the convex or discontinuous points of cost profiles. We omit to count the number of concave points of a cost profile because every gradient of a piece must be in $\{\omega_0, \ldots, \omega_r\}$, so the number of consecutive concave points in a cost profile is limited by r.

Initially, a profile f_v^{c} of a vertex v has no convex or discontinuous points. Such points may be introduced in the third step of an iteration of the algorithm when the auxiliary profile $\tilde{f}_v^{\text{c}}$ is merged into f_v^{c}. In the second step of an iteration, no new convex or discontinuous points can

arise in $\tilde{f}_v^c$, so all such points must be created in $\tilde{f}_v^c$ in the first step. Since $\delta = \omega_0$, $\tilde{f}_v^c(\tau)$ is set to $f_u^c(\tau - f_e^{tt}(\tau)) + \delta \cdot f_e^{tt}(\tau)$ (compare Equation 11.1) for some edge $e = uv$ in this step. If τ_v is a convex or discontinuous point of $\tilde{f}_v^c$, then f_e^{tt} must be convex or discontinuous in the same point in time, or f_u^c must be convex or discontinuous in $\tau_u = \tau_v - f_e^{tt}(\tau_v)$. In the former case, τ_v inherits the assignment of the same point in time in f_e^{tt}, whereas in the latter case, τ_v inherits the assignment of τ_u in f_u^c. Since the cost profiles change during the algorithm, we do not only assign a ban interval to every convex or discontinuous point but also an iteration. Again, in the former case, τ_v is assigned the current iteration, whereas in the latter case, τ_v inherits the iteration assignment of τ_u in f_u^c.

Lemma 11.3. *If $\delta = \omega_0$ then a cost profile after iteration i has at most ib convex and at most ib discontinuous points.*

Proof. In the following, we denote the state of the profile f_v^c after iteration i by $f_{i,v}^c$. Let τ_v be a convex or discontinuous point of $f_{i,v}^c$ that is assigned both to an iteration k and to a ban interval of some edge with head x. We can follow the inheritance relation until we finally reach a convex or discontinuous point τ_x in $f_{k,x}^c$. By induction, we have $f_{i,v}^c(\tau_v) = f_{k,x}^c(\tau_x) + \delta \cdot (\tau_v - \tau_x)$. Now suppose there are two convex or two discontinuous points $\tau_v^1 < \tau_v^2$ in the profile $f_{i,v}^c$ that are assigned to the same ban interval and the same iteration k, so they can be traced back to the same point τ_x in $f_{k,x}^c$. Then the previous observation implies that $f_{i,v}^c(\tau_v^2) - f_{i,v}^c(\tau_v^1) = \delta \cdot (\tau_v^2 - \tau_v^1)$ holds, that is, the profile $f_{i,v}^c$ must contain a piece with gradient d that contains both τ_v^1 and τ_v^2. But then τ_v^2 can neither be convex nor discontinuous. Hence, two convex or two discontinuous points must differ in their assigned ban interval or their assigned iteration and there can only be ib discontinuous and convex points, respectively. □

Lemma 11.4. *If $\delta = \omega_0$ then the total number of iterations is at most $2n(b(r+1)+1)$.*

Proof. Similarly to the proof of Lemma 11.3, we show that every vertex is visited at most $\mathcal{O}(rb)$ times. For this, we uniquely assign the elements of the priority queue to the start or end of a ban interval. When a vertex is inserted into the priority queue the key τ^* is the earliest point in time such that $\tilde{f}_v^c(\tau^*) < f_v^c(\tau^*)$. Such a τ^* always coincides with a non-differentiable point of a cost profile, though this point may not necessarily be included in the final cost profile. As the number of concave points between two convex or discontinuous points is at most r, we only track priority queue elements induced by convex or discontinuous points. Additionally, there is one first point of each profile without an assignment to a ban interval.

To assign the points, we distinguish several cases depending on the time $\tau^* - 1$:

- If f_v^c is ∞ at $\tau^* - 1$ then $(\tau^*, \tilde{f}_v^c(\tau^*))$ is the new first point of the resulting profile at v. Due to the correctness of the algorithm, the queue element for the previous first point is still in the queue. It will be updated and its key decreased.
- Otherwise, if $\tilde{f}_v^c(\tau^* - 1) > \tilde{f}_v^c(\tau^*)$ and there is a jump discontinuity in $\tilde{f}_v^c(\tau^*)$ then there is a corresponding ban interval. We assign τ^* to the end of that corresponding ban

interval. This also covers that case that $\tilde{f}_v^c(\tau^* - 1) = \infty$. In this case a corresponding ban interval has to exist, too. For contradiction, assume that no such ban interval exists. Thus, waiting is never beneficial and the entire time was spent driving and $\tilde{f}_v^c(\tau^*) = \delta(\tau^* - \tau^{\text{dep}})$. However, by definition $\tilde{f}_v^c(\tau^*) < f_v^c(\tau^*)$ which is a contradiction since $\delta(\tau^* - \tau^{\text{dep}})$ is the maximum possible cost value at τ^*. Thus, a ban interval has to exist and can be used for the assignment.

- Otherwise, if $\tilde{f}_v^c(\tau^* - 1) > f_v^c(\tau^* - 1)$ then there is a concave point between $\tau^* - 1$ and τ^*.
- Otherwise, if $\tilde{f}_v^c(\tau^* - 1) = f_v^c(\tau^* - 1)$ then $\tau^* - 1$ is either a concave point of $\tilde{f}_v^c$ or a convex point of f_v^c. In the case of a convex point, we assign τ^* to the start of the corresponding ban interval. Note that the first piece of the resulting profile from $\tau^* - 1$ to τ^* has an incline less than d.

We show by contradiction that a vertex at time τ^{visit} which is assigned to a specific start or end of a ban interval is visited only once: Assume a vertex is visited again at τ_2 with the same assignment as a previous visit at τ_1. Due to the correctness of the algorithm all cost profiles up to τ_2 are correct. If the points are assigned to the same start of a ban interval, we have $f_v^c(\tau_2 - 1) = f_v^c(\tau_1 - 1) + d(\tau_2 - \tau_1)$. This is a contradiction to the remark above that the incline at $\tau_1 - 1$ is less than d. If the points are assigned to the same end of a ban interval, we have $C_v(\tau_2) = f_v^c(\tau_1) + d(\tau_2 - \tau_1)$. If the incline at τ_1 was less than d, this leads to the same contradiction as in the previous case. If the incline at τ_1 was d, we distinguish two cases: Either τ_1 was inserted into the queue before τ_2. This is a contradiction because the cost at τ_2 was not improved and τ_2 would not have been inserted. If τ_2 was inserted before τ_1, τ_2 would have been removed from the queue when τ_1 was inserted.

We conclude that for one vertex, due to the assignment there are at most b visits assigned to the start of a ban interval and b visits assigned to the end of a ban interval. Additionally, there is one visit for the first point of the profile and up to $(r + 1)$ visits due to concave points between two consecutive visits assigned to ban intervals. In total, we have at most $2b(r + 1) + 1$ visits of a vertex. □

Theorem 11.5. *If $\delta = \omega_0$ then the running time of the algorithm is polynomial.*

Proof. From Lemma 11.3 with the bound from Lemma 11.4 it follows that the number of pieces of any profile that is constructed during the algorithm is polynomial.

We now estimate the overall running time of our algorithm: Lemma 11.4 states that the total number of iterations is polynomial. In every iteration of the algorithm one vertex is considered and for its outgoing edges the profiles are updated with a running time linear in the number of pieces of the profiles. The adjacent vertices are inserted into the priority queue or their keys are decreased. Since the size of the priority queue is at most the total number of vertices also the running time of the priority queue operations is polynomial. □

11.4 Implementation

This section describes the speedup techniques we employ in our implementation and some implementation details.

We store cost profiles as a sorted list of pieces. Each piece is represented as a triple: a point in time from which this piece is valid, the costs it takes to reach the vertex at the beginning of the piece and the incline of the piece. For each piece we also store a parent vertex. This allows us to efficiently reconstruct routes by traversing the parent pointers.

We employ A* to guide the search toward the destination vertex. The queue is ordered by the original key plus an estimate of the remaining distance (here: driving time) to the destination. The estimate for vertex u is denoted by $\pi_t(u)$. We use the exact shortest driving time to t without driving restrictions as the potential. This is the best possible potential in our case. We efficiently extract these exact distances from a Contraction Hierarchies using Lazy RPHAST as described in Chapter 6. Since our algorithm has to run until the queue is empty, we cannot immediately terminate when we reach the destination. However, we get a tentative cost profile at the destination. This allows for effective pruning. Additionally, we do not need to insert a vertex u into the queue when $\tau^{\text{visit}} + \pi_t(u) > \tau^{\max}$ holds, that is, we cannot reach the destination from u within the planning horizon.

We employ pruning to avoid linking and merging when possible using the following rules:

- Consider a vertex u that is visited at τ^{visit}. Before relaxing any outgoing edges, we first check if u can actually contribute to any optimal route to t. If $f_u^{\text{c}}(\tau) + \pi_t(u) \cdot \delta > f_t^{\text{c}}(\tau + \pi_t(u))$ for all τ with $\tau^{\text{visit}} \leq \tau < \tau^{\max}$, u can not contribute to an optimal route to t and can thus be skipped.

- Let $\mathsf{c}_u = \min\{\tau \mid f_u^{\text{c}}(\tau) < \infty\}$ be the first point in time such that u can be reached with finite costs and ∞ if no such point exists. For each vertex u, we maintain a lower bound $\underline{\mathsf{b}}[u] = \min_\tau\{f_u^{\text{c}}(\tau)\}$ and an upper bound $\overline{\mathsf{b}}[u] = \max_{\tau > \mathsf{c}_u}\{f_u^{\text{c}}(\tau)\}$ or ∞, if there are no finite costs. They can be updated efficiently during the merge operation. An edge uv only needs to be relaxed if $\underline{\mathsf{b}}[u] + \ell(uv) \cdot \delta \leq \overline{\mathsf{b}}[v]$ or $\mathsf{c}_u + \ell(uv) < \mathsf{c}_v$.

- When all of the pieces of the cost profile of a vertex u share the same parent vertex v and $p(u) = 0$, the edge uv back to the parent does not need to be relaxed as loops can never be part of an optimal route unless they include waiting at a parking location.

11.5 Evaluation

Environment. Our algorithm is implemented in C++14 and compiled with Visual C++. For the CH-potentials, we build upon the Contraction Hierarchy implementation of RoutingKit.[1] All experiments were conducted on a Windows 10 Pro machine with an Intel i7-7600 CPU with a base frequency of 3.4 GHz and 32 GB of DDR4 RAM. The implementation is single-threaded.

[1] `https://github.com/RoutingKit/RoutingKit`

Table 11.1: Rating and default waiting cost by capacity of parking locations. The driving cost is the same as the cost for waiting at unrated vertices.

Capacity of parking locations	≥ 80	≥ 40	≥ 15	≥ 5	≥ 1	–
Rating	5	4	3	2	1	0
Default waiting costs	3	4	5	6	7	14
Number of parking locations	448	997	2 664	5 418	5 748	21.9 M

Inputs. Our experimental setup is taken from [Brä18]. We perform experiments on a road network used in production by PTV[2]. The network is adapted from data by TomTom[3]. It covers Austria, France, Germany, Italy, Liechtenstein, Luxembourg, and Switzerland. It has 21.9 million vertices and 47.6 million edges. We use travel times, driving bans, and road closures for a truck with a gross combined weight of 40 tons. Driving bans were derived from the current legislation of the respective countries. This includes Sunday driving bans in all countries, a late Saturday driving ban in Austria and night driving bans in Austria, Liechtenstein and Switzerland. Additionally, there is a Saturday driving ban in Italy during the summer holidays. The data set also includes several local road closures in city centers.

Parking locations were taken from data by Truck Parking Europe[4]. There is a total of 15 317 vertices classified as parking locations in our data set. The data set also contains the capacity of each parking location. We assign each parking location a rating between 1 and 5 depending on its capacity. Table 11.1 shows the number of parking locations for each rating and our default waiting costs. We also evaluate different parameterizations. The waiting costs are calculated such that for an hour of waiting a detour of up to four minutes will be taken to get to a parking location rated better by one. For waiting at the source vertex of a query, we assign zero waiting costs regardless of the rating.

Methodology. We generate two sets of source-destination pairs and combine them with different planning horizons. The first set is used to evaluate the practicality of our model. It is designed to make the algorithm cope with the night driving ban in Austria and Switzerland. We select 100 pairs of vertices. One vertex is randomly selected from the area around southern Germany. The other vertex is selected from the area around northern Italy. See Appendix F for exact coordinates and a visualization. We store each pair in both directions. Hence, we have 200 vertex pairs in this set. The planning horizon starts at Monday 2018/7/2, 18:00 with length one day (query set A1) and two days (A2). Figure 11.4 depicts an example query from A1.

The second set is generated by selecting 100 source vertices uniformly at random. From each source vertex, we run Dijkstra's algorithm without a specific target ignoring any driving

[2] `https://ptvgroup.com`
[3] `https://tomtom.com`
[4] `https://truckparkingeurope.com`

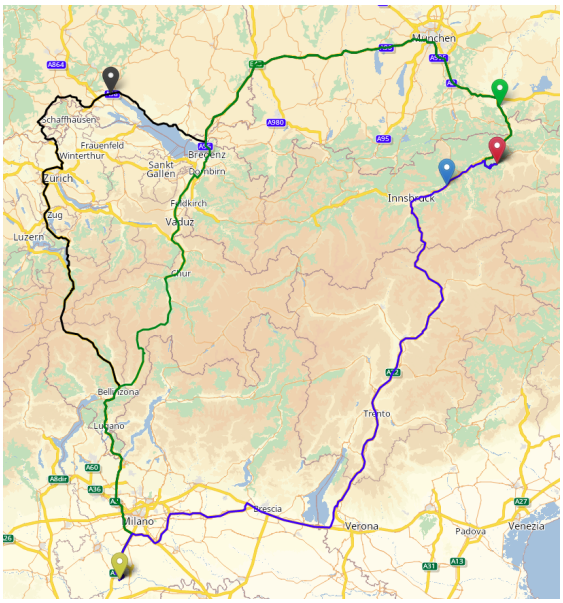


Figure 11.4: Optimal paths of an example query from northwestern Austria to northern Italy, slightly south of Milano. The source is indicated by a red, the destination by a yellow marker. The other markers indicate the parking locations along the respective routes. The blue route in the east has the shortest driving time, around 10.5 hours, but the latest arrival. It schedules a waiting time of seven hours during the night driving ban at a parking location of rating 4 and afterwards takes the fastest route to the destination. The green route in the middle arrives an hour earlier at the destination but the driving time is over two hours longer. This route includes three hours of waiting at a parking location of rating 5. The black route in the west takes 16 hours to drive, includes only a few minutes of waiting and arrives six minutes before the green one.

restrictions. Dijkstra's algorithm explores the graph by traversing vertices in increasing distance of the source vertex. We use the order in which vertices are settled to select destination vertices with different distances from the source. Every 2^ith settled vertex with $i \in [12, 24]$ is stored. We denote i as the *rank* of the query. This results in 1 300 source-destination pairs. We combine these vertex pairs with four planning horizons: starting at Friday 2018/7/6, 06:00 for one day (denoted as query set B1), for two days (B2) and starting later that day at 18:00 for one day (B3) and for two days (B4).

Experiments. We first investigate whether allowing waiting everywhere (albeit penalized) may lead to unwanted results in practice. On the one hand, routes with many stops are impractical. Our experiments indicate that this is not the case: Accross all routes for A1, there is at most one additional stop scheduled (0.2 on average). On the other hand, let us call a route *precarious* if waiting is scheduled at an unrated location (other than the source vertex). For 187 of the 200 queries of A1, there is no precarious route in the Pareto set. For the other 13 queries, the Pareto

Table 11.2: Query statistics for different waiting cost parameters for query set A1. The first six columns show the waiting cost parameters. Waiting costs at the source are always set to zero. The waiting time columns depict the share of the time spent waiting at vertices with the respective rating summed up over all routes. The routes column gives the average number of optimal routes per query. The arrival time deviation column contains the average of the difference between earliest and latest arrival time among all optimal routes for all queries. Running times are also averaged.

						Waiting time by rating [%]							Optimal Routes	Arrival time deviation	Running time
ω_5	ω_4	ω_3	ω_2	ω_1	$\omega_0 = \delta$	s	5	4	3	2	1	0	[#]	[h:mm]	[ms]
1	10	50	100	1000	10000	59.4	2.5	5.8	23.3	3.1	2.8	3.1	3.02	2:21	364.1
1	2	4	8	16	128	62.2	3.5	6.5	19.8	2.1	2.8	3.1	3.02	2:20	412.3
1	2	4	8	16	32	70.8	6.0	5.1	12.1	1.0	1.9	3.1	2.96	2:20	435.4
3	4	5	6	7	14	79.3	6.2	3.1	4.6	1.5	2.0	3.3	2.86	2:17	529.4
16	24	28	30	31	32	85.2	4.6	1.1	3.3	1.1	1.1	3.6	2.71	2:14	742.2

set always contains more than one route, and it is always only the quickest route in the Pareto set that is precarious. So filtering out such routes in a postprocessing step does not make a query infeasible. On average, the second quickest route in the Pareto set arrives 422 s later than the quickest but precarious route (minimum 38 s, maximum 877 s).

We also evaluate the influence of different waiting cost parameterizations on the performance and the results of our algorithm. Table 11.2 depicts the results. We observe that the parametrization has only limited influence on the results of the algorithm. The average number of optimal routes and the arrival time deviation change only very little even between the two most extreme configurations. Since waiting at the source vertex costs nothing, the majority of the waiting in all configurations is scheduled there. When waiting at parking locations is much cheaper than driving, less waiting time will be scheduled at the source and more waiting at parking locations. Also, clear differences between the costs lead to a better running time, because cost profiles become less complex.

We next investigate the algorithm's performance for each of the different query sets. We report the same numbers limited to non-trivial queries. A query is denoted as *trivial* if there is exactly one optimal route which is also optimal when ignoring all driving restrictions. Table 11.3 depicts the results. Clearly, the query set has a strong influence on the running time of the algorithm. Average running times range from ten milliseconds to one second when looking at all queries. However, median query times are significantly smaller. The reason for this is that our algorithm can answer trivial queries in a few milliseconds or less. Due to the perfect potentials, the algorithm only traverses the optimal path. Once the destination is reached, because of the target pruning, all other vertices in the queue are skipped and the algorithm terminates. Excluding trivial queries, we get a clearer picture of the algorithm's performance when solving the harder part of the problem.

Table 11.3: Query statistics for all six query sets. First, for all queries. Second, only for non-trivial queries. A query is denoted as trivial if there is exactly one optimal route which is also optimal when ignoring all driving restrictions. All numbers are averages unless reported otherwise. The arrival time deviation column contains the average of the difference between earliest and latest arrival time among all optimal routes for all queries. The routes column contains the number of optimal routes.

