Algorithm Engineering for Adaptive Route Planning

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Ben Armand Léon Strasser

aus Luxemburg

Tag der mündlichen Prüfung:	19. Juli 2017
Erster Gutachter:	Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Prof. Dr. Matthias Müller-Hannemann

Deutsche Zusammenfassung

Routenplanung beschäftigt sich mit der Frage, wie man am besten durch ein Transportnetz navigiert – ein für die meisten Menschen quasi alltägliches Problem. Die Anwendungsszenarien für automatisierte Methoden in diesem Feld sind zahlreich: PKW-Navigationsgeräte, LKW-Tourenplanung, Zugfahrplanauskunft, Smartphone-Navigation und viele mehr. Der Schwerpunkt meiner Forschung liegt in der Entwickelung effizienter Verfahren zur Wegfindung in sich verändernden Umgebungen. Dabei wird die Methodik “Algorithm Engineering” eingesetzt. Neben dem Algorithmenentwurf an sich und der theoretischen Analyse der vorgeschlagenen Algorithmen, werden alle Verfahren auch implementiert und experimentell, auf für die Anwendungen relevanten Daten, evaluiert. Die dabei gewonnenen Erkenntnisse fließen in den Entwurf neuer Algorithmen ein bzw. werden für die Anpassung bestehender Algorithmen verwendet.

Die präsentierten Algorithmen haben alle zum Ziel, die Wegesuche in Straßennetzen oder in fahrplanbasierten Netzen zu beschleunigen. Fahrplanbasierte Netze beinhalten unter anderem Züge, Busse und Flugzeuge. Dabei tauchen auch Subprobleme auf, denen man die Relevanz in diesem Kontext nicht direkt ansieht. Beispielsweise bauen viele Algorithmen auf einer Zerlegung des Netzes auf. Dies führt zum Problem der balancierten Graph-Bipartitionierung welche auch in dieser Arbeit behandelt wird.

Bei der adaptiven Routenplanung werden Veränderungen in Verkehrsnetzen mit in Betracht gezogen. Diese können vielfältiger Art sein. Hier wird der Fokus auf Staus und Verspätungen gelegt. Die klassischen Algorithmen aus diesem Gebiet, wie zum Beispiel Dijkstras Algorithmus, sind sehr flexibel bezüglich variierender Netze. Allerdings sind sie auch vergleichsweise langsam. Dijkstras Algorithmus braucht, selbst auf moderner Hardware, auf einem etablierten Westeuropa-Testgraph im Durchschnitt rund zwei Sekunden pro Anfrage. Für viele Anwendungen, wie beispielsweise einem Webserver mit tausenden von Anfragen pro Sekunde, ist dies zu langsam. Über die Zeit wurden deswegen viele Algorithmen entwickelt, die in zwei Phasen arbeiten. In der ersten, langsameren Vorberechnungsphase, wird das Netz vorverarbeitet und in der zweiten, schnelleren Anfragephase werden die Wege berechnet. Wie vielfach gezeigt wurde [7], lassen sich damit Pfadanfragen auf realitätsnahen Straßengraphen in weit unter einer Millisekunde beantworten. Dieser Ansatz hat aber ein massives Problem: Ändert sich das Netz, so muss die Vorberechnung erneut durchgeführt werden. Es besteht ein Bedarf an Verfahren die sowohl schnelle Pfadanfragen zulassen, als auch flexibel genug sind um sich schnell an verändernde Umstände in Verkehrsnetzen anpassen zu können.

In dieser Arbeit werden mehere Verfahren entwickelt. Das “Customizable Contraction Hierarchy" (CCH) Verfahren wird entwickelt um schnell Routen in Straßennetzen berechnen zu können wenn unvorhergesehene Staus in Betracht gezogen werden. Dieses Verfahren setzt voraus, dass sich der Eingabegraph entlang kleiner Graphschnitte zerlegen lässt. Zu diesem Zweck wird das "FlowCutter" Verfahren entwickelt. Um vorhersagbare Staus wie Pendlerströme einbeziehen zu können, wird das "TD-S" Verfahren vorgeschlagen. Um in schienengebundenen Netzen flexibel Routen finden zu können wird der "Connection Scan Algorithm" (CSA) weiterentwickelt. Aufbauend auf CSA werden adaptive Algorithm für diverse Varianten des Fahrplanauskunftsproblems entwickelt. CSA wird mit einem Multilevelansatz kombiniert und zu "CSAccel" weiterentwickelt.

Contents

Acknowledgments

Foremost, I want to thank my advisor Dorothea Wagner for her support and advice given during my doctoral studies. I am grateful for being given the opportunity to join her research group. I want to thank all my colleagues for the friendly, cooperative, and productive environment. I am especially thankful to Julian Dibbelt and my room mate Michael Hamann for many fruitful and interesting discussions. Further, I'd like to thank Matthias Müller-Hannemann for reviewing my dissertation.