			Query	Optimal	Arrival time	Running time	
	Set	Planning horizon	share [%]	Routes [#]	deviation [h:mm]	Avg. [ms]	Median [ms]
	A1	Mon. 18:00, 1 day	100.0	2.86	2:17	529.4	266.3
	A2	Mon. 18:00, 2 days	100.0	3.54	3:19	648.1	405.6
	B1	Fri. 06:00, 1 day	100.0	1.04	0:10	10.0	0.6
	B2	Fri. 06:00, 2 days	100.0	1.08	0:16	79.5	0.7
	B3	Fri. 18:00, 1 day	100.0	1.13	0:08	205.8	0.6
	B4	Fri. 18:00, 2 days	100.0	1.32	0:20	1 028.1	0.7
Only non-trivial	A1	Mon. 18:00, 1 day	67.5	3.82	3:13	764.1	560.6
Only non-trivial	A2	Mon. 18:00, 2 days	72.0	4.53	4:37	899.2	655.0
Only non-trivial	B1	Fri. 06:00, 1 day	4.1	2.19	4:10	42.5	6.6
Only non-trivial	B2	Fri. 06:00, 2 days	4.8	2.76	5:43	1 105.6	35.8
Only non-trivial	B3	Fri. 18:00, 1 day	9.2	2.73	1:25	1 359.0	475.2
Only non-trivial	B4	Fri. 18:00, 2 days	11.6	3.79	2:51	5 819.4	1 947.2

For the query sets B1 and B2, only 4% to 5% of the queries have to deal with driving restrictions. This is mostly due to closures for individual roads in certain cities and not country-wide driving bans. When the planning horizon begins later at 18:00 (B3 and B4), we get around twice as many non-trivial queries. These are primarily caused by the night driving bans in Austria and Switzerland. Road closures and country-wide driving bans lead to different optimal routes. When there is a road closure on the shortest path ignoring any driving restrictions, we often have two optimal routes. One which takes a (small) detour around the closure, and one waiting at the source until the closed road opens and then taking that slightly shorter path. Thus, we have two routes with very similar driving times but (often vastly) diverging arrival times. When dealing with night driving bans, we get more optimal results with different trade-offs as in the example of Figure 11.4.

Increasing the length of the planning horizon to two days leads to more non-trivial queries, more optimal routes per query, and a greater deviation in arrival time. The reason are routes with a travel time longer than 24 hours which were not valid for the shorter planning horizon.

Even when we restrict ourselves to queries with non-trivial results, running times still vary depending on the query set. Average and median deviate not as strong as when considering all queries, but the distribution of running times is still skewed by a few long running queries,

especially on set B4. The reason for this is that the running time heavily depends on the types and lengths of driving restrictions in the search space. The Saturday driving ban in Italy causes heavy outliers in B4 (but also B2 and B3), when the destination lies in an area blocked for most of the planning horizon. This causes the algorithm to explore large parts of the graph, until the driving ban is over. The worst of these queries took 49 seconds to answer. Nevertheless, when looking at query sets A1 and A2, we clearly see that the algorithm can answer queries affected by country-wide night driving bans in less than a second.

11.6 Conclusion

We have introduced a variant of the shortest path problem where driving on edges may be forbidden at times, both driving and waiting entail costs, and the cost for waiting depends on the rating of the respective location. The objective is to find a Pareto set of both quickest paths and minimum cost paths in a road graph. We have presented an exact algorithm for this problem and shown that it runs in polynomial time if the cost for driving is the same as for waiting in an unrated location. With this algorithm, we can solve routing problems that arise in practice in the context of temporary driving bans for trucks as well as temporary closures of roads or even larger parts of the road network.

Our experiments demonstrate that our implementation can answer queries with realistic driving restrictions in less than a second on average. We exploit CH-Potentials to achieve practical running times. Thus, the algorithm can also be used in a dynamic scenario when combined with CCH-Potentials. There are a few slow outlier queries when the destination vertex lies in a blocked area. Bidirectional variants of our algorithm might help avoiding these outliers. A natural extension of our problem at hand is to consider time-dependent driving times or rules for truck drivers that enforce a break after a certain accumulated driving time.

12 Conclusion

In this thesis, we studied efficient algorithms for dynamic and time-dependent route planning problems. We revisited the A* algorithm and introduced CH-Potentials, a new potential function based on CH which can compute *optimal* distance estimates with respect to lower bound weights derived at preprocessing time. CH-Potentials can be applied to any routing problem where reasonable lower bounds can be obtained during the preprocessing. We proposed several refinements to the CCH framework and showed that CCH is now competitive with CH and CRP both in terms of supported features and running times. Further, we presented CATCHUp, the first space-efficient fast and exact speedup technique for time-dependent routing. We also introduced time-dependent A* potentials. This allowed us to design the first approach for routing with combined live and predicted traffic, which achieves interactive running times for exact queries while allowing live traffic updates in a fraction of a minute. Moreover, we studied extended problem models for routing with imperfect data and routing for truck drivers and also presented efficient algorithms for these variants. Finally, we also presented various complexity results for non-FIFO time-dependent routing and the extended problem models.

We already discussed conclusions and future work opportunities in the respective chapters for each result. However, there is a recurring theme in this work worth highlighting here. One idea proven effective and fruitful throughout this thesis is the CH-Potentials framework. This effectiveness was a surprise because the approach is so simple. The entire algorithmic idea of CH-Potentials can be stated in a single sentence: use DFS on CH as an A* potential. After almost two decades of active research on practical route planning algorithms, one could expect that all the simple ideas already have been tried. The research works published in the past few years even appear to confirm this expectation. After the initial burst of results sparked by the DIMACS implementation challenge [GH05, GW05, SS05, Del+06, SS07, GSSD08, BD09,

Bau+10, ADGW12], only few results [HS19, BFMP22] on speedup techniques for the *classical* shortest path problem have been published in recent years. The results published recently mainly studied extended problems [Bat14, Kob15, DSW16, DGPW17, Bau18, BSW21, BW21].

We believe such a simple approach was not found earlier because most works prioritize fast running times over everything else. Therefore, the spectrum of trade-offs between preprocessing effort and query times has been thoroughly explored [Bas+16]. In contrast, we know few works focused on flexible and extensible algorithms [DGPW17]. As CH-Potentials show, flexibility and extensibility come at the cost of some query performance. However, our work also shows that paying this cost presents many new promising opportunities. While CH-Potentials did not consistently achieve the fastest running times, we were able to derive a multitude of *practical* algorithms. We conclude that it might be beneficial for future work to shift the focus from highly competitive running times toward simpler and more extensible algorithms. We are not the first to make this observation. In their work on CRP [DGPW17], one of the prime examples of a technique heavily engineered for practicality, the authors stress that competitive query running times with the fastest known techniques are *not* their goal. In a practical setting, it is entirely sufficient if the shortest path computations are not the bottleneck of the application. Other factors, such as an extensible algorithmic framework and manageable implementation complexity, are much more critical. Designing extensible and simple algorithms is, of course, a challenge. Scientifically evaluating these attributes is even more challenging as they are difficult to quantify. Often, simple techniques will only be discovered after the complicated ones, as illustrated by the development from Highway Hierarchies [SS05] over Highway Node Routing [SS07] to CH [GSSV12]. However, we argue that the algorithm engineering methodology is uniquely qualified to tackle these issues. Our work underlines that the continuous and incremental refinement of problem models, algorithms and implementations complemented with rigorous theoretical and experimental analysis are crucial elements of practical algorithm design. Algorithm engineering helps us to turn deepened understanding of the problem structure not only into faster but also into simpler algorithms. Therefore, it is a critical ingredient of algorithm design for the emerging mobility applications of the future.

Bibliography

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **Alternative routes in road networks**. In *ACM Journal of Experimental Algorithmics* volume 18, 2013. DOI: 10.1145/2444016.2444019.
Cited on pages 4, 64, 79, 144, 150.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. **Hierarchical Hub Labelings for Shortest Paths**. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*. Ed. by Leah Epstein and Paolo Ferragina. Volume 7501 of Lecture Notes in Computer Science, pages 24–35. Springer, 2012. DOI: 10.1007/978-3-642-33090-2_4.
Cited on pages 5, 178.

[ALPR12] Javed A. Aslam, Sejoon Lim, Xinghao Pan, and Daniela Rus. **City-scale traffic estimation from a roving sensor network**. In *The 10th ACM Conference on Embedded Network Sensor Systems, SenSys '12, Toronto, ON, Canada, November 6-9, 2012*. Ed. by M. Rasit Eskicioglu, Andrew Campbell, and Koen Langendoen, pages 141–154. ACM, 2012. DOI: 10.1145/2426656.2426671.
Cited on page 32.

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. **Alternative Route Graphs in Road Networks**. In *Theory and Practice of Algorithms in (Computer) Systems - First International ICST Conference, TAPAS 2011, Rome, Italy, April 18-20, 2011. Proceedings*. Ed. by Alberto Marchetti-Spaccamela and Michael Segal. Volume 6595 of Lecture Notes in Computer Science, pages 21–32. Springer, 2011. DOI: 10.1007/978-3-642-19754-3_5.
Cited on pages 64, 79.

[BFMP22] Daniel Bahrdt, Stefan Funke, Sokol Makolli, and Claudius Proissl. **Distance Closures: Unifying Search- and Lookup-based Shortest Path Speedup Techniques**. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2022, Alexandria, VA, USA, January 9-10, 2022*. Ed. by Cynthia A. Phillips and Bettina Speckmann, pages 1–12. SIAM, 2022. DOI: 10.1137/1.9781611977042.1.
Cited on page 178.

[Bas+16] 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**. In *Algorithm Engineering - Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Volume 9220. Lecture Notes in Computer Science. 2016, pages 19–80. DOI: 10.1007/978-3-319-49487-6_2.
Cited on pages 3, 15, 29, 45, 56, 83, 97, 178, 242, 243.

[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* volume 18, 2013. DOI: 10.1145/2444016.2444020.
Cited on pages 7, 18, 20, 30, 34, 45, 53, 63, 85, 97, 98, 106, 110, 122, 242.

[Bat14] Gernot Veit Eberhard Batz. **Time-Dependent Route Planning with Contraction Hierarchies**. PhD thesis. Karlsruhe Institute of Technology, 2014. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000047759.
Cited on pages 7, 19, 44, 45, 178.

[BCRW16] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. **Search-space size in contraction hierarchies**. In *Theoretical Computer Science* volume 645, pages 112–127, 2016. DOI: 10.1016/j.tcs.2016.07.003.
Cited on pages 67, 75, 76.

[BD09] Reinhard Bauer and Daniel Delling. **SHARC: Fast and robust unidirectional routing**. In *ACM Journal of Experimental Algorithmics* volume 14, 2009. DOI: 10.1145/1498698.1537599.
Cited on pages 5, 6, 97, 177.

[Bau+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* volume 15, 2010. DOI: 10.1145/1671970.1671976.
Cited on pages 5, 97, 178.

[Bau18] Moritz Baum. **Engineering Route Planning Algorithms for Battery Electric Vehicles**. PhD thesis. Karlsruhe Institute of Technology, 2018. 317 pp. DOI: 10.5445/IR/1000082225.
Cited on pages 8, 45, 178.

[Bau+19a] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution**. In *27th Annual European Symposium on Algorithms, ESA 2019, September 9-11, 2019, Munich/Garching, Germany*. Ed. by Michael A. Bender, Ola Svensson, and Grzegorz Herman. Volume 144 of LIPIcs, pages 14:1–14:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. DOI: 10.4230/LIPIcs.ESA.2019.14.
Cited on page 8.

[Bau+19b] Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *Transportation Science* volume 53:6, pages 1627–1655, 2019. DOI: 10.1287/trsc.2018.0889.
Cited on page 8.

[Bau+20] Moritz Baum, Julian Dibbelt, Thomas Pajor, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **Energy-Optimal Routes for Battery Electric Vehicles**. In *Algorithmica* volume 82:5, pages 1490–1546, 2020. DOI: 10.1007/s00453-019-00655-9.
Cited on page 8.

[BDPW16] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **Dynamic Time-Dependent Route Planning in Road Networks with User Preferences**. In *Experimental Algorithms - 15th International Symposium, SEA 2016, St. Petersburg, Russia, June 5-8, 2016, Proceedings*. Ed. by Andrew V. Goldberg and Alexander S. Kulikov. Volume 9685 of Lecture Notes in Computer Science, pages 33–49. Springer, 2016. DOI: 10.1007/978-3-319-38851-9_3.
Cited on pages 6, 7, 20, 45, 53, 97, 106, 122.

[Bel58] Richard Bellman. **On a Routing Problem**. In *Quarterly of Applied Mathematics* volume 16:1, pages 87–90, 1958. DOI: 10.1090/qam/102435.
Cited on pages 15, 23.

[BGHS19] Massimo Bono, Alfonso Emilio Gerevini, Daniel Damir Harabor, and Peter J. Stuckey. **Path Planning with CPD Heuristics**. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus, pages 1199–1205. ijcai.org, 2019. DOI: 10.24963/ijcai.2019/167.
Cited on page 3.

[Brä18] Christian Bräuer. **Route Planning with Temporary Road Closures**. Master Thesis. Karlsruhe Institute of Technology, 2018. URL: https://i11www.iti.kit.edu/_media/teaching/theses/ma-braeuer-18.pdf.
Cited on pages 171, 239.

[BSW19] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. **Real-time Traffic Assignment Using Engineered Customizable Contraction Hierarchies**. In *ACM Journal of Experimental Algorithmics* volume 24:1, pages 2.4:1–2.4:28, 2019. DOI: 10.1145/3362693.
Cited on pages 6, 63, 66, 71, 74, 75, 76, 83, 85, 86, 88, 105, 108.

[BSW21] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. **Fast, Exact and Scalable Dynamic Ridesharing**. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*. Ed. by Martin Farach-Colton and Sabine Storandt, pages 98–112. SIAM, 2021. DOI: 10.1137/1.9781611976472.8.
Cited on page 178.

[BW21] Valentin Buchhold and Dorothea Wagner. **Nearest-Neighbor Queries in Customizable Contraction Hierarchies and Applications**. In *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. Ed. by David Coudert and Emanuele Natale. Volume 190 of LIPIcs, pages 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPIcs.SEA.2021.18.
Cited on pages 6, 64, 66, 78, 79, 91, 178, 214, 215, 216, 217.

[BWZZ20] Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. **Customizable Contraction Hierarchies with Turn Costs**. In *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2020, September 7-8, 2020, Pisa, Italy (Virtual Conference)*. Ed. by Dennis Huisman and Christos D. Zaroliagis. Volume 85 of OASIcs, pages 9:1–9:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/OASIcs.ATMOS.2020.9.
Cited on pages 45, 66, 81, 84, 94, 217.

[Bus21] Jakob Bussas. **Applying Customizable Contraction Hierarchy Potentials to the Shortest Epsilon-Smooth Path Problem**. Bachelor Thesis. Karlsruhe Institute of Technology, 2021.
Cited on page 144.

[Cal61] Tom Caldwell. **On finding minimum routes in a network with turn penalties**. In *Communications of the ACM* volume 4:2, pages 107–108, 1961. DOI: 10.1145/366105.366184.
Cited on pages 53, 80.

[CZL12] Pablo Samuel Castro, Daqing Zhang, and Shijian Li. **Urban Traffic Modelling and Prediction Using Large Scale Taxi GPS Traces**. In *Pervasive Computing - 10th International Conference, Pervasive 2012, Newcastle, UK, June 18-22, 2012. Proceedings*. Ed. by Judy Kay, Paul Lukowicz, Hideyuki Tokuda, Patrick Olivier, and Antonio Krüger. Volume 7319 of Lecture Notes in Computer Science, pages 57–72. Springer, 2012. DOI: 10.1007/978-3-642-31205-2_4.
Cited on page 32.

[Coh+18] Liron Cohen, Tansel Uras, Shiva Jahangiri, Aliyah Arunasalam, Sven Koenig, and T. K. Satish Kumar. **The FastMap Algorithm for Shortest Path Computations**. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by Jérôme Lang, pages 1427–1433. ijcai.org, 2018. DOI: 10.24963/ijcai.2018/198.
Cited on page 3.

[Jon14] Kyle Jones. **Is the subset product problem NP-complete?** Accessed: 2022-02-24. 2014. URL: https://cs.stackexchange.com/a/27973.
Cited on page 24.

[Del09] Daniel Delling. **Engineering and Augmenting Route Planning Algorithms**. PhD thesis. Karlsruhe Institute of Technology, 2009. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000011046.
Cited on page 45.

[Del11] Daniel Delling. **Time-Dependent SHARC-Routing**. In *Algorithmica* volume 60:1, pages 60–94, 2011. DOI: 10.1007/s00453-009-9341-0.
Cited on pages 6, 34, 122.

[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* volume 73:7, pages 940–952, 2013. DOI: 10.1016/j.jpdc.2012.02.007.
Cited on pages 4, 42.

[DGPW17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Customizable Route Planning in Road Networks**. In *Transportation Science* volume 51:2, pages 566–591, 2017. DOI: 10.1287/trsc.2014.0579.
Cited on pages 1, 2, 4, 5, 6, 45, 53, 64, 65, 81, 83, 84, 86, 88, 92, 94, 95, 97, 147, 178, 241, 242, 243.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato Fonseca F. Werneck. **Customizable Route Planning**. In *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*. Ed. by Panos M. Pardalos and Steffen Rebennack. Volume 6630 of Lecture Notes in Computer Science, pages 376–387. Springer, 2011. DOI: 10.1007/978-3-642-20662-7_32.
Cited on page 5.

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato Fonseca F. Werneck. **Graph Partitioning with Natural Cuts**. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*, pages 1135–1146. IEEE, 2011. DOI: 10.1109/IPDPS.2011.108.
Cited on page 68.

[DGW13] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **Hub Label Compression**. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 18–29. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_4.
Cited on pages 5, 63.

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. **Faster Batched Shortest Paths in Road Networks**. In *ATMOS 2011 - 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Saarbrücken, Germany, September 8, 2011*. Ed. by Alberto Caprara and Spyros C. Kontogiannis. Volume 20 of OASIcs, pages 52–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. DOI: 10.4230/OASIcs.ATMOS.2011.52.
Cited on pages 4, 43, 55, 56, 153, 220.

[Del+06] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. **High-Performance Multi-Level Routing**. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*. Ed. by Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 73–91. DIMACS/AMS, 2006. DOI: 10.1090/dimacs/074/04.
Cited on pages 4, 177.

[DN12] Daniel Delling and Giacomo Nannicini. **Core Routing on Dynamic Time-Dependent Road Networks**. In *INFORMS Journal on Computing* volume 24:2, pages 187–201, 2012. DOI: 10.1287/ijoc.1110.0448.
Cited on pages 5, 6, 7, 53, 122.

[DPW15] Daniel Delling, Thomas Pajor, and Renato F. Werneck. **Round-Based Public Transit Routing**. In *Transportation Science* volume 49:3, pages 591–604, 2015. DOI: 10.1287/trsc.2014.0534.
Cited on page 8.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. **Engineering Route Planning Algorithms**. In *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation [DFG priority program 1126]*. Ed. by Jürgen Lerner, Dorothea Wagner, and Katharina Anna Zweig. Volume 5515 of Lecture Notes in Computer Science, pages 117–139. Springer, 2009. DOI: 10.1007/978-3-642-02094-0_7.
Cited on pages 1, 242.

[DSS18] Daniel Delling, Dennis Schieferdecker, and Christian Sommer. **Traffic-Aware Routing in Road Networks**. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pages 1543–1548. IEEE Computer Society, 2018. DOI: 10.1109/ICDE.2018.00172.
Cited on pages 8, 64, 92, 143, 144, 146, 147, 148, 149, 152, 158.

[DW07] Daniel Delling and Dorothea Wagner. **Landmark-Based Routing in Dynamic Graphs**. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*. Ed. by Camil Demetrescu. Volume 4525 of Lecture Notes in Computer Science, pages 52–65. Springer, 2007. DOI: 10.1007/978-3-540-72845-0_5.
Cited on pages 5, 7, 53.

[DW09] Daniel Delling and Dorothea Wagner. **Time-Dependent Route Planning**. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*. Ed. by Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis. Volume 5868. Lecture Notes in Computer Science. Springer, 2009, pages 207–230. DOI: 10.1007/978-3-642-05465-5_8.
Cited on pages 19, 20.

[DW13] Daniel Delling and Renato F. Werneck. **Faster Customization of Road Networks**. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 30–42. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_5.
Cited on pages 5, 94.

[DW15] Daniel Delling and Renato F. Werneck. **Customizable Point-of-Interest Queries in Road Networks**. In *IEEE Transactions on Knowledge and Data Engineering* volume 27:3, pages 686–698, 2015. DOI: 10.1109/TKDE.2014.2345386.
Cited on pages 6, 65, 215.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (editors). **The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006**. Volume 74. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. DIMACS/AMS, 2009. ISBN: 978-0-8218-4383-3. DOI: 10.1090/dimacs/074.
Cited on page 29.

[DV00] Guy Desaulniers and Daniel Villeneuve. **The Shortest Path Problem with Time Windows and Linear Waiting Costs**. In *Transportation Science* volume 34:3, pages 312–319, 2000. DOI: 10.1287/trsc.34.3.312.12298.
Cited on page 160.

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. **Intriguingly Simple and Fast Transit Routing**. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 43–54. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_6.
Cited on page 8.

[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 SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*. Ed. by Jie Bao, Christian Sengstock, Mohammed Eunus Ali, Yan Huang, Michael Gertz, Matthias Renz, and Jagan Sankaranarayanan, pages 66:1–66:4. ACM, 2015. DOI: 10.1145/2820783.2820856.
Cited on page 48.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. **Customizable Contraction Hierarchies**. In *ACM Journal of Experimental Algorithmics* volume 21:1, pages 1.5:1–1.5:49, 2016. DOI: 10.1145/2886843.
Cited on pages 2, 6, 45, 65, 69, 71, 74, 75, 76, 83, 85, 86, 87, 94, 97, 98, 105, 106, 178, 242.

[Dij59] Edsger W. Dijkstra. **A note on two problems in connexion with graphs**. In *Numerische Mathematik* volume 1, pages 269–271, 1959. DOI: 10.1007/BF01386390.
Cited on pages 2, 15, 39.

[DM02] Elizabeth D. Dolan and Jorge J. Moré. **Benchmarking optimization software with performance profiles**. In *Mathematical Programming* volume 91:2, pages 201–213, 2002. DOI: 10.1007/s101070100263.
Cited on page 155.

[DP73] David H. Douglas and Thomas K. Peucker. **Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature**. In *Cartographica: The International Journal for Geographic Information and Geovisualization* volume 10, pages 112–122, 1973. DOI: 10.3138/FM57-6770-U75U-7727.
Cited on page 106.

[Dre69] Stuart E. Dreyfus. **An Appraisal of Some Shortest-Path Algorithms**. In *Operations Research* volume 17:3, pages 395–412, 1969. DOI: 10.1287/opre.17.3.395.
Cited on pages 19, 39, 167.

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. **Optimal Route Planning for Electric Vehicles in Large Networks**. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*. Ed. by Wolfram Burgard and Dan Roth. AAAI Press, 2011. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3637.
Cited on page 8.

[FHS14] Luca Foschini, John Hershberger, and Subhash Suri. **On the Complexity of Time-Dependent Shortest Paths**. In *Algorithmica* volume 68:4, pages 1075–1097, 2014. DOI: 10.1007/s00453-012-9714-7.
Cited on pages 19, 20.

[FNS16] Stefan Funke, André Nusser, and Sabine Storandt. **On k-Path Covers and their applications**. In *The VLDB Journal* volume 25:1, pages 103–123, 2016. DOI: 10.1007/s00778-015-0392-3.
Cited on page 8.

[GJ79] M. R. Garey and David S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.
Cited on pages 22, 24, 25, 145, 165.

[GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. **Some Simplified NP-Complete Graph Problems**. In *Theoretical Computer Science* volume 1:3, pages 237–267, 1976. DOI: 10.1016/0304-3975(76)90059-1.
Cited on page 67.

[Gei15] Robert Geisberger. **Route planning**. US Patent 9,175,972. 2015.
Cited on page 8.

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. **Route Planning with Flexible Objective Functions**. In *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, January 16, 2010*. Ed. by Guy E. Blelloch and Dan Halperin, pages 124–137. SIAM, 2010. DOI: 10.1137/1.9781611972900.12.
Cited on page 8.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. **Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks**. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*. Ed. by Catherine C. McGeoch. Volume 5038 of Lecture Notes in Computer Science, pages 319–333. Springer, 2008. DOI: 10.1007/978-3-540-68552-4_24.
Cited on pages 2, 177.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. **Exact Routing in Large Road Networks Using Contraction Hierarchies**. In *Transportation Science* volume 46:3, pages 388–404, 2012. DOI: 10.1287/trsc.1110.0401.
Cited on pages 2, 4, 7, 41, 43, 47, 55, 84, 88, 97, 178, 242.

[GV11] Robert Geisberger and Christian Vetter. **Efficient Routing in Road Networks with Turn Costs**. In *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*. Ed. by Panos M. Pardalos and Steffen Rebennack. Volume 6630 of Lecture Notes in Computer Science, pages 100–111. Springer, 2011. DOI: 10.1007/978-3-642-20662-7_9.
Cited on pages 4, 45, 53, 64, 81.

[GGG15] Michel Gendreau, Gianpaolo Ghiani, and Emanuela Guerriero. **Time-dependent routing problems: A review**. In *Computers & Operations Research* volume 64, pages 189–197, 2015. DOI: 10.1016/j.cor.2015.06.001.
Cited on pages 8, 19.

[Geo73] Alan George. **Nested Dissection of a Regular Finite Element Mesh**. In *SIAM Journal on Numerical Analysis* volume 10:2, pages 345–363, SIAM, 1973. DOI: 10.1137/0710032.
Cited on page 67.

[GH05] Andrew V. Goldberg and Chris Harrelson. **Computing the shortest path: *A* search meets graph theory**. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165. SIAM, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070455.
Cited on pages 2, 3, 6, 49, 50, 177.

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato Fonseca F. Werneck. **Better Landmarks Within Reach**. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*. Ed. by Camil Demetrescu. Volume 4525 of Lecture Notes in Computer Science, pages 38–51. Springer, 2007. DOI: 10.1007/978-3-540-72845-0_4.
Cited on page 5.

[GW05] Andrew V. Goldberg and Renato Fonseca F. Werneck. **Computing Point-to-Point Shortest Paths from External Memory**. In *Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO 2005, Vancouver, BC, Canada, 22 January 2005*. Ed. by Camil Demetrescu, Robert Sedgewick, and Roberto Tamassia, pages 26–40. SIAM, 2005. URL: http://www.siam.org/meetings/alenex05/papers/03agoldberg.pdf.
Cited on pages 3, 58, 177.

[GHUW19] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. **Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies**. In *Algorithms* volume 12:9, page 196, 2019. DOI: 10.3390/a12090196.
Cited on pages 66, 68, 81, 84, 85, 86.

[HMPV00] Michel Habib, Ross M. McConnell, Christophe Paul, and Laurent Viennot. **Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing**. In *Theoretical Computer Science* volume 234:1-2, pages 59–84, 2000. DOI: 10.1016/S0304-3975(97)00241-7.
Cited on page 69.

[HS18] Michael Hamann and Ben Strasser. **Graph Bisection with Pareto Optimization**. In *ACM Journal of Experimental Algorithmics* volume 23, 2018. DOI: 10.1145/3173045.
Cited on page 68.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**. In *IEEE Transactions on Systems Science and Cybernetics* volume 4:2, pages 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.
Cited on pages 3, 41.

[Heg06] Pinar Heggernes. **Minimal triangulations of graphs: A survey**. In *Discrete Mathematics* volume 306:3, pages 297–317, 2006. DOI: 10.1016/j.disc.2005.12.003.
Cited on page 67.

[HS19] Demian Hespe and Peter Sanders. **More Hierarchy in Route Planning Using Edge Hierarchies**. In *19th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2019, September 12-13, 2019, Munich, Germany*. Ed. by Valentina Cacchiani and Alberto Marchetti-Spaccamela. Volume 75 of OASIcs, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. DOI: 10.4230/OASIcs.ATMOS.2019.10.
Cited on page 178.

[HB15] Torsten Hoefler and Roberto Belli. **Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results**. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. Ed. by Jackie Kern and Jeffrey S. Vetter, pages 73:1–73:12. ACM, 2015. DOI: 10.1145/2807591.2807644.
Cited on page 35.

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. **Engineering multilevel overlay graphs for shortest-path queries**. In *ACM Journal of Experimental Algorithmics* volume 13, 2008. DOI: 10.1145/1412228.1412239.
Cited on page 4.

[HZVG17] Yixiao Huang, Lei Zhao, Tom Van Woensel, and Jean-Philippe Gross. **Time-dependent vehicle routing problem with path flexibility**. In *Transportation Research Part B: Methodological* volume 95, pages 169–195, Elsevier, 2017.
Cited on page 8.

[II87] H. Imai and Masao Iri. **An optimal algorithm for approximating a piecewise linear function**. In *Journal of Information Processing* volume 9:3, pages 159–162, 1987.
Cited on page 106.

[IOAI91] Kunihiro Ishikawa, Michima Ogawa, Shigetoshi Azuma, and Tooru Ito. **Map navigation software of the electro-multivision of the '91 Toyoto Soarer**. In *Vehicle Navigation and Information Systems Conference, 1991*. Volume 2 of, pages 463–473. IEEE, 1991. DOI: 10.1109/VNIS.1991.205793.
Cited on page 2.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. **An Introduction to Statistical Learning**. Volume 112 of. Springer, 2013. DOI: 10.1007/978-1-4614-7138-7.
Cited on page 32.

[JK13] Erik Jenelius and Haris N Koutsopoulos. **Travel time estimation for urban road networks using low frequency probe vehicle data**. In *Transportation Research Part B: Methodological* volume 53, pages 64–81, Elsevier, 2013. DOI: 10.1016/j.trb.2013.03.008.
Cited on page 32.

[Joh81] David S. Johnson. **The NP-completeness column: an ongoing guide**. In *Journal of Algorithms* volume 2:4, pages 393–405, Elsevier, 1981.
Cited on page 24.

[KSWZ20] Alexander Kleff, Frank Schulz, Jakob Wagenblatt, and Tim Zeitz. **Efficient Route Planning with Temporary Driving Bans, Road Closures, and Rated Parking Areas**. In *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*. Ed. by Simone Faro and Domenico Cantone. Volume 160 of LIPIcs, pages 17:1–17:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/LIPIcs.SEA.2020.17.
Cited on page 160.

[Kno+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 Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007. DOI: 10.1137/1.9781611972870.4.
Cited on pages 43, 90.

[Kob15] Moritz Kobitzsch. **Alternative Route Techniques and their Applications to the Stochastics on-time Arrival Problem**. PhD thesis. Karlsruhe Institute of Technology, 2015. DOI: 10.5445/IR/1000050750.
Cited on pages 6, 8, 64, 79, 92, 178.

[Kob21] Moritz Kobitzsch. Personal communication. July 2021.
Cited on page 79.

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. **Evolution and Evaluation of the Penalty Method for Alternative Graphs**. In *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2013, September 5, 2013, Sophia Antipolis, France*. Ed. by Daniele Frigioni and Sebastian Stiller. Volume 33 of OASIcs, pages 94–107. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.94.
Cited on page 79.

[KMPZ22] Spyros Kontogiannis, Paraskevi-Maria-Malevi Machaira, Andreas Paraskevopoulos, and Christos Zaroliagis. **REX: A Realistic Time-Dependent Model for Multimodal Public Transport**. In *22nd Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2022)*. Ed. by Mattia D'Emidio and Niels Lindner. Volume 106 of Open Access Series in Informatics (OASIcs), pages 12:1–12:16. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. ISBN: 978-3-95977-259-4. DOI: 10.4230/OASIcs.ATMOS.2022.12.
Cited on page 8.

[Kon+16] Spyros C. Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos D. Zaroliagis. **Engineering Oracles for Time-Dependent Road Networks**. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*. Ed. by Michael T. Goodrich and Michael Mitzenmacher, pages 1–14. SIAM, 2016. DOI: 10.1137/1.9781611974317.1.
Cited on page 7.

[Kon+17] Spyros C. Kontogiannis, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos D. Zaroliagis. **Improved Oracles for Time-Dependent Road Networks**. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2017, September 7-8, 2017, Vienna, Austria*. Ed. by Gianlorenzo D'Angelo and Twan Dollevoet. Volume 59 of OASIcs, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/OASIcs.ATMOS.2017.4.
Cited on pages 7, 121, 122.

[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*. Volume 22. IfGI prints, 2004, pages 219–230.
Cited on pages 2, 3.

[Lau06] Ulrich Lauther. **An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Road Networks with Precalculated Edge-Flags**. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*. Ed. by Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–39. DIMACS/AMS, 2006. DOI: 10.1090/dimacs/074/02.
Cited on page 3.

[LGT03] Maxim Likhachev, Geoffrey J. Gordon, and Sebastian Thrun. **ARA*: Anytime A* with Provable Bounds on Sub-Optimality**. In *Advances in Neural Information Processing Systems 16 [Neural Information Processing Systems, NIPS 2003, December 8-13, 2003, Vancouver and Whistler, British Columbia, Canada]*. Ed. by Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, pages 767–774. MIT Press, 2003. URL: https://proceedings.neurips.cc/paper/2003/hash/ee8fe9093fbbb687bef15a38facc44d2-Abstract.html.
Cited on page 64.

[CP12] CP. **Bing Maps New Routing Engine**. Accessed: 2020-01-25. 2012. URL: https://blogs.bing.com/maps/2012/01/05/bing-maps-new-routing-engine/.
Cited on pages 6, 242.

[MS10] Matthias Müller-Hannemann and Stefan Schirra (editors). **Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice [outcome of a Dagstuhl Seminar]**. Volume 5971. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-14865-1. DOI: 10.1007/978-3-642-14866-8.
Cited on pages 2, 241.

[NDSL12] Giacomo Nannicini, Daniel Delling, Dominik Schultes, and Leo Liberti. **Bidirectional A^* search on time-dependent road networks**. In *Networks* volume 59:2, pages 240–251, 2012. DOI: 10.1002/net.20438.
Cited on pages 6, 19, 20, 30, 53, 223.

[OR89] Ariel Orda and Raphael Rom. **Traveling without waiting in time-dependent networks is NP-hard**. In tech. rep., 1989.
Cited on page 19.

[OR90] Ariel Orda and Raphael Rom. **Shortest-Path and Minimum-Delay Algorithms in Networks with Time-Dependent Edge-Length**. In *Journal of the ACM* volume 37:3, pages 607–625, 1990. DOI: 10.1145/79147.214078.
Cited on pages 19, 20.

[OR91] Ariel Orda and Raphael Rom. **Minimum weight paths in time-dependent networks**. In *Networks* volume 21:3, pages 295–319, 1991. DOI: 10.1002/net.3230210304.
Cited on page 19.

[PZWS13] Bei Pan, Yu Zheng, David Wilkie, and Cyrus Shahabi. **Crowd sensing of traffic anomalies based on human mobility and social media**. In *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, November 5-8, 2013*. Ed. by Craig A. Knoblock, Markus Schneider, Peer Kröger, John Krumm, and Peter Widmayer, pages 334–343. ACM, 2013. DOI: 10.1145/2525314.2525343.
Cited on page 32.

[Pan+13] Gang Pan, Guande Qi, Wangsheng Zhang, Shijian Li, Zhaohui Wu, and Laurence Tianruo Yang. **Trace analysis and mining for smart cities: issues, methods, and applications**. In *IEEE Communications Magazine* volume 51:6, 2013. DOI: 10.1109/MCOM.2013.6525604.
Cited on page 32.

[PZ13] Andreas Paraskevopoulos and Christos Zaroliagis. **Improved Alternative Route Planning**. In *13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Ed. by Daniele Frigioni and Sebastian Stiller. Volume 33 of OpenAccess Series in Informatics (OASIcs), pages 108–122. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013. ISBN: 978-3-939897-58-3. DOI: 10.4230/OASIcs.ATMOS.2013.108.
Cited on page 79.

[Par61] Seymour Parter. **The use of linear graphs in Gauss elimination**. In *SIAM review* volume 3:2, pages 119–130, SIAM, 1961. DOI: 10.1137/1003021.
Cited on page 67.

[PYJ20] Simon Aagaard Pedersen, Bin Yang, and Christian S. Jensen. **Fast stochastic routing under time-varying uncertainty**. In *VLDB J.* volume 29:4, pages 819–839, 2020. DOI: 10.1007/s00778-019-00585-6.
Cited on page 8.

[PG13] Luigi Di Puglia Pugliese and Francesca Guerriero. **A survey of resource constrained shortest path problems: Exact solution approaches**. In *Networks* volume 62:3, pages 183–200, 2013. DOI: 10.1002/net.21511.
Cited on page 160.

[PSWZ07] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Efficient models for timetable information in public transportation systems**. In *ACM Journal of Experimental Algorithmics* volume 12, pages 2.4:1–2.4:39, 2007. DOI: 10.1145/1227161.1227166.
Cited on page 8.

[RT78] Donald J Rose and Robert Endre Tarjan. **Algorithmic aspects of vertex elimination on directed graphs**. In *SIAM Journal on Applied Mathematics* volume 34:1, pages 176–197, SIAM, 1978. DOI: 10.1137/0134014.
Cited on page 67.

[San09] Peter Sanders. **Algorithm Engineering - An Attempt at a Definition**. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Ed. by Susanne Albers, Helmut Alt, and Stefan Näher. Volume 5760 of Lecture Notes in Computer Science, pages 321–340. Springer, 2009. DOI: 10.1007/978-3-642-03456-5_22.
Cited on pages 2, 241.

[SS05] Peter Sanders and Dominik Schultes. **Highway Hierarchies Hasten Exact Shortest Path Queries**. In *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Volume 3669 of Lecture Notes in Computer Science, pages 568–579. Springer, 2005. DOI: 10.1007/11561071_51.
Cited on pages 2, 4, 119, 134, 152, 177, 178.

[SS12] Peter Sanders and Christian Schulz. **Distributed Evolutionary Graph Partitioning**. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments, ALENEX 2012, The Westin Miyako, Kyoto, Japan, January 16, 2012*. Ed. by David A. Bader and Petra Mutzel, pages 16–29. SIAM / Omnipress, 2012. DOI: 10.1137/1.9781611972924.2.
Cited on page 68.

[SS13] Peter Sanders and Christian Schulz. **Think Locally, Act Globally: Highly Balanced Graph Partitioning**. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 164–175. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_16.
Cited on page 68.

[SS15] Aaron Schild and Christian Sommer. **On Balanced Separators in Road Networks**. In *Experimental Algorithms - 14th International Symposium, SEA 2015, Paris, France, June 29 - July 1, 2015, Proceedings*. Ed. by Evripidis Bampis. Volume 9125 of Lecture Notes in Computer Science, pages 286–297. Springer, 2015. DOI: 10.1007/978-3-319-20086-6_22.
Cited on pages 67, 68, 81, 84.