Moreover, I wish to thank all my co-authors Adi Botea, Ulrik Brandes, Lars Briem, Sebastian Buck, Julian Dibbelt, Holger Ebhart, Nicolai Mallig, Michael Hamann, Daniel Harabor, Bastian Katz, Nathan Sturtevant, Thomas Pajor, Ignaz Rutter, Peter Vortisch, Dorothea Wagner, and Tobias Zündorf for many fruitful and inspiring discussions. As sign of respect, all chapters of this thesis are written using "we" instead of "I".

Finally, I am grateful to my fiancée Stefanie Weißenbach for supporting me in my endeavors and proofreading this thesis. I would not be here without her.

1 Introduction

1.1 Adaptive Route Planning

Navigation devices and services are used by millions of people on a daily basis to find routes through various types of transportation networks. Such services must quickly answer routing queries. These queries consist of finding a route from a user given source location to a user given target location. Route planning algorithms are the foundation of these services. The topic of this thesis is to study and improve existing and develop new route planning algorithms.

The problem setting specifics vary depending on type of the considered transportation vehicles. Finding routes in networks with vehicles that depart when a traveler wants differs from a setting where vehicles have fixed schedules. A prominent example for the first type is finding a fastest route for a car through a road network. An example for the second type consists of finding routes containing trains and buses. We refer to first type as *road-based network* and to the second type as *timetable-based network*. In Part I of this thesis, we study the first type. The second type is studied in Part II.

There exists a lot of research into this subject. We refer to [7] for a survey and overview of the existing research. In each chapter, we present a more detailed overview over the related work specific to the chapter. A common problem consists in dealing with the size of large transportation networks. Queries should ideally be answered within milliseconds. However, it is often not possible to search the whole network within this timeframe. Many proposed algorithms therefore work in two phases [122]. In a slow offline phase, called *preprocessing phase*, auxiliary data is computed that only depends on static information that does not change between routing queries. An example of such static information is the road graph. On the other hand, the source and the target locations vary between queries and are therefore unknown during the preprocessing phase. The various proposed algorithms significantly vary with respect to the auxiliary data they compute. An example of auxiliary data is which nodes are dead ends.

Routes are computed in a fast second phase, called *query phase*. The query phase may use the auxiliary data computed during the preprocessing phase. If the auxiliary data contains for example the information which nodes are dead ends, then the query phase could use this information to avoid scanning most dead-ends. Algorithms that work with such a preprocessing and query phase setup are commonly called *speedup techniques*.

In this thesis, we focus on *adaptive* route planning. This means that the transportation network is allowed to change over time. Ideally, route planning algorithms should take these changes into account.

We distinguish between two types of changes: *Realtime* and *predicted* changes. Realtime changes are not known in advance but the algorithms must be able to adapt quickly to the new situation. An example for a realtime change is a road closure because of an accident. Realtime changes also occur in timetable-based networks. An obvious example is a delayed train. However, different reasons for realtime changes exist. For example, some train operators do not allow their trains to be overbooked. This means that once all tickets are sold, the routing engine must find new journeys with other trains for future customers. Supporting realtime changes is a complex extension of the existing setup because changes to the transportation network can induce changes to the auxiliary data. However, rerunning the full preprocessing step is usually too slow. Adapting algorithms to account for realtime changes tends to be easier the simpler the structure of the auxiliary data is. This observation plays a very important role in the second part of this thesis, where we present an algorithm whose preprocessing phase essentially consists of sorting the input data.

Predicted changes are used to model recurring predictable events. In road-based networks the prime example of a predicted change is a daily rush hour. In many regions, one can predict with near certainty that in the morning of a weekday certain roads will be congested. This effect is due to numerous people simultaneously commuting to work. Similarly, in the evening, the roads are congested in the opposite direction. In timetable-based networks some trains have a higher likelihood to be delayed than others. For example, a train that needs to wait for connecting trains is more likely to be delayed than a train that does not need to wait. We can therefore predict that some trains are more likely to be delayed than others. Supporting predicted changes does not require a modifiable auxiliary data structure as needed with realtime changes. The prediction is already known during the preprocessing phase and can therefore be directly integrated into the auxiliary data. However, integrating these predictions into existing algorithms is far from trivial.

Network changes can be predicted either through a simulation or by aggregating past data. For example, in-car navigation device providers can record the positions of their customers using GPS. From these GPS traces, typical speed profiles can be derived that formalize the concept of a rush hour. We refer to these speed profiles as *prediction*. Computing these predictions is beyond the scope of this thesis. Within this thesis we assume that the predictions are given in the input to the preprocessing phase.

We can consider combinations of the problem settings. We obtain eight different problem settings depending on whether we consider road- or timetable-based networks, whether realtime changes are considered, or whether predicted changes are considered. In this thesis, we present contributions to all eight combinations.