[SS07] Dominik Schultes and Peter Sanders. **Dynamic Highway-Node Routing**. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Rome, Italy, June 6-8, 2007, Proceedings*. Ed. by Camil Demetrescu. Volume 4525 of Lecture Notes in Computer Science, pages 66–79. Springer, 2007. DOI: 10.1007/978-3-540-72845-0_6.
Cited on pages 4, 5, 177, 178.

[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* volume 5, page 12, 2000. DOI: 10.1145/351827.384254.
Cited on pages 2, 4.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Using Multi-level Graphs for Timetable Information in Railway Systems**. In *Algorithm Engineering and Experiments, 4th International Workshop, ALENEX 2002, San Francisco, CA, USA, January 4-5, 2002, Revised Papers*. Ed. by David M. Mount and Clifford Stein. Volume 2409 of Lecture Notes in Computer Science, pages 43–59. Springer, 2002. DOI: 10.1007/3-540-45643-0_4.
Cited on page 4.

[GL78] Alan George and Joseph W. Liu. **A Quotient Graph Model for Symmetric Factorization**. In *Sparse Matrix Proceedings*, pages 154–175. SIAM, 1978.
Cited on page 69.

[Str17] Ben Strasser. **Dynamic Time-Dependent Routing in Road Networks Through Sampling**. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems, ATMOS 2017, September 7-8, 2017, Vienna, Austria*. Ed. by Gianlorenzo D'Angelo and Twan Dollevoet. Volume 59 of OASIcs, pages 3:1–3:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/OASIcs.ATMOS.2017.3.
Cited on page 7.

[SHB14] Ben Strasser, Daniel Harabor, and Adi Botea. **Fast First-Move Queries through Run-Length Encoding**. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. Ed. by Stefan Edelkamp and Roman Barták. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS14/paper/view/8906.
Cited on page 3.

[SWZ20] Ben Strasser, Dorothea Wagner, and Tim Zeitz. **Space-Efficient, Fast and Exact Routing in Time-Dependent Road Networks**. In *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, 2020, Pisa, Italy (Virtual Conference)*. Ed. by Fabrizio Grandoni, Grzegorz Herman, and Peter Sanders. Volume 173 of LIPIcs, pages 81:1–81:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/LIPIcs.ESA.2020.81.
Cited on page 98.

[SWZ21] Ben Strasser, Dorothea Wagner, and Tim Zeitz. **Space-Efficient, Fast and Exact Routing in Time-Dependent Road Networks**. In *Algorithms* volume 14:3, page 90, 2021. DOI: 10.3390/a14030090.
Cited on page 98.

[SZ21] Ben Strasser and Tim Zeitz. **A Fast and Tight Heuristic for A* in Road Networks**. In *19th International Symposium on Experimental Algorithms, SEA 2021, June 7-9, 2021, Nice, France*. Ed. by David Coudert and Emanuele Natale. Volume 190 of LIPIcs, pages 6:1–6:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPIcs.SEA.2021.6.
Cited on page 45.

[SZ22] Ben Strasser and Tim Zeitz. **Using Incremental Many-to-One Queries to Build a Fast and Tight Heuristic for A* in Road Networks**. In *ACM Journal of Experimental Algorithmics*, 2022. ISSN: 1084-6654. DOI: 10.1145/3571282.
Cited on page 45.

[Stu+15] Nathan R. Sturtevant, Jason M. Traish, James R. Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. **The Grid-Based Path Planning Competition: 2014 Entries and Results**. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel*. Ed. by Levi Lelis and Roni Stern, page 241. AAAI Press, 2015. URL: http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11290.
Cited on page 64.

[Tar72] Robert Endre Tarjan. **Depth-First Search and Linear Graph Algorithms**. In *SIAM Journal on Computing* volume 1:2, pages 146–160, 1972. DOI: 10.1137/0201010.
Cited on page 49.

[Tho04] Mikkel Thorup. **Integer priority queues with decrease key in constant time and the single source shortest paths problem**. In *Journal of Computer and System Sciences* volume 69:3, pages 330–353, 2004. DOI: 10.1016/j.jcss.2004.04.003.
Cited on page 3.

[TWB18] Marieke S. van der Tuin, Mathijs de Weerdt, and G. Veit Batz. **Route Planning with Breaks and Truck Driving Bans Using Time-Dependent Contraction Hierarchies**. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*. Ed. by Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs T. J. Spaan, pages 356–365. AAAI Press, 2018. URL: https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17745.
Cited on pages 8, 160.

[UK14] Tansel Uras and Sven Koenig. **Identifying Hierarchies for Fast Optimal Search**. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*. Ed. by Carla E. Brodley and Peter Stone, pages 878–884. AAAI Press, 2014. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8497.
Cited on page 64.

[Vet09] Christian Vetter. **Parallel time-dependent contraction hierarchies**. Student Research Project. Karlsruhe Institute of Technology, 2009. URL: https://http://algo2.iti.kit.edu/download/vetter_sa.pdf.
Cited on page 84.

[Wag19] Jakob Wagenblatt. **Route Planning with Temporary Road Closures**. Bachelor Thesis. Karlsruhe Institute of Technology, 2019. URL: https://i11www.iti.kit.edu/_media/teaching/theses/ba-wagenblatt-19.pdf.
Cited on page 160.

[Wei97] Karsten Weihe. **Reuse of Algorithms: Still a Challenge to Object-Oriented Programming**. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1997, Atlanta, Georgia, October 5-9, 1997*. Ed. by Mary E. S. Loomis, Toby Bloom, and A. Michael Berman, pages 34–48. ACM, 1997. DOI: 10.1145/263698.263704.
Cited on page 44.

[WZ22] Nils Werner and Tim Zeitz. **Combining Predicted and Live Traffic with Time-Dependent A* Potentials**. In *30th Annual European Symposium on Algorithms, ESA 2022*. Ed. by Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman. Volume 244 of LIPIcs, pages 89:1–89:15. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. ISBN: 978-3-95977-247-1. DOI: 10.4230/LIPIcs.ESA.2022.89.
Cited on page 125.

[Win02] Stephan Winter. **Modeling Costs of Turns in Route Planning**. In *GeoInformatica* volume 6:4, pages 363–380, 2002. DOI: 10.1023/A:1020853410145.
Cited on pages 53, 80.

[Wit15] Sascha Witt. **Trip-Based Public Transit Routing**. In *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*. Ed. by Nikhil Bansal and Irene Finocchi. Volume 9294 of Lecture Notes in Computer Science, pages 1025–1036. Springer, 2015. DOI: 10.1007/978-3-662-48350-3_85.
Cited on page 8.

[Woj18] Dominik Wojtczak. **On Strong NP-Completeness of Rational Problems**. In *Computer Science - Theory and Applications - 13th International Computer Science Symposium in Russia, CSR 2018, Moscow, Russia, June 6-10, 2018, Proceedings*. Ed. by Fedor V. Fomin and Vladimir V. Podolskii. Volume 10846 of Lecture Notes in Computer Science, pages 308–320. Springer, 2018. DOI: 10.1007/978-3-319-90530-3_26.
Cited on pages 23, 25.

[Zei13] Tim Zeitz. **Weak Contraction Hierarchies Work!** Bachelor Thesis. Karlsruhe Institute of Technology, 2013. URL: https://i11www.iti.kit.edu/_media/teaching/theses/weak_ch_work-1.pdf.
Cited on page 67.

[Zei22a] Tim Zeitz. **Fast Computation of Shortest Smooth Paths and Uniformly Bounded Stretch with Lazy RPHAST**. In *20th International Symposium on Experimental Algorithms, SEA 2022, July 25-27, 2022, Heidelberg, Germany*. Ed. by Christian Schulz and Bora Uçar. Volume 233 of LIPIcs, pages 3:1–3:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/LIPIcs.SEA.2022.3.
Cited on page 144.

[Zei22b] Tim Zeitz. **NP-Hardness of Shortest Path Problems in Networks with Non-FIFO Time-Dependent Travel Times** . In *Information Processing Letters*, May 2022. DOI: 10.1016/j.ipl.2022.106287.
Cited on pages 15, 22.

[ZH02] Rong Zhou and Eric A. Hansen. **Multiple Sequence Alignment Using Anytime A*** . In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada*. Ed. by Rina Dechter, Michael J. Kearns, and Richard S. Sutton, pages 975–977. AAAI Press / The MIT Press, 2002. URL: http://www.aaai.org/Library/AAAI/2002/aaai02-155.php.
Cited on page 64.

[Zün19] Michael Zündorf. **Customizable Contraction Hierarchies with Turn Costs**. Bachelor Thesis. Karlsruhe Institute of Technology, 2019. URL: https://i11www.iti.kit.edu/_media/teaching/theses/ba-zuendorf-19.pdf.
Cited on pages 66, 69.

[Zün22] Tobias Zündorf. **Multimodal Journey Planning and Assignment in Public Transportation Networks**. PhD thesis. Karlsruhe Institute of Technology, 2022. 204 pp. DOI: 10.5445/IR/1000145076.
Cited on pages 2, 8.

List of Acronyms

ALT	A*, Landmarks, and Triangle Inequality Used on pages 3, 5–7, 53, 58–61
ATCH	Approximated Time-Dependent Contraction Hierarchies Used on pages 7, 121, 123, 124
BCCH	Bucket CCH Used on pages 90, 91
BFS	Breadth-First Search Used on page 68
CALT	Core ALT Used on pages 6, 121
CATCHUp	Customizable Approximated Time-Dependent Contraction Hierarchies through Unpacking Used on pages 9, 98–101, 104, 105, 121–125, 131, 132, 135, 136, 177
CCH	Customizable Contraction Hierarchies Used on pages iii, 6, 9, 63, 65–67, 69–71, 74–81, 86, 88–95, 99, 102–104, 106, 107, 125, 130–132, 134–138, 144, 147–149, 175, 177, 213, 214, 217, 219, 227, 229, 242
CH	Contraction Hierarchies Used on pages iii, 4–9, 41–44, 46, 47, 51–55, 57–67, 69, 74–76, 79, 81, 83–85, 88–90, 94, 95, 105, 107, 108, 121, 123–126, 129, 132–134, 170, 175, 177, 178, 213, 214, 219, 227, 228, 242
CRP	Customizable Route Planning Used on pages 5–7, 9, 65, 66, 83, 86, 88, 92, 94, 95, 121, 122, 124, 143, 147, 158, 177, 178, 215

DAG	Directed Acyclic Graph Used on pages 47, 48
DFS	Depth-First Search Used on pages 48, 68, 76, 85, 89, 90, 131–133, 177, 213
FIFO	First-in, first-out Used on pages iii, 19, 20, 23, 26, 98, 126, 128, 129, 131–133, 177, 242
HL	Hub Labeling Used on page 5
IFC	InertialFlowCutter Used on pages 68, 84–86
IMP	Interval-Minimum Potentials Used on pages 131–139, 148, 149, 158, 227, 231
IPB	Iterative Path Blocking Used on pages 143, 144, 146, 147, 149
IPB-E	Exact Iterative Path Blocking Used on pages 147, 154–158
IPB-H	Heuristic Iterative Path Blocking Used on pages 147, 152–158
IPF	Iterative Path Fixing Used on pages 151, 154–158
MLD	Multilevel Dijkstra Used on pages 4–6
MMP	Multi-Metric Potentials Used on pages 130–138, 227, 230
OSM	Open Street Map Used on pages 31, 33, 35, 36, 54, 56, 58–62, 84, 87, 114, 134, 136, 138, 152–154, 157, 158
POI	Point of Interest Used on pages 77, 90, 91, 215
PPLF	Periodic Piecewise Linear Function Used on pages 12, 17, 18
SPP	Shortest Path Problem Used on pages 15, 16
SSPP	Shortest Smooth Path Problem Used on pages 143–146

TCH Time-Dependent Contraction Hierarchies
Used on pages 7, 8, 63, 64, 98, 110, 121, 123, 124

TD-S Time-Dependent Sampling
Used on pages 7, 8, 121, 122, 124

TD-SPP Time-Dependent Shortest Path Problem
Used on pages 17–19, 22, 39, 98

UBS Uniformly Bounded Stretch
Used on pages 144–147, 149–153, 158

List of Symbols

Fields

$\mathbb{N}$ The set of all natural numbers.
Used on page 161

$\mathbb{Z}$ The set of all integer numbers.
Used on pages 15, 18, 22–24, 26, 53, 166

$\mathbb{Q}$ The set of all rational numbers.
Used on pages 18, 24, 26, 145, 162, 163

$\mathbb{R}$ The set of all real numbers.
Used on pages 12, 18, 98

Basic Variables

b Total number of driving ban intervals in a graph.
Used on pages 161, 168, 169

c Number of cores of the benchmark machine.
Used on pages 74, 75

e An edge in a graph ($e \in \mathcal{E}$).
Used on pages 12, 17, 21, 23, 26, 27, 34, 35, 41, 51, 53, 80, 81, 99, 126, 127, 130, 146, 161–164, 167, 168

ε An infinitesimal small positive number.
Used on pages 11, 19, 127, 128

m Number of edges in a graph.
Used on pages 11, 23, 39, 75, 161

μ Tentative distance found by a bidirectional search.
Used on pages 40, 42, 50, 51

n Number of vertices in a graph.
Used on pages 11, 23, 39, 41, 44, 69, 72, 74, 79, 145, 146, 150, 161, 165–168

q Distance of the closest remaining element in a queue `Q`.
Used on pages 39, 40, 50

r Number of different parking area ratings.
Used on pages 161, 162, 167–169

s The source vertex.
Used on pages 12, 15–17, 20–24, 26, 27, 39–43, 46, 47, 49–52, 76–80, 106–112, 127–131, 144–151, 162–164, 173

t The target vertex.
Used on pages 12, 15–17, 20–24, 26, 27, 39–43, 46–52, 55, 76–80, 106–112, 127–130, 132, 144–147, 149, 151, 162, 164, 165, 170

u A vertex ($u \in \mathcal{V}$).
Used on pages 11, 12, 18, 23, 24, 39–41, 43, 47–50, 52, 53, 66, 69–74, 77, 78, 80, 99–105, 107–110, 112, 113, 127–129, 131–133, 145–148, 151, 161–164, 168, 170

v A vertex ($v \in \mathcal{V}$).
Used on pages 11, 12, 18, 20, 22–24, 26, 39–41, 43, 47–53, 58, 66, 67, 69–74, 76–78, 80, 81, 99–105, 107–110, 112, 113, 127–133, 144–151, 161–170

w A vertex ($w \in \mathcal{V}$).
Used on pages 12, 18, 41, 48, 49, 51, 53, 70–74, 77, 81, 99–104, 109, 110, 112, 113, 129, 145, 148

x A CATCHUp shortcut expansion.
Used on pages 99–102, 113

Times

τ An instant in time.
Used on pages 12, 18–24, 26, 27, 40, 44, 53, 99–102, 104, 105, 109, 113, 126–133, 163, 164, 167–170

τ^{dep} Earliest time of departure at the source vertex in a time-dependent shortest path query.
Used on pages 12, 17, 20–24, 26, 27, 39, 40, 107, 109, 110, 127, 129–131, 133, 134, 162–164, 169

τ^{max} Latest arrival at t such that an st-path in the time-dependent shortest path problem is considered feasible.
Used on pages 22–24, 26, 27, 130, 133, 162, 164, 170

τ^{now} Time of the current live traffic snapshot.
Used on pages 21, 126, 130, 134

τ^{end} Instant when the real-time traffic travel time of an edge is considered outdated and the predicted travel times can be considered again as more accurate
Used on pages 21, 126, 127

τ^{closed} Beginning of a temporary driving ban interval.
Used on pages 161, 166

τ^{open} End of a temporary driving ban interval.
Used on page 161

τ^{visit} Time when a vertex is visited.
Used on pages 163, 164, 169, 170

Length Functions

ℓ A general constant or time-dependent edge length function.
Used on pages 12, 15–18, 20–24, 26, 27, 34, 35, 40, 41, 50, 53, 70, 77, 78, 80, 81, 100–102, 104, 105, 109, 110, 113, 120, 127–132, 144–146, 148, 149, 151, 154, 156, 161–164, 170, 226

$\overleftarrow{\ell}$ Length function for reversed edges when ℓ has only constant lengths.
Used on page 12

ℓ^{+} Length function for a CH augmented graph.
Used on pages 41, 43, 47, 70, 71, 77, 99, 100, 102, 104–106, 131

ℓ^{+}_{lb} Approximated time-dependent lower bound for shortcuts during CATCHUp customization.
Used on pages 106, 131

ℓ^{+}_{ub} Approximated time-dependent upper bound for shortcuts during CATCHUp customization.
Used on page 106

ℓ^{+}_{pclb} Piecewise constant lower bounds for the augmented graph for Interval-Minimum Potentials.
Used on pages 131, 132

ℓ_{free} Scalar free-flow travel times.
Used on pages 17, 54, 77

ℓ_{pred} Time-dependent travel time functions for predicted traffic.
Used on pages 12, 21, 54, 126, 127, 130, 131

ℓ_{live} Scalar travel times considering the current traffic situation.
Used on pages 12, 21, 126, 127

ℓ_{comb} Time-dependent travel time functions for combined predicted traffic and real-time traffic information.
Used on pages 21, 54, 126, 127, 130

ℓ_{pre} A preprocessing-time edge length function.
Used on pages 51–54, 57–63, 219–221

ℓ_q A query-time edge length function.
Used on pages 17, 51–54, 57–63, 219–221

ℓ_{pen} A length function with penalties for alternative routes.
Used on page 80

ℓ_t A weight function for turn costs.
Used on pages 53, 81

ℓ_e A weight function for the turn-expanded graph.
Used on page 53

ℓ_v The volatile length function in the shortest smooth path problem.
Used on pages 145–148, 151, 154–157

Time-Dependent Functions

f A time-dependent travel time function.
Used on pages 12, 19, 20, 40, 44, 97, 98, 100, 102, 104, 110, 127, 226

$\hat{f}$ Arrival time function of travel time function $\hat{f}(\tau) = f(\tau) + \tau$.
Used on page 12

g A time-dependent travel time function.
Used on pages 44, 97, 98, 102, 110, 127

h A time-dependent travel time function.
Used on pages 98, 102

f^c A time-dependent cost function.
Used on pages 163, 164, 167–170

f^{tt} A time-dependent travel time function derived from ban intervals.
Used on pages 163, 164, 167, 168

Sequences, Tuples, and Intervals

A A sequence of arrivals for vertices of a path.
Used on pages 162, 164

D A sequence of departures for vertices of a path.
Used on pages 162, 164

G A graph ($G = (\mathcal{V}, \mathcal{E})$).
Used on pages 11, 15–17, 20–22, 24, 39, 41, 42, 49, 53, 67, 69, 74, 77, 78, 81–83, 104, 128, 144–146

$\overleftarrow{G}$ A reversed graph ($\overleftarrow{G} = (\mathcal{V}, \overleftarrow{\mathcal{E}})$).
Used on pages 11, 40

G^+ A CH augmented graph ($G = (\mathcal{V}, \mathcal{E}^+)$).
Used on pages 41, 42, 47, 52, 67, 68, 70, 75, 78, 87, 88, 90, 94, 95, 99, 104, 105, 130–132, 151

$G^{\uparrow}$ CH augmented graph with only upward edges ($G^{\uparrow} = (\mathcal{V}, \mathcal{E}^{\uparrow})$).
Used on pages 41–43, 47, 48, 68, 69, 71, 73–76, 78, 107, 131

$G^{\downarrow}$ CH augmented graph with only downward edges ($G^{\downarrow} = (\mathcal{V}, \mathcal{E}^{\downarrow})$).
Used on pages 41–43, 47, 48, 68, 71, 73–77, 107, 131, 132, 150, 151

G^{*} A CH augmented graph ($G = (\mathcal{V}, \mathcal{E}^{*})$).
Used on pages 70, 71, 78, 83, 87, 88, 94, 95

G_{e} A turn expanded graph ($G_{\mathrm{e}} = (\mathcal{V}_{\mathrm{e}} = \mathcal{E}, \mathcal{E}_{\mathrm{e}})$).
Used on pages 53, 81–83

G_{pre} A preprocessing-time graph in the CH-Potentials framework.
Used on pages 51–53

G_{q} A query-time graph in the CH-Potentials framework.
Used on pages 51–53

H Definition interval of periodic travel time functions, typically a day.
Used on pages 12, 44, 98, 104, 110, 126, 162–164

P A path in a graph, represented as sequence of vertices ($P = (v_0, \ldots, v_k)$).
Used on pages 11, 12, 21–23, 26, 27, 41, 80, 101, 109, 110, 113, 128, 144–146, 148–150, 153, 162, 164

T A timespan or time domain considered in an algorithm or problem.
Used on pages 12, 17–20, 22, 102, 110, 113, 127

V Validity interval for a CATCHUp expansion.
Used on pages 99–102, 113

Sets

$\mathcal{B}$ A subset of vertices used for experimental evaluation with terminals drawn from regions of specific sizes.
Used on pages 55–57, 88–90, 214, 219

$\mathcal{B}$ Set of time intervals where driving is not allowed on an edge.
Used on pages 161–163

$\mathcal{C}$ A set of vertices in a cell $\mathcal{C} \subseteq \mathcal{V}$.
Used on pages 78, 83

$\mathcal{E}$ The edges of a graph ($\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$).
Used on pages 11, 12, 15–17, 20–22, 24, 34, 35, 39, 40, 48, 51–53, 66, 77, 81, 99–102, 104, 109, 113, 127, 145, 146, 161–163

$\overleftarrow{\mathcal{E}}$ Set containing every edge of $\mathcal{E}$ in reverse direction.
Used on page 11

$\mathcal{E}^{+}$ Edges of a CH augmented graph.
Used on pages 41, 68, 70, 75, 78, 99, 102, 104, 105, 131

$\mathcal{E}^{\uparrow}$ Upward edges, i.e. edges where the head has higher rank than the tail, of a CH augmented graph.
Used on pages 41, 43, 47, 69, 71–73, 77, 104, 132

$\mathcal{E}^{\downarrow}$ Downward edges, i.e. edges where the head has lower rank than the tail, of a CH augmented graph.
Used on pages 41, 71–73, 104, 132

$\mathcal{E}^{*}$ Edges of a CCH minimal augmented graph.
Used on page 75

$\mathcal{N}$ The neighborhood of a vertex.
Used on page 11

$\overleftrightarrow{\mathcal{N}}$ The undirected neighborhood of a vertex.
Used on page 11

$\mathcal{P}$ The perimeter of cell $\mathcal{P}(\mathcal{C})$ contains vertices directly adjacent but not part of the cell.
Used on page 78

$\mathcal{S}$ A set of source vertices $\mathcal{S} \subseteq \mathcal{V}$.
Used on pages 43, 55–57, 214, 219

$\mathcal{T}$ A set of target vertices $\mathcal{T} \subseteq \mathcal{V}$.
Used on pages 77–79

$\mathcal{V}$ The vertices of a graph.
Used on pages 11, 15–17, 20–22, 24, 39–41, 43, 47, 51–53, 55, 68, 71–74, 77, 78, 81, 99, 104, 109, 127, 128, 145, 146, 161–164

$\mathcal{X}$ Time-dependent expansions of CATCHUp augmented graph edges.
Used on pages 99–102, 104, 105, 111, 113, 120, 226

Implementation

$\underline{\mathtt{b}}$ Scalar lower bound maintained for CATCHUp shortcuts.
Used on pages 99, 104, 105, 108–110, 131

$\overline{\mathtt{b}}$ Scalar upper bound maintained for CATCHUp shortcuts.
Used on pages 99, 104, 105, 108

`D` An array of length n containing distances for each vertex.
Used on pages 11, 39–41, 43, 44, 47–50, 76, 77, 108, 132, 133

`ET` An array of length n containing parents in the CCH elimination tree.
Used on pages 69, 70, 76, 77

`P` An array of length n containing parents in the shortest path tree.
Used on pages 40, 151

`Q` A priority queue (min-heap) for Dijkstra's algorithm.
Used on pages 39, 40

S A stack.
Used on pages 76, 77

Other Functions

π_t An A* potential/heuristic function estimating distances to the target t.
Used on pages 41, 48–50, 52, 58, 109, 110, 127–129, 131, 170

ϕ A function in the CH-potentials framework mapping query to preprocessing vertices.
Used on pages 52, 53

p A function mapping vertices to their parking area rating.
Used on pages 161–164, 170

deg The degree of a vertex, i.e. the number of directly reachable neighbors.
Used on pages 11, 71, 73, 74

$\overleftrightarrow{\deg}$ The undirected degree of a vertex, i.e. the number of adjacent vertices in either direction.
Used on pages 11, 48, 49

dist A minimal travel time or distance.
Used on pages 12, 15, 17, 20, 39, 41, 43, 46, 49, 50, 52, 53, 70, 71, 76, 78, 80, 105, 109, 110, 127, 129, 130, 132, 144, 146, 149, 150

$\text{dist}^{<}$ Shortest distance using only lower-ranked vertices.
Used on pages 41, 70, 71, 105, 132

exp Function to expand a CATCHUp expansion.
Used on pages 99–102, 113

opt A path of minimal travel time or distance.
Used on pages 144, 149, 151, 154, 156

UBS Uniformly Bounded Stretch of a path.
Used on pages 144–146, 152

Parameters

α A scaling factor for evaluating A* potentials.
Used on pages 57, 58

β Parameter for maximum number of breakpoints for CATCHUp approximation. Set to 1000 by default.
Used on pages 106, 111, 116, 117

δ Driving cost parameter for truck driver routing.
Used on pages 162–170, 173

ϵ Maximum error parameter for CATCHUp approximation. Set to 1 s by default.
Used on pages 106, 116, 117

ϵ A stretch or UBS limit parameter.
Used on pages 80, 144–148, 152–154, 156–158

η A tuning parameter for the parallelization of the CCH customization. Set to 32.
Used on page 74

κ A tuning parameter for the parallelization of the CCH customization. Set to 4.
Used on page 75

γ Duration of Interval-Minimum Potentials buckets. Set 15 minutes.
Used on page 131

λ Duration of the live window for Multi-Metric Potentials. Set to 59 minutes.
Used on page 130

ω_i Waiting cost parameter at vertex of rating i for truck driver routing.
Used on pages 162–169, 173

ψ The multiplicative penalization factor for edges on the shortest path in the penalty method. Set to 1.1.
Used on page 80

ψ_r The additive penalization factor for edges adjacent to the shortest path in the penalty method. Set to 0.01.
Used on page 80

A The Customizable Contraction Hierarchies Framework: Additional Experimental Results

A.1 Customization

Table A.1 depicts customization times on the Germany instance. We observe similar scaling behavior as with Europe and overall decent running times.

Table A.1: Running times by number of threads of different steps of the customization phase on Germany The experiment was conducted on our main benchmark machine for Chapter 7.

	Germany [s]			
Threads	Basic	Perfect	Construct	Total
1	3.22	3.59	1.52	8.32
2	1.76	1.81	0.76	4.33
4	1.05	0.96	0.41	2.42
8	0.75	0.54	0.27	1.56
16	0.55	0.29	0.24	1.08

A.2 Lazy RPHAST

Figure A.1 depicts Lazy RPHAST incremental query performance on the Germany instance. We observe marginal differences in absolute running times and otherwise only one significant difference: On small ball sizes, Lazy RPHAST on CCH is in both the elimination tree and the DFS variant faster than the CH variant.

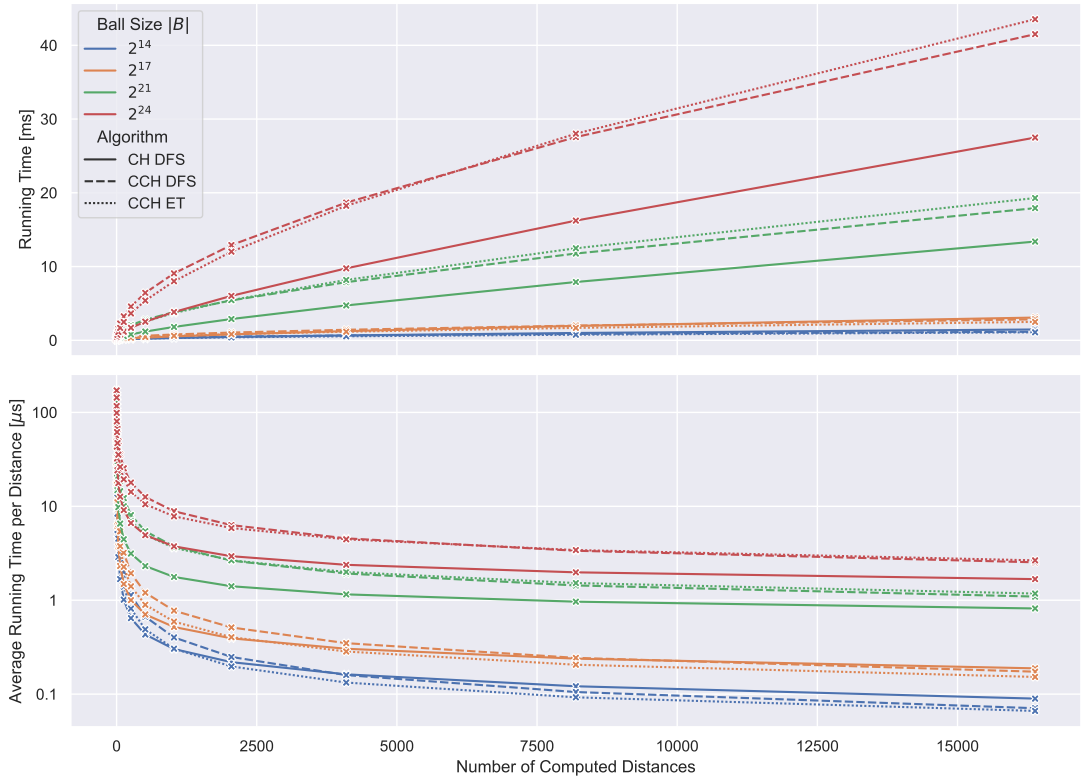


Figure A.1: Average running times of Lazy RPHAST on CH and CCH while incrementally querying $|\mathcal{S}| = 2^{14}$ sources from a ball of varying size $|\mathcal{B}|$ on Germany excluding selection times. The experiment was conducted on our main benchmark machine for Chapter 7. The upper figure contains the total elapsed running time. The lower figure contains the averaged running time per source, i.e. y/x from the upper figure. Note the different y-axis scales and units.

A.3 Nearest-Neighbor

Figure A.2 depicts nearest-neighbor query performance on the Germany instance. We observe no significant differences other than absolute running times. Table A.2, Figure A.3 and Figure A.3 depict the results of our reproduction of the experiments in [BW21]. We evaluated both our own implementation and the open-source implementation from [BW21]. We could fully reproduce the results observed in [BW21]. The direct comparison also clearly shows the impact of our optimizations.

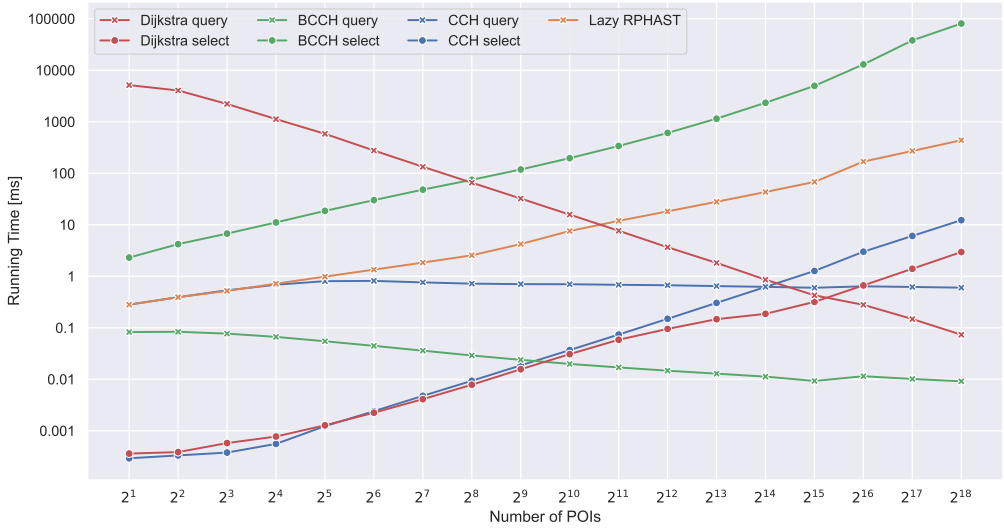


Figure A.2: Average running times for different nearest-neighbor algorithms on Germany to find the $k = 4$ closest targets from a POI set of varying size on our benchmark machine for Chapter 7.

Table A.2: Overview over the performance of various k-nearest-neighbor algorithms for different target distributions. We report the time to index a set of POIs (selection time), the space consumed by the index (selection space), and the time to find the $k = 1, 4, 8$ closest POIs (query time). The entire experiment was conducted on the same benchmark machine used in [BW21]. For CRP we list unscaled results as reported in [DW15]. LR is Lazy RPHAST. The [orig] row contains results for the CCH separator-based approach with the open-source implementation from [BW21] and [ours] is our own improved implementation.

	$\lvert P\rvert = 2^{12}, \lvert B\rvert = 2^{20}$					$\lvert P\rvert = 2^{14}, \lvert B\rvert = \lvert V\rvert$				
	selection		query time [μs]			Selection		Query time [μs]		
	Space [MB]	Time [ms]	POIs to be reported $k = 1$	$k = 4$	$k = 8$	Space [MB]	Time [ms]	POIs to be reported $k = 1$	$k = 4$	$k = 8$
Dijk.	2.2	0.1	712 763.1	723 458.3	730 865.8	2.2	0.2	165.6	665.7	1 346.0
CRP	–	–	–	–	–	0.0	8.0	–	640.0	–
[orig]	72.0	12.5	3 150.7	4 704.7	6 230.4	72.0	13.8	363.6	594.2	852.7
[ours]	0.0	0.1	278.6	311.7	338.4	0.0	0.6	359.0	445.0	532.5
LR	0.0	0.0	3 157.3	3 157.6	3 158.8	0.0	0.0	44 441.2	44 442.1	44 451.0
BCCH	89.7	324.6	25.8	27.8	29.5	141.5	1 327.3	7.4	12.5	16.6

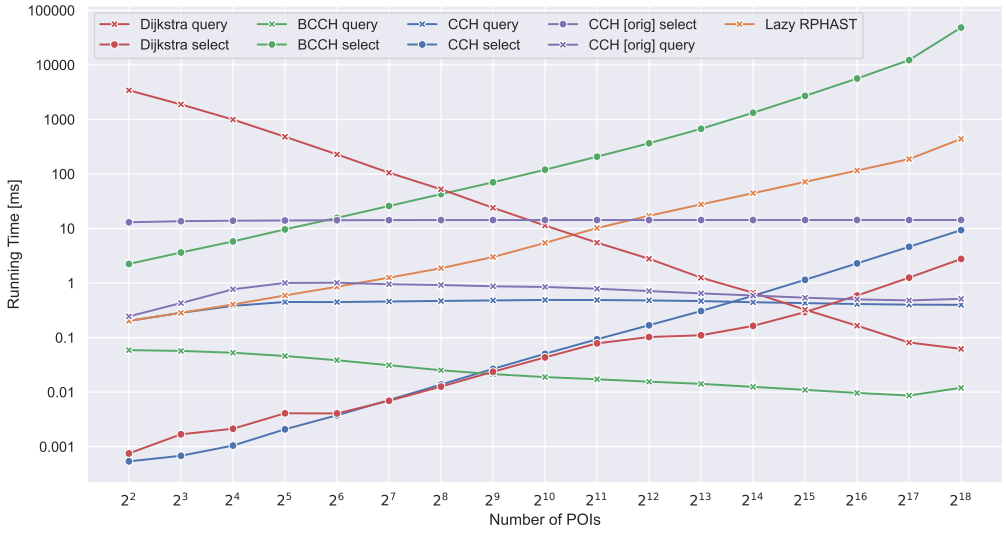


Figure A.3: Average running times for different nearest-neighbor algorithms on Europe to find the $k = 4$ closest targets from a POI set of varying size. The [orig] results were obtained with the open-source implementation form [BW21]. Also, the experiment was conducted on the same benchmark machine used in [BW21].

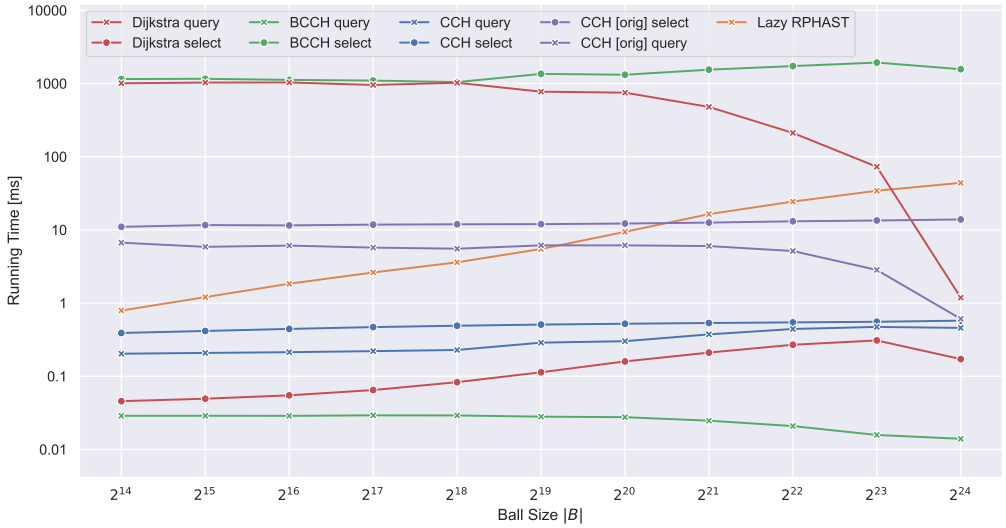


Figure A.4: Average running times for different nearest-neighbor algorithms on Europe to find the $k = 4$ closest targets from a POI set with 2^{14} targets picked from a ball of varying size. The [orig] results were obtained with the open-source implementation form [BW21]. Also, the experiment was conducted on the same benchmark machine used in [BW21].

A.4 Turn Costs

Table A.3 depicts characteristics of the full instance set used for the evaluation of our turn-aware CCH implementation. Europe Turns is the Europe subgraph induced by the largest strongly connected component in the turn-expanded graph which we used in [BWZZ20]. Tables A.4 and A.5 show running times for all phases and optimizations for the full instance set and confirm the observations presented in the main part of this work.