1.1.1 Eight Routing Problem Settings

Road-based route planning without realtime nor predicted changes is the classical problem setting studied by most papers in this domain. A plethora of algorithms has been proposed to solve this problem setting. We refer to [7] for an overview. Algorithms exist that are known to work very well in practice. Proposing yet another algorithm therefore does not seem useful. However, while many algorithms have been demonstrated to be very effective in practice, theoretical insights into their inner working are often lacking or are incomplete. Our research into Contraction Hierarchies (CH) [76, 77] provides some theoretical insights into the graph structures exploited by the algorithm. We are able to show that the Contraction Hierarchy algorithm is efficient if the road graph has small, recursive, balanced node separators. An equivalent formulation is that if the road graph has a sufficiently small tree width, a CH is efficient. Further, we are able to show that realworld road graphs have this structure.

Significantly fewer algorithms exist that address the road-based route planning problem with realtime changes. A prominent example is based on a multilevel overlay extension of Dijkstra's algorithm [88]. The corresponding problem setting is nowadays often referred to as *Customizable Route Planning* (CRP) [45]. If one is able to quickly adapt to realtime changes in the transportation network, then one is also capable of quickly adjusting to the individual needs of every passenger. This the reason why the problem setting is called "customizable". Beside the preprocessing and query phases, a third phase called *customization phase* is introduced that incorporates edge weight changes into the auxiliary data. In Chapter 2, we extend Contraction Hierarchies to handle realtime changes and refer to the resulting algorithm as Customizable Contraction Hierarchies (CCH). Further, we are able to prove that our proposed algorithm is efficient for all possible edge weights as long as the road graph has small, recursive, balanced node separators.

Several algorithms exist that handle the road-based routing problem with only predicted changes. One usually refers to the problem as *time-dependent* (TD) route planning. While a lot of research has been put into this topic [53, 56, 40, 107, 50, 13, 72, 18, 95], we demonstrate that existing algorithms have problems such as being too slow or requiring too much memory on large instances. Further, most algorithms are very complex, which can prevent the integration of the algorithm into software products. We therefore propose a comparatively simple, sampling-based, heuristic approach called TD-S in Chapter 5. It uses the Contraction Hierarchy algorithm as subroutine. In an experimental study, we demonstrate using a large, current Europe instance that all competitors either have slow query running times or use a prohibitive amount of space. Unfortunately, while our approach is able to solve the problem on realistic inputs "good enough", it is highly heuristic. Contrary to the road-based route planning problem with only realtime changes, we cannot provide any deep theoretical insights into why the algorithm works. Further, the algorithm inherently depends on the structure of realistic inputs. When confronted with worst-case inputs, the quality of the computed routes deteriorates rapidly.

Routing with predicted changes inherently depends on the departure time of the passenger. As result, the input or output of the query phase must be modified compared to the basic problem setting. One modification consists of adding a departure time to the input besides the source and target destinations. The resulting problem setting is called *earliest arrival* problem. Alternatively, the input only contains the source and target destinations and the output should contain the earliest arrival time for every departure time. The output is thus a function. This problem variant is called *profile* problem. Efficiently solving the profile problem is usually harder than solving the earliest arrival problem.

A road-based routing problem with realtime and predicted changes is sometimes also called *dynamic* routing problem. Solving the road-based routing problem with only predicted changes is already difficult. Combining it with realtime changes does not make the problem easier. It is therefore usually difficult to extend algorithms solving the problem with only predicted changes but papers into this direction exist [50, 18]. Fortunately, as TD-S is comparatively simple, we can extend it heuristically to the combined setting quite easily. We refer to the extended algorithm as TD-S+D and describe it in Section 5.7.3. TD-S+D uses the Customizable Contraction Hierarchy algorithm as subroutine.

There is a lot of research into timetable-based route planning without realtime nor predicted changes. A very influential work is [122] which introduced the idea of accelerating queries in transportation networks using overlays. Many algorithms work by modeling the timetable either as time-expanded or time-dependent graph. We refer to [112] for a detailed description of these two graph types. Unfortunately, adaptations of many speedup techniques for road-based networks work significantly worse on timetable-based networks [19, 21]. In Chapter 6, we describe the Connection Scan algorithm (CSA). It is a very simple algorithm, that relies on nearly no preprocessing. The CSA preprocessing phase essentially only needs to sort the scheduled vehicles in the timetable by departure time. CSA scales astonishingly well because of its very simple, processor friendly code structure. However, running times of complex queries on large country-wide networks with a timetable that contains both trains and buses are higher than desirable. We therefore present in Chapter 8 a multilevel overlay extension of CSA named CSAccel. It follows the three phase setup of CRP [45]. Unfortunately, the running time of the customization phase of CSAccel requires several minutes of running time instead of the usual seconds needed by algorithms for road-based networks.

The timetable-based route planning problem with only realtime changes