Table A.3: Additional road networks used for the evaluation of our turn cost optimizations.

	Source	Vertices $[\cdot 10^3]$	Edges $[\cdot 10^3]$	Turns $[\cdot 10^3]$	Turn data
Chicago	Transp. Networks	13.0	39.0	135.3	100 s U-Turns
London	PTV	37.0	85.5	137.2	Costs, Restrictions
Stuttgart	PTV	109.5	252.1	394.2	Costs, Restrictions
Germany	OSM	16 169.0	35 442.2	54 800.3	Restrictions
Europe Turns	DIMACS	17 350.0	39 936.5	106 371.3	100 s U-Turns
Europe	DIMACS	18 010.2	42 188.7	113 953.6	100 s U-Turns

Table A.4: Performance of different CCH variants and optimizations to support turn costs. We report the number of directed edges in the augmented graph and running times for each phase. Directed hierarchies imply the removal of infinite shortcuts and reordering separator vertices builds on directed hierarchies and the removal of infinite shortcuts and all three variants used a cut order. The experiment was conducted on the same benchmark machine used in [BW21].

		CCH Edges	Prepro.	Customization [ms]				Query [μs]	
		$[\cdot 10^3]$	[s]	Basic		Perfect		Basic	Perfect
	Threads			1	16	1	16		
Chicago	No turns	235.7	0.9	7.4	3.9	13.3	3.0	20.6	11.1
	CCH-Pot.	235.7	0.9	7.4	3.9	13.3	3.0	–	485.0
	Naive exp.	1 637.2	1.3	64.7	21.9	133.2	15.5	72.0	27.1
	Cut order	1 703.9	1.5	65.6	23.1	134.9	15.7	71.6	29.2
	Infinity	1 571.9	1.5	70.3	19.2	104.9	14.6	70.9	29.3
	Directed	873.5	1.5	38.4	11.6	103.3	16.4	40.4	40.4
	Reorder	737.0	1.5	27.8	7.2	66.0	8.5	29.6	35.6

Table A.5: Continuation of Table A.4.

		CCH Edges	Prepro.	Customization [ms]				Query [μs]	
		[$\cdot 10^3$]	[s]	Basic		Perfect		Basic	Perfect
	Threads			1	16	1	16		
London	No turns	363.7	0.6	9.9	3.5	16.6	3.0	20.9	11.8
	CCH-Pot.	363.7	0.6	9.9	3.5	16.6	3.0	–	720.5
	Naive exp.	1 534.2	1.1	48.0	15.2	108.2	12.3	58.9	22.3
	Cut order	1 680.5	1.5	53.7	16.5	114.8	13.0	61.1	26.0
	Infinity	1 447.7	1.5	51.8	13.9	82.5	11.3	60.3	26.0
	Directed	785.2	1.5	33.1	8.8	75.1	10.7	37.0	33.5
	Reorder	697.6	1.5	26.3	6.2	53.0	7.0	26.1	31.4
Stuttgart	No turns	724.2	0.9	15.9	3.9	24.5	3.8	17.5	10.2
	CCH-Pot.	724.2	0.9	15.9	3.9	24.5	3.8	–	321.1
	Naive exp.	3 214.9	2.4	72.8	14.0	158.4	16.6	44.2	18.0
	Cut order	3 359.7	1.1	74.9	13.8	159.9	17.3	42.7	18.1
	Infinity	2 874.6	1.2	69.4	11.7	121.3	13.8	42.2	18.2
	Directed	1 550.5	1.2	52.8	9.1	104.0	12.0	26.8	22.3
	Reorder	1 440.8	1.2	47.2	7.5	82.8	9.2	19.3	21.3
Germany	No turns	93 443.1	165.6	2 883.3	937.9	4 696.1	628.3	330.8	104.6
	CCH-Pot.	93 443.1	165.6	2 883.3	937.9	4 696.1	628.3	–	1 359.3
	Naive exp.	440 812.8	1 584.6	14 425.8	3 238.3	31 005.2	3 194.1	953.9	192.0
	Cut order	467 191.1	299.0	15 097.5	3 747.9	32 568.6	3 521.3	1 103.3	234.4
	Infinity	388 201.7	318.5	15 174.2	3 339.2	24 890.1	2 975.8	1 104.3	234.5
	Directed	206 970.0	319.6	9 754.1	2 403.7	23 459.0	2 573.2	586.0	337.6
	Reorder	190 597.1	319.8	8 273.4	2 050.3	17 331.0	2 018.6	408.2	347.9
Europe Turns	No turns	107 042.6	202.3	3 095.6	1 092.0	5 479.4	763.1	246.8	102.3
	CCH-Pot.	107 042.6	202.3	3 095.6	1 092.0	5 479.4	763.1	–	3 783.6
	Naive exp.	622 426.7	2 474.0	17 759.1	4 041.4	38 486.3	4 070.2	712.3	170.4
	Cut order	663 587.3	300.3	18 112.5	3 983.6	38 566.7	4 126.7	786.8	190.9
	Infinity	585 062.3	319.3	18 424.2	3 590.4	32 506.1	3 686.3	781.7	190.5
	Directed	326 627.9	322.7	12 693.0	2 747.8	29 559.5	3 137.0	419.8	273.5
	Reorder	302 307.5	323.1	10 620.0	2 331.7	22 436.9	2 588.7	301.8	304.5
Europe	No turns	117 727.5	263.5	3 366.5	982.4	6 070.5	767.8	189.3	89.1
	CCH-Pot.	117 727.5	263.5	3 366.5	982.4	6 070.5	767.8	–	2 311.1
	Naive exp.	692 995.8	2 873.9	19 670.2	3 590.2	43 816.2	4 523.2	575.6	159.3
	Cut order	737 433.4	327.2	20 082.1	3 567.5	44 365.9	4 597.9	584.4	167.9
	Infinity	651 921.7	355.1	20 145.2	3 721.1	36 436.2	3 918.9	582.7	167.2
	Directed	363 663.3	359.7	13 797.5	2 808.6	33 040.0	3 495.1	645.8	260.9
	Reorder	334 755.9	357.0	11 686.7	2 450.2	25 097.4	2 889.0	469.0	299.2

B CH-Potentials and CCH-Potentials in Comparison

Figure B.1 and Tables B.1 to B.4 depict experimental results on CH-Potentials from Chapter 6. However, they also include running times for CCH-Potentials realized with the elimination tree-based Lazy RPHAST. Generally, CH and CCH-Potentials perform similarly. Usually, CCH-Potentials are slightly slower. Only in the bidirectional $\ell_q = 1.05 \cdot \ell_{pre}$ case, the CCH variant is marginally faster. This is caused by the varying performance characteristics of Lazy RPHAST depending on the number of queried distances; compare Figure A.1

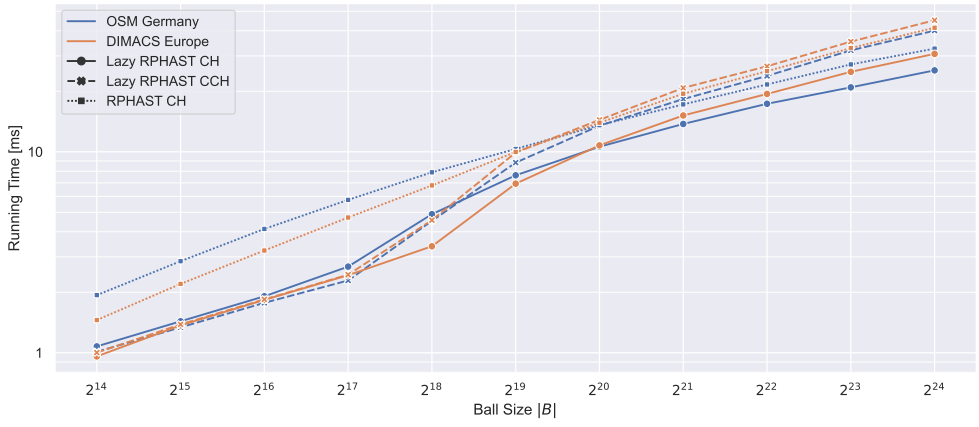


Figure B.1: Running times of (C)CH-based Lazy RPHAST and CH-based RPHAST for many-to-one queries with $|\mathcal{S}| = 2^{14}$ sources picked from a ball of varying size $|\mathcal{B}|$ including both the selection and the time to compute all distances. The experiment was conducted on our benchmark machine for Chapter 6.

Figure B.1 also includes running times of our own implementation of RPHAST [DGW11] for a more accurate comparison. These confirm our statements on the relative performance difference between Lazy RPHAST and RPHAST and also roughly reproduce the results from [DGW11].

Table B.1: Average query running times and number of queue pushes with different heuristics and optimizations on Germany with $\ell_q = 1.05 \cdot \ell_{pre}$. The experiment was conducted on our main benchmark machine for Chapter 6.

	BCC	Deg2	Deg3	Zero	ALT	CH	CCH	Oracle
Running time [ms]	○	○	○	2 133.0	317.9	47.9	54.4	34.3
	●	○	○	1 355.3	233.9	36.3	38.5	24.8
	●	●	○	753.4	122.6	19.5	22.1	12.7
	●	●	●	580.7	90.8	15.9	18.1	10.1
Queue [$\cdot 10^3$]	○	○	○	8 087.1	863.1	137.1	137.1	137.1
	●	○	○	6 298.2	685.7	112.7	112.7	112.7
	●	●	○	2 901.4	303.4	43.3	43.3	43.3
	●	●	●	1 681.4	179.7	26.8	26.8	26.8

Table B.2: Performance of different variants of bidirectional A* on OSM Ger with $\ell_q = 1.05 \cdot \ell_{pre}$. All variants alternate between the forward and the backward search. The experiment was conducted on our main benchmark machine for Chapter 6.

			Running time [ms]					Queue pushes [$\cdot 10^3$]		
Low Deg. Opt.	Bidir. Pot.	New Pruning	Zero	ALT	CH	CCH	Oracle	Zero	ALT	(C)CH/ Oracle
○	Avg.	○	1 441.41	126.46	62.61	53.91	37.29	4 493.97	292.01	125.16
○	Avg.	●	1 451.96	128.20	62.48	54.28	38.89	4 491.56	290.92	125.08
○	Sym.	○	5 779.64	795.56	122.70	111.78	88.66	16 042.82	1 688.60	259.78
○	Sym.	●	1 453.58	261.80	59.22	51.97	37.37	4 491.56	624.25	116.71
●	Avg.	○	365.82	33.22	19.34	18.66	9.96	916.15	57.27	23.60
●	Avg.	●	369.51	33.37	19.54	18.88	9.98	908.55	56.09	23.25
●	Sym.	○	1 512.48	241.27	40.98	38.99	26.36	3 317.81	334.90	44.67
●	Sym.	●	368.94	72.67	21.54	20.39	11.22	908.55	123.77	20.72

Table B.3: Performance of different direction selection criteria of bidirectional A* on OSM Ger with different query weights. The symmetric variant uses the improved pruning, the average variant does not. All variants use all low degree optimizations. The experiment was conducted on our main benchmark machine for Chapter 6.

			Running time [ms]					Queue pushes [$\cdot 10^3$]		
ℓ_q	Bidir. Pot.	Choose Direction	Zero	ALT	CH	CCH	Oracle	Zero	ALT	(C)CH/ Oracle
ℓ_{pre}	Avg.	Alternating	373.18	12.83	0.79	1.13	0.18	916.15	23.08	0.60
	Avg.	Min. Key	406.35	13.68	1.44	1.75	0.56	986.40	26.39	1.15
	Sym.	Alternating	376.72	40.19	0.69	0.92	0.19	908.55	76.61	0.57
	Sym.	Min. Key	427.51	50.46	1.77	1.99	0.83	978.62	99.62	1.44
$\ell_{pre} \cdot 1.05$	Avg.	Alternating	365.82	33.22	19.34	18.66	9.96	916.15	57.27	23.60
	Avg.	Min. Key	391.70	38.06	21.76	20.44	11.30	986.41	67.65	26.42
	Sym.	Alternating	368.94	72.67	21.54	20.39	11.22	908.55	123.77	20.72
	Sym.	Min. Key	394.38	84.84	27.28	24.64	14.53	978.63	145.28	24.82
$\ell_{pre} \cdot 1.5$ if speed $<$ 80kph	Avg.	Alternating	361.83	19.50	10.92	10.94	5.34	845.06	34.03	13.25
	Avg.	Min. Key	391.47	31.65	21.05	20.10	11.00	917.13	52.23	23.78
	Sym.	Alternating	364.55	37.33	11.89	11.75	6.00	836.44	57.93	11.53
	Sym.	Min. Key	395.04	54.90	23.36	22.48	12.54	908.12	84.33	22.01

Table B.4: Performance of bidirectional and unidirectional A* on OSM Ger with different query weights. The symmetric variant uses the improved pruning, the average variant does not. All variants use all low degree optimizations. The experiment was conducted on our main benchmark machine for Chapter 6.

		Running time [ms]					Queue pushes [$\cdot 10^3$]		
ℓ_q		Zero	ALT	CH	CCH	Oracle	Zero	ALT	(C)CH/ Oracle
ℓ_{pre}	Unidirectional	584.87	43.02	0.47	0.64	0.16	1 674.35	96.21	0.66
	Average	373.18	12.83	0.79	1.13	0.18	916.15	23.08	0.60
	Symmetric	376.72	40.19	0.69	0.92	0.19	908.55	76.61	0.57
$\ell_{pre} \cdot 1.05$	Unidirectional	580.66	90.79	15.91	18.09	10.06	1 681.39	179.66	26.78
	Average	365.82	33.22	19.34	18.66	9.96	916.15	57.27	23.60
	Symmetric	368.94	72.67	21.54	20.39	11.22	908.55	123.77	20.72
$\ell_{pre} \cdot 1.5$ if speed $<$ 80kph	Unidirectional	637.24	96.62	21.78	21.37	14.62	1 674.26	171.02	36.54
	Average	361.83	19.50	10.92	10.94	5.34	845.06	34.03	13.25
	Symmetric	364.55	37.33	11.89	11.75	6.00	836.44	57.93	11.53

C CATCHUp: Additional Experimental Results

Table C.1 contains preprocessing results for all graphs. The other days for Ger06 and Ber behave roughly as expected. The weekend instances feature less time-dependent edges and preprocessing accordingly runs faster. SynEur with medium and high traffic produces some surprising results regarding the unpacking data. Even the medium traffic instance has a higher average number of expansions than Eur17. With high traffic, the number is even greater than on Eur20. The number of edges with only a single expansion is correspondingly small. We suspect that the reason for this are the extreme relative delays of the predicted travel times. These extreme fluctuations lead to many shortest path changes despite the little amount of time-dependent information. The instances derived from raw traffic observation (Mun and Ger19) have the largest averages of expansions per edge. But even for the Mun instance, the number of edges with only one expansion is still 90%. This suggests that our approach is robust.

The results for query experiments on all instances reported in Table C.2 also confirm our observations from the main part of this article. Each optimization yields similar accelerations. Again, SynEur exhibits surprising behavior. On the one hand, unoptimized queries are surprisingly fast, i.e., up to four times faster than on Eur17. On the other hand, with all optimizations, SynEur with high traffic has the slowest query times among all instances. Again, the reason is the combination of few time-dependent edges with high relative delays. Because there is little time-dependent information in the instance, the basic query algorithm is not as slow as one could expect. However, because of the high delays, the corridor search and the A* optimizations are not as effective. In [NDSL12] it is stated that unimportant edges (with respect to a Highway Hierarchy) will never get a non-constant travel time function. In combination with the high relative delays, detours through unimportant parts of the network can often become the shortest paths. This also decreases the effectiveness of our A* optimization.

Table C.1: Preprocessing statistics. Running times are for parallel execution on 16 cores.

		CCH edges	Expansions per edge			Data	Running time [s]	
		[$\cdot 10^3$]	Avg.	Max.	= 1 [%]	[GB]	Prepr.	Custom.
Mun	Tuesday	125	2.087	129	89.9	0.01	0.6	1.3
Ber	Monday	1 977	1.040	25	98.6	0.09	1.5	6.2
	Tuesday	1 977	1.039	31	98.6	0.09	1.5	6.2
	Wednesday	1 976	1.038	19	98.6	0.09	1.5	6.2
	Thursday	1 977	1.039	23	98.6	0.09	1.6	6.2
	Friday	1 975	1.037	28	98.7	0.09	1.5	5.8
	Saturday	1 961	1.023	21	99.1	0.09	1.5	3.8
	Sunday	1 957	1.021	27	99.2	0.09	1.6	3.3
Ger06	Monday	22 499	1.073	42	98.4	1.06	30.0	20.9
	midweek	22 519	1.075	44	98.4	1.06	30.1	21.6
	Friday	22 445	1.064	43	98.6	1.05	30.2	17.2
	Saturday	22 229	1.031	37	99.2	1.03	30.2	6.0
	Sunday	22 128	1.019	39	99.5	1.02	29.8	3.6
SynEur	Low	88 884	1.036	23	99.2	4.14	238.3	82.7
	Medium	90 514	1.109	24	97.6	4.31	231.5	224.8
	High	94 302	1.264	31	94.6	4.71	233.3	523.0
Ger17	Tuesday	31 488	1.090	107	98.5	1.50	35.0	107.4
Eur17	Tuesday	114 857	1.099	115	98.4	5.47	189.6	557.0
Ger19	Tuesday	75 800	1.668	369	96.1	4.30	135.7	11 581.1
Eur20	Tuesday	128 921	1.191	109	96.9	6.32	209.6	1 039.5

Table C.3 contains profile query results. The results mostly conform to the already reported observations, Mun behaves like Ger19 but smaller, and SynEur deviates. Here, the SynEur results are particularly surprising. While the travel time profiles are comparatively simple because of the low complexity of the input functions, the number of path switches is so high that we suspected bugs as the cause. In addition, it decreases as the amount of traffic increases. Nevertheless, we claim that the numbers are correct and that the reason lies in the combination of high relative delays with few time-dependent edges. When there are few time-dependent edges and the slowdown due to a predicted traffic jam on an edge is very high, there will always be a faster detour using less important arcs without travel time predictions. This leads to the extremely high number of switches and distinct paths. As the amount of time-dependent edges is increased, the spatial consistency increases and an increasing amount of detours will now also have an increased travel time. Thus, the number of path switches decreases.

Table C.2: Query performance with different optimizations. We report the number of nodes popped from the queue, the number of evaluated travel time functions and the running time. All values are arithmetic means over 100 000 queries executed in bulk with source, target and departure time drawn uniformly at random.

		Queue pops				Evaluated travel time functions				Running time [ms]			
		Basic	+ Cor.	+ Lazy	+ A*	Basic	+ Cor.	+ Lazy	+ A*	Basic	+ Cor.	+ Lazy	+ A*
Mun	Tue.	77.5	58.8	1 314.7	209.3	11 241.9	5 517.5	1 677.3	321.0	0.6	0.3	0.3	0.1
Ber	Mon.	167.4	38.2	1 605.0	618.6	99 629.0	5 480.3	1 762.4	674.9	8.6	0.6	0.6	0.3
	Tue.	167.4	38.1	1 603.6	635.2	100 820.5	5 224.1	1 747.4	691.5	8.8	0.6	0.6	0.3
	Wed.	167.4	38.6	1 640.4	643.7	101 938.1	5 405.3	1 786.6	702.0	8.9	0.6	0.7	0.3
	Thu.	167.4	38.9	1 647.9	642.5	101 584.1	5 498.9	1 799.8	701.8	8.8	0.6	0.7	0.3
	Fri.	167.4	37.8	1 591.1	619.6	99 142.5	5 061.0	1 722.9	674.0	8.5	0.5	0.6	0.3
	Sat.	167.1	26.1	926.5	491.5	86 470.8	2 124.5	967.4	514.1	6.4	0.3	0.3	0.2
	Sun.	167.1	24.7	864.1	476.3	84 796.6	1 865.9	895.3	495.1	6.1	0.2	0.3	0.2
Ger06	Mon.	492.3	68.6	2 649.0	727.0	751 679.3	22 542.0	3 029.0	853.0	42.8	1.8	1.4	0.5
	midw.	492.3	79.7	3 323.2	831.0	818 721.3	31 740.8	3 838.0	995.1	46.4	2.3	1.7	0.6
	Fri.	491.9	62.9	2 349.2	731.3	780 031.8	21 423.2	2 665.4	848.3	42.6	1.6	1.2	0.5
	Sat.	490.7	24.1	339.4	211.1	541 331.4	2 457.5	360.0	223.6	26.8	0.4	0.3	0.2
	Sun.	490.0	20.2	219.2	163.5	503 009.4	1 599.4	226.8	169.1	24.2	0.3	0.2	0.2
SynEur	Low	742.8	341.3	6 626.2	1 704.8	4 871 967.5	997 409.9	16 521.1	4 730.7	201.8	39.9	5.1	2.0
	Med.	746.8	461.8	17 209.3	3 796.9	5 742 442.6	1 596 401.3	35 066.3	9 389.8	253.0	69.4	13.1	4.0
	High	749.7	554.1	33 572.2	7 018.4	6 142 257.3	2 031 399.6	60 234.2	15 685.2	289.2	96.4	25.5	6.9
Ger17	Tue.	510.3	143.4	18 450.0	3 099.2	2 100 731.8	164 372.5	19 910.5	3 495.5	169.7	13.7	9.1	1.7
Eur17	Tue.	861.6	229.3	39 714.8	6 876.5	9 951 623.1	806 727.8	43 581.1	7 911.0	808.6	62.3	20.8	4.1
Ger19	Tue.	894.2	618.1	193 779.2	23 218.1	40 675 596.0	11 563 911.1	220 416.1	29 816.5	4 329.0	1 166.1	151.2	16.5
Eur20	Tue.	871.0	335.6	62 677.7	7 231.9	10 527 072.7	1 222 655.6	70 145.4	8 844.7	813.2	92.9	33.7	4.7

Table C.3: Running times of profile queries and characteristics of the obtained profiles. We report total running times and running times of each phase (Corridor, Reconstruction, Contraction, Extraction) of the query. The total running time is slightly larger than the sum of all phases as it includes some additional initialization and cleanup work. We report the number of breakpoints in the obtained travel time profile (Column $|f|$). Column $|\mathcal{X}|$ contains the number of times the shortest path changes during the day. Since the same path may be the fastest for several times, we also report the number of distinct paths. All values are averages over 1000 random queries.

		Running time [ms]								Distinct
		Corridor	Reconstruct	Contract	Profile	Paths	Total	$\|\ell\|$	$\|\mathcal{X}\|$	paths
Mun	Tuesday	0.0	189.6	106.9	0.6	1.7	301.1	5 702.7	82.2	19.4
Ber	Monday	0.1	39.7	14.9	2.6	0.4	58.4	29 090.5	2.7	2.3
	Tuesday	0.1	37.8	13.9	2.8	0.4	55.6	30 974.9	2.7	2.3
	Wednesday	0.1	38.4	14.1	2.8	0.4	56.5	31 126.2	2.6	2.2
	Thursday	0.1	40.1	14.8	2.8	0.4	58.8	30 662.1	2.7	2.3
	Friday	0.1	34.2	12.5	2.4	0.3	50.2	27 671.5	2.6	2.2
	Saturday	0.1	11.2	3.4	1.5	0.2	16.6	17 892.8	1.6	1.8
	Sunday	0.1	8.9	2.5	1.4	0.2	13.2	16 768.7	1.6	1.8
Ger06	Monday	0.3	42.2	17.3	0.6	0.8	62.9	9 036.7	7.0	3.1
	midweek	0.4	56.5	23.0	0.7	0.8	83.6	9 359.6	6.9	3.3
	Friday	0.3	32.4	14.0	0.5	0.7	49.3	7 896.2	6.0	3.0
	Saturday	0.2	1.7	0.9	0.1	0.4	3.6	2 047.2	2.8	2.0
	Sunday	0.2	0.9	0.4	0.1	0.3	2.1	1 386.3	2.2	1.8
SynEur	Low	2.1	591.3	501.7	0.2	7.3	1 128.2	3 379.4	151.4	144.9
	Medium	3.1	1 664.3	962.7	0.4	6.1	2 694.1	5 226.1	99.0	90.1
	High	3.2	3 521.5	1 484.9	0.5	5.3	5 102.2	5 939.1	78.0	69.2
Ger17	Tuesday	0.8	452.2	189.0	6.1	2.4	660.1	66 146.0	9.7	3.7
Eur17	Tuesday	1.7	1 135.8	732.9	13.0	7.8	1 913.2	122 192.1	16.5	6.8
Ger19	Tuesday	2.3	34 345.4	230 643.8	79.3	51.8	265 453.7	525 196.4	91.4	34.3
Eur20	Tuesday	2.8	3 166.6	1 507.7	12.3	10.3	4 747.5	107 690.7	24.4	11.6

D Time-Dependent A* Potentials: Additional Experimental Results

Here, we report the performance of all our time-dependent A* potentials functions on all time-dependent problem instances: Table D.1 contains results for CH-Potentials, Table D.2 for CCH-Potentials, Table D.3 for MMP, and Table D.4 for IMP. The experiments were conducted on the benchmark machine of Chapter 9. The results confirm our observations from Chapter 9. Further we note, that our potentials work best on graphs with traffic predictions derived from traffic models such as the PTV instances. Instances using raw trace speeds (Mun and Ger19) are harder but still manageable. The SynEur instance with its short simulated traffic jams in the predictions has a completely different structure than what our potentials are tuned to. This is the reason why we do not achieve good speedups on SynEur.

Table D.1: Query and preprocessing performance results of CH-Potentials on different graphs and traffic predictions. We report average running times, number of queue pops, relative increases of the result distance over the initial distance estimate and speedups over Dijkstra's algorithm for 10 k random queries. Additionally, we report the total preprocessing time and the memory consumption of precomputed auxiliary data.

		Running time [ms]	Queue [$\cdot 10^3$]	Length incr. [%]	Speedup	Prepro. [s]	Space [GB]
Mun	Tuesday	0.10	0.16	8.66	18.31	0.14	0.00
Ber	Monday	0.68	0.67	3.47	80.20	2.97	0.02
	Tuesday	0.77	0.77	3.75	69.45	2.95	0.02
	Wednesday	0.83	0.83	3.91	65.10	2.96	0.02
	Thursday	0.80	0.81	3.83	70.42	2.98	0.02
	Friday	0.73	0.72	3.51	75.60	2.97	0.02
	Saturday	0.38	0.37	1.97	129.57	2.90	0.02
	Sunday	0.34	0.32	1.77	140.53	2.92	0.02
Ger06	Monday	3.59	4.00	2.71	147.97	56.76	0.21
	midweek	4.58	5.37	3.07	115.50	56.85	0.21
	Friday	4.38	5.18	2.90	115.72	56.53	0.21
	Saturday	0.87	0.65	0.90	588.31	56.57	0.21
	Sunday	0.65	0.41	0.59	768.09	56.06	0.21
SynEur	Low	219.37	342.42	8.11	10.07	295.09	0.78
	Medium	267.13	416.38	10.58	8.13	297.55	0.78
	High	306.97	455.24	12.36	7.43	295.19	0.78
Ger17	Tuesday	16.68	15.83	5.28	62.98	63.71	0.30
Eur17	Tuesday	87.23	77.50	3.84	48.01	333.61	1.08
Ger19	Tuesday	134.93	92.79	12.21	24.20	289.97	0.68
Eur20	Tuesday	97.45	66.40	4.16	61.77	361.79	1.19

Table D.2: Query and preprocessing performance results of CCH-Potentials on different graphs and traffic predictions. We report average running times, number of queue pops, relative increases of the result distance over the initial distance estimate and speedups over Dijkstra's algorithm for 10 k random queries. Additionally, we report the total preprocessing time and the memory consumption of precomputed auxiliary data.

		Running time [ms]	Queue [$\cdot 10^3$]	Length incr. [%]	Speedup	Prepro. [s]	Space [GB]
Mun	Tuesday	0.09	0.16	8.66	20.18	0.19	0.00
Ber	Monday	0.64	0.67	3.47	84.88	1.68	0.03
	Tuesday	0.73	0.77	3.75	73.59	1.71	0.03
	Wednesday	0.78	0.83	3.91	68.91	1.71	0.03
	Thursday	0.76	0.81	3.83	74.45	1.60	0.03
	Friday	0.68	0.72	3.51	80.70	1.70	0.03
	Saturday	0.34	0.37	1.97	144.44	1.84	0.03
	Sunday	0.30	0.32	1.77	156.67	1.73	0.03
Ger06	Monday	3.06	4.00	2.71	173.65	35.59	0.32
	midweek	3.98	5.37	3.07	132.81	35.81	0.32
	Friday	3.73	5.18	2.90	135.60	35.01	0.32
	Saturday	0.68	0.65	0.90	743.67	35.06	0.32
	Sunday	0.50	0.41	0.59	988.45	34.16	0.32
SynEur	Low	213.98	342.42	8.11	10.32	261.82	1.27
	Medium	258.08	416.38	10.58	8.41	252.94	1.27
	High	300.48	455.24	12.36	7.59	257.44	1.27
Ger17	Tuesday	16.37	15.83	5.28	64.15	40.50	0.45
Eur17	Tuesday	84.96	77.50	3.84	49.29	226.04	1.65
Ger19	Tuesday	126.03	92.79	12.21	25.91	156.13	1.06
Eur20	Tuesday	94.96	66.40	4.16	63.39	244.63	1.80

Table D.3: Query and preprocessing performance results of MMP on different graphs and traffic predictions. We report average running times, number of queue pops, relative increases of the result distance over the initial distance estimate and speedups over Dijkstra's algorithm for 100 k random queries. Additionally, we report the total preprocessing time and the memory consumption of precomputed auxiliary data.

		Running time [ms]	Queue [$\cdot 10^3$]	Length incr. [%]	Speedup	Prepro. [s]	Space [GB]
Mun	Tuesday	0.11	0.15	7.88	17.78	0.44	0.06
Ber	Monday	0.44	0.34	1.99	125.50	3.99	0.87
	Tuesday	0.48	0.38	2.12	112.61	4.07	0.87
	Wednesday	0.50	0.41	2.18	108.74	4.03	0.87
	Thursday	0.49	0.40	2.16	116.17	3.94	0.87
	Friday	0.46	0.36	1.96	120.70	4.03	0.87
	Saturday	0.30	0.23	1.17	163.65	4.12	0.87
	Sunday	0.28	0.21	1.05	167.64	3.99	0.87
Ger06	Monday	1.84	1.80	1.55	289.72	61.39	9.86
	midweek	2.14	2.26	1.69	247.60	63.81	9.87
	Friday	2.00	2.12	1.66	253.30	60.17	9.83
	Saturday	0.65	0.37	0.48	783.30	59.88	9.72
	Sunday	0.56	0.28	0.30	886.24	59.25	9.66
SynEur	Low	224.76	333.02	8.00	9.83	398.73	38.95
	Medium	295.18	416.91	10.41	7.36	367.32	39.73
	High	319.64	453.30	12.16	7.14	392.41	41.49
Ger17	Tuesday	9.56	8.51	3.26	109.88	76.43	13.84
Eur17	Tuesday	79.35	69.24	3.27	52.78	383.28	50.64
Ger19	Tuesday	107.87	74.17	9.97	30.27	271.56	33.71
Eur20	Tuesday	89.04	58.84	3.48	67.61	426.18	56.62

Table D.4: Query and preprocessing performance results of IMP on different graphs and traffic predictions. We report average running times, number of queue pops, relative increases of the result distance over the initial distance estimate and speedups over Dijkstra's algorithm for 10 k random queries. Additionally, we report the total preprocessing time and the memory consumption of precomputed auxiliary data.

		Running time [ms]	Queue [$\cdot 10^3$]	Length incr. [%]	Speedup	Prepro. [s]	Space [GB]
Mun	Tuesday	0.12	0.07	2.57	15.54	1.89	0.05
Ber	Monday	0.42	0.12	0.29	130.15	12.33	0.80
	Tuesday	0.43	0.12	0.31	125.95	11.99	0.80
	Wednesday	0.43	0.12	0.31	126.99	14.03	0.80
	Thursday	0.43	0.12	0.31	132.67	14.19	0.80
	Friday	0.42	0.12	0.29	130.65	13.27	0.80
	Saturday	0.36	0.11	0.16	134.72	9.65	0.79
	Sunday	0.36	0.11	0.15	131.52	10.83	0.79
Ger06	Monday	1.60	0.25	0.20	332.61	92.05	9.08
	midweek	1.68	0.26	0.21	315.61	94.98	9.08
	Friday	1.55	0.23	0.17	326.73	88.71	9.05
	Saturday	1.17	0.18	0.05	434.81	94.84	8.97
	Sunday	1.11	0.18	0.03	446.56	85.88	8.93
SynEur	Low	115.35	88.18	1.29	19.15	590.04	35.84
	Medium	219.97	158.70	2.03	9.87	656.60	36.50
	High	285.18	193.17	2.67	8.00	1 042.24	36.50
Ger17	Tuesday	2.39	0.32	0.31	439.72	230.84	12.72
Eur17	Tuesday	8.50	1.28	0.27	492.69	1 097.59	46.43
Ger19	Tuesday	20.89	5.08	1.76	156.29	13 111.69	30.66
Eur20	Tuesday	10.91	1.73	0.36	551.95	1 610.65	52.09

E Smooth Path Performance Profiles

Here, we show performance profiles broken down by instance and query set. Figure E.1 contains the running time profile for dynamic smooth paths and Figure E.2 the quality profile. Figure E.3 contains the running time profile for time-dependent smooth paths and Figure E.4 the quality profile. Finally, Figure E.5 shows the aggregated profile for time-dependent smooth paths.

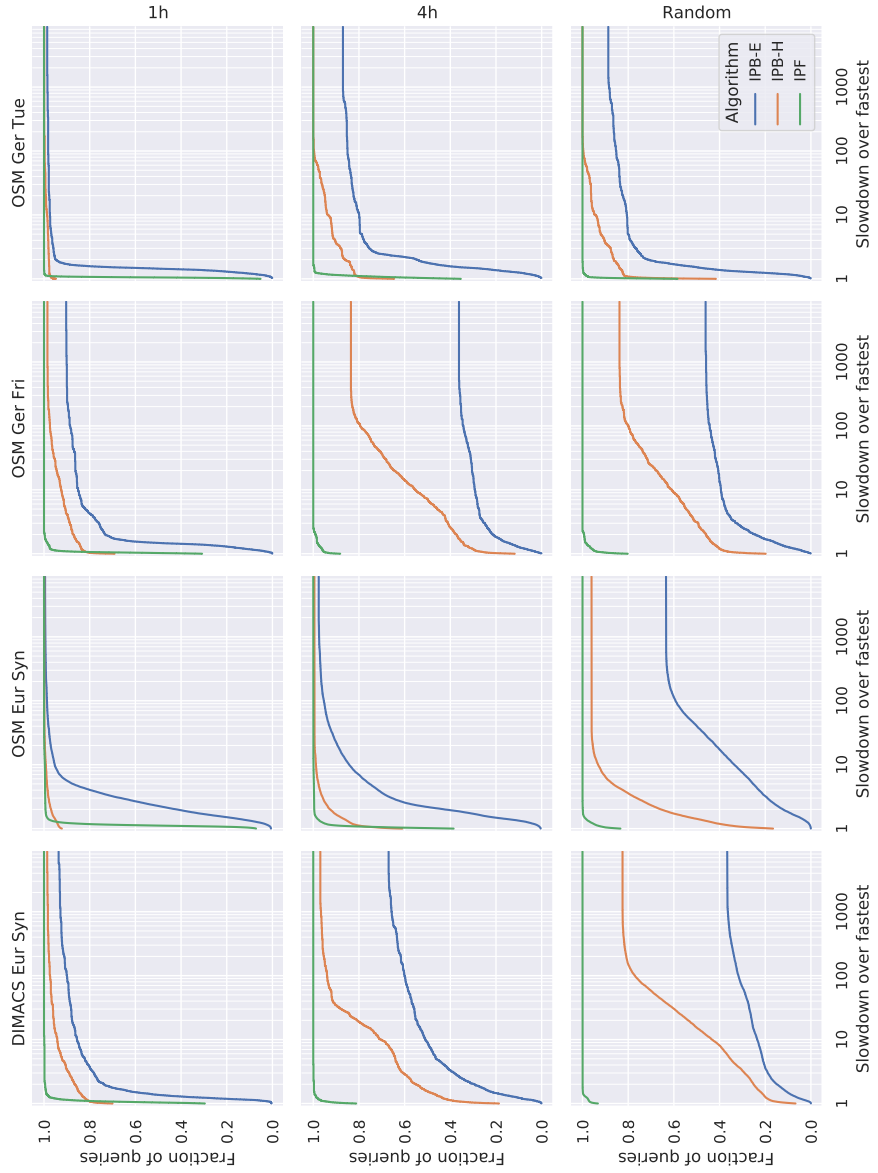


Figure E.1: Relative performance profile for the running time of our algorithms for dynamic smooth paths on all queries from Table 10.1 split by graph and query set.

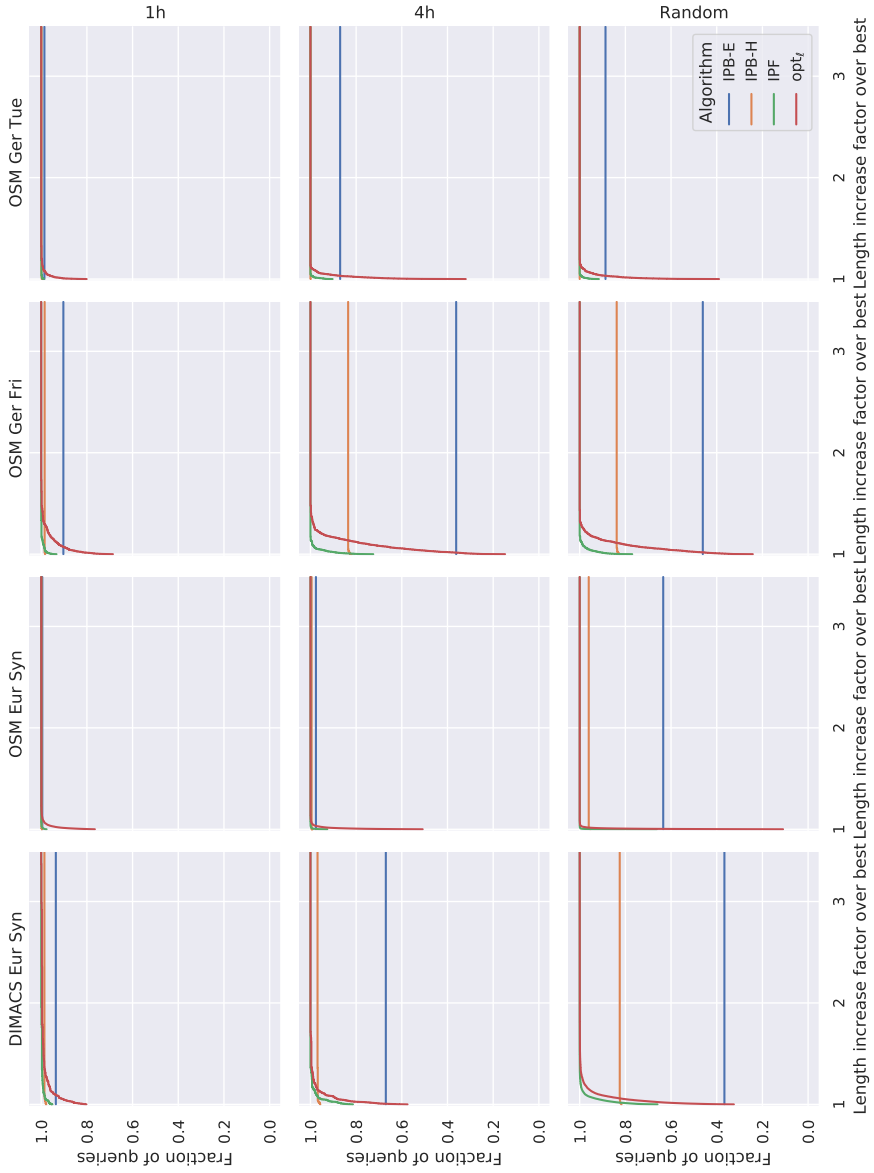


Figure E.2: Relative performance profile for solution quality of our algorithms for dynamic smooth paths on all queries from Table 10.1 split by graph and query set.

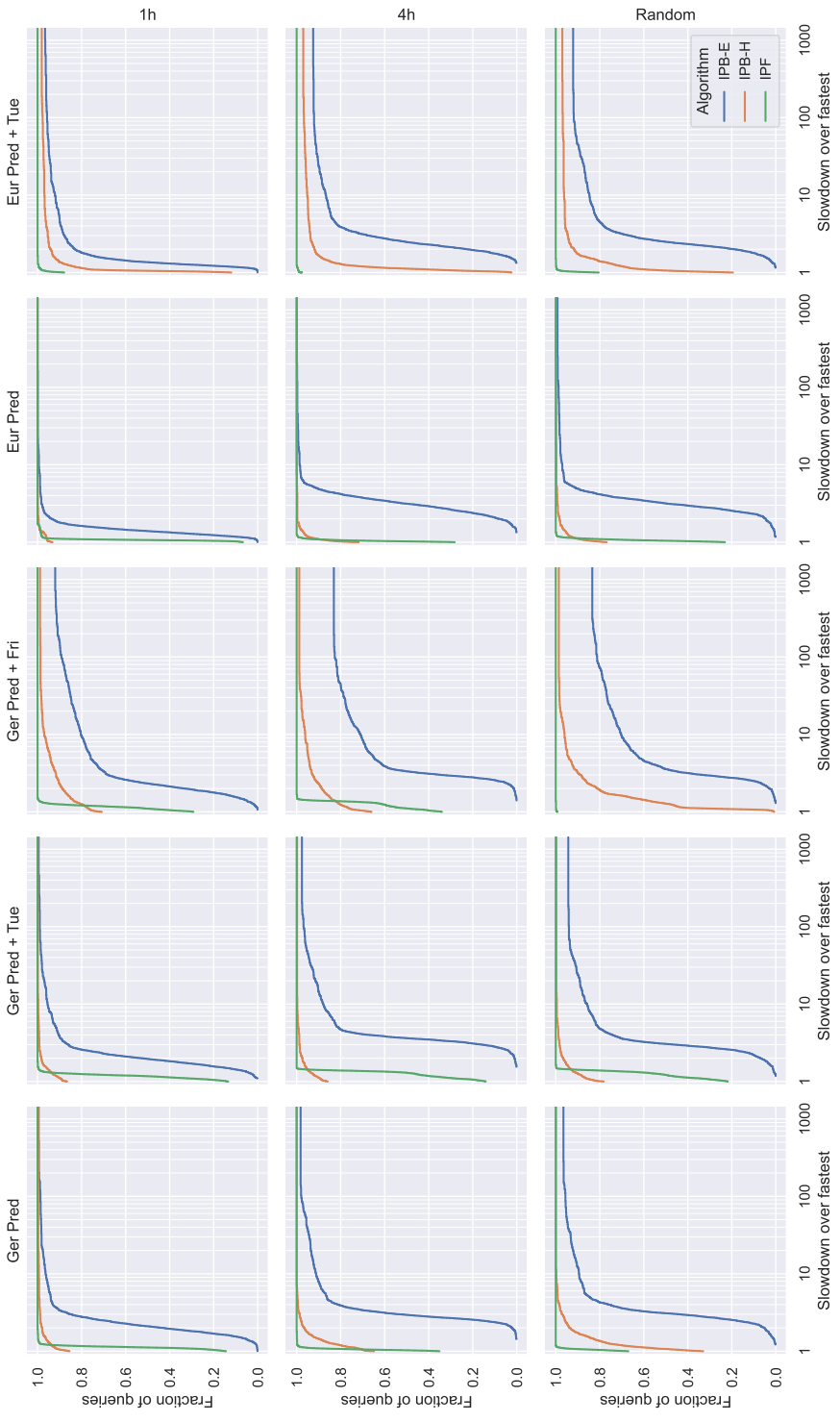


Figure E.3: Relative performance profile for the running time of our algorithms for time-dependent and combined dynamic and time-dependent smooth paths on all queries from Table 10.2 split by graph and query set.

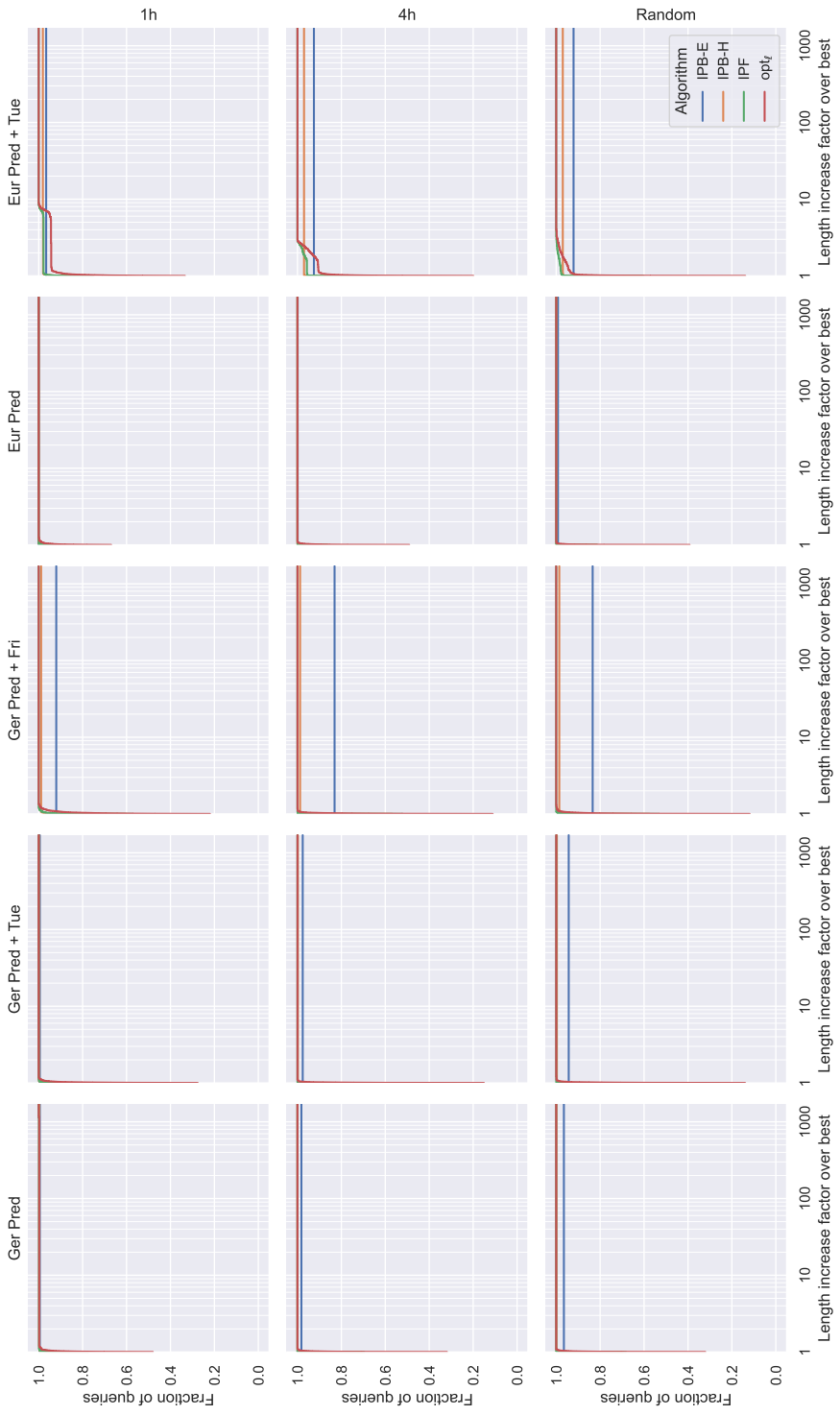


Figure E.4: Relative performance profile for solution quality of our algorithms for time-dependent and combined dynamic and time-dependent smooth paths on all queries from Table 10.2 split by graph and query set.

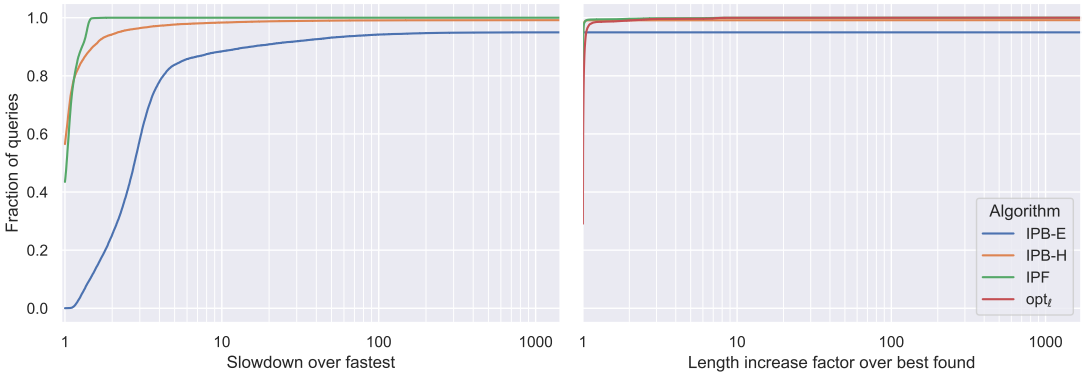


Figure E.5: Aggregated relative performance profiles of our algorithms for time-dependent smooth paths on all queries from Table 10.2.

F Temporary Road Closures: Visualization of Query Sets A1 and A2

The source and destination vertices for the query sets A1 and A2 are drawn from the regions A and B as shown in Figure F.1. Region A is the area southeast of 49°N 4°E and northwest of 47°N 18°E. Region B is the area southeast of 46°N 4°E and northwest of 42°N 18°E. From each region a vertex is drawn. Queries are generated in both directions. This setup is taken from [Brä18].

Figure F.1: Source and destination areas of query sets A1 and A2.

Deutsche Zusammenfassung

Die Nutzung mobiler Navigationsanwendung hat über die vergangen Jahre stark zugenommen. Wurden früher Reisen noch mit Straßenatlas geplant und über Verkehrsfunk gemeldete Staus ad-hoc mehr oder weniger erfolgreich umfahren, so lässt man sich heute von einer passenden Navigationsapplikation im Handumdrehen eine passende Route vorschlagen. Dank Integration aktueller Verkehrsdaten bleibt einem sogar die Entscheidung erspart, ob es sich wirklich lohnt den nächsten Stau zu umfahren. In Kombination mit passenden Navigations-/Fahranweisungen ergibt sich ein erheblicher Komfortgewinn.

Die Integration von Verkehrsdaten ist essenziell um "gute" Routen zu berechnen [DGPW17] und sicherlich auch ein wichtiger Grund, warum viele Nutzer mittlerweile eine Navigationsanwendung benutzen, selbst wenn sie die Strecke eigentlich kennen. Verkehrsdaten existieren dabei in zwei Varianten: Das sind zum einen Daten zur aktuellen Verkehrssituation. Diese sind dynamisch und hängen vom aktuellen Zeitpunkt ab. Zum anderen gibt es aber auch Verkehrsvorhersagen zu regelmäßigen Verkehrsströmen beispielsweise durch Pendler. Um diese in das Routing zu integrieren, wird die erwartete Fahrtzeit auf einem Straßensegment als abhängig von der Tageszeit angenommen, zu der es befahren wird. Solche zeitabhängigen Daten ändern sich deutlich seltener, sind also in der Regel nicht dynamisch. Dynamische Daten hingegen haben auch eine zeitabhängige Komponente. Aktuelle Staus können sich in der Zukunft wieder auflösen. Eine Umfahrung für einen 400 Kilometer entfernten Stau einzuplanen, ist nicht unbedingt sinnvoll.

Die (Weiter-)Entwicklung von praktikablen Algorithmen zur Routenplanung unter Einbeziehung solcher dynamischer und zeitabhängiger Daten ist Gegenstand dieser Dissertation. Dabei folgen wir der Algorithm-Engineering-Methodik [San09, MS10]. Diese hat sich im Bereich von Routenplanungsproblemen bereits in der Vergangenheit als extrem produktiv er-

wiesen [DSSW09]. Eine Reihe von *Beschleunigungstechniken* wurde entwickelt [Bas+16]. Viele davon werden in der Praxis eingesetzt [DGPW17, CP12]. Contraction Hierarchies (CH) [GSSV12] ist eine solche Technik, die eine gewisse Popularität erreicht hat. Mit CH können Kürzeste-Wege-Anfragen auf Straßennetzwerken mit Millionen von Knoten und Kanten nach einigen Minuten Vorberechnungszeit innerhalb von weniger als einer Millisekunde optimal beantwortet werden. Das ist ca. vier Größenordnungen schneller als mit dem Algorithmus von Dijkstra. Die in dieser Arbeit vorgestellten Algorithmen bauen auf CH auf. CH wurde bereits in der Vergangenheit für dynamische [Bas+16] und zeitabhängige [BGSV13] Problemvarianten erweitert. Diese Erweiterungen bringen aber ihre eigenen Probleme mit sich. Beispielsweise benötigt die zeitabhängige CH-Variante auf aktuellen Netzwerken mit detaillierten Verkehrsvorhersagen hunderte von Gigabytes an Hauptspeicher, was nicht praktikabel ist. Im Verlauf der Arbeit entwickeln wir daher vielfältige Verbesserungen und Erweiterungen.

Resultate. Die Arbeit beinhaltet die folgenden Hauptresultate: Erstens präsentieren wir CH-Potentiale, ein A*-basiertes Routing-Framework. Dieses Framework kann für alle Problemvarianten verwendet werden, für die sinnvolle untere Schranken der Kantengewichte zur Vorberechnungszeit zur Verfügung stehen. Mittels einer CH-Anfragevariante kann unser Algorithmus bezüglich der unteren Schranke *perfekte* Distanzabschätzungen berechnen und als A*-Potential verwenden. Die so erzielte Beschleunigung liegt zwischen einer und drei Größenordnungen gegenüber dem Algorithmus von Dijkstra, abhängig davon, wie akkurat die unteren Schranken sind. Zweitens stellen wir einige Verbesserungen und Erweiterungen von Customizable Contraction Hierarchies (CCH) [DSW16], der CH-Variante für dynamisches Routing, vor. Unsere Verbesserungen erzielen Laufzeitbeschleunigungen um bis zu einer Größenordnung. Mit unseren Erweiterungen unterstützt CCH praktisch wichtige erweiterte Problemszenarien wie Abbiegekosten, Alternativroutenberechnung sowie sogenannte Point-of-Interest-Anfragen. Drittens stellen wir die erste Beschleunigungstechnik für zeitabhängige Routenplanung vor, die speichereffizient ist und Kürzeste-Wege-Anfragen schnell und exakt beantworten kann. Im Vergleich zur bisherigen Realisierung von CH im zeitabhängigen Szenario hat unsere Technik einen um bis zu 40-fach reduzierten Speicherverbrauch, benötigt maximal ein Drittel der Vorberechnungszeit und beantwortet Anfragen nur unwesentlich langsamer. Viertens präsentieren wir eine Generalisierung von A* mit zeitabhängigen Potentialen. Darauf aufbauend entwerfen wir einen Ansatz für Routing mit einer Kombination aus Verkehrsvorhersage und aktueller Verkehrssituation. Updates zur Verkehrssituation können innerhalb eines Bruchteils einer Minute eingepflegt werden; die Laufzeiten zur optimalen Beantwortung von Routinganfragen sind interaktiv. Fünftens untersuchen wir erweiterte Problemstellungen zu Routing mit unvollständigen und verrauschten Verkehrsdaten sowie Routing für LKW unter Einbeziehung zeitabhängiger Sperrungen und Fahrverbote. Für beide Varianten stellen wir effiziente Algorithmen vor. Sechstens beinhaltet die Arbeit einige Komplexitätsresultate für zeitabhängiges Routing ohne die FIFO-Eigenschaft und für die erweiterten Problemstellungen.

Schlussfolgerungen. Ein Ansatz, der sich im Verlauf der Arbeit als extrem flexibel und fruchtbar erwiesen hat, ist das CH-Potentiale-Framework. Der Grund dafür ist, dass wir für CH-Potentiale Flexibilität über schnellstmögliche Anfragelaufzeit priorisiert haben. Wir beobachten, dass es eine Vielzahl an Forschungsresultate über Algorithmen gibt, die zuallererst auf schnellstmögliche Anfragelaufzeiten fokussiert sind [Bas+16]. Flexible und erweiterbare Routingalgorithmen wurden jedoch bisher deutlich weniger erforscht. Dabei ist diese Dimension für die praktische Anwendbarkeit von Routingalgorithmen teilweise sogar wichtiger als die Anfragelaufzeiten [DGPW17]. Die Entwicklung von solch flexiblen Algorithmen ist jedoch eine Herausforderung. Flexibilität ist keine einfach zu evaluierende, quantifizierbare Dimension. Oft lassen sich wirklich einfachen Ansätze erst nach Umwegen über komplizierte Herangehensweisen finden. Nichtsdestoweniger zeigt unsere Arbeit, dass auch nach 20 Jahren aktiver Routenplanungsforschung noch einfache Techniken zu entdecken sind. Die einzige algorithmische Neuerung bei CH-Potentialen ist die Anwendung einer Tiefensuche auf CH. Den Fokus weniger stark auf Anfragelaufzeiten zu legen und dafür mehr auf einfache und erweiterbare Algorithmen abzuzielen, halten wir daher für einen vielversprechenden Ansatz für zukünftige Arbeiten im Gebiet Routenplanung. Die inkrementelle Herangehensweise des Algorithm Engineering ist dabei essenziell, um praxistaugliche Algorithmen für zukünftige Mobilitätsanwendungen zu entwickeln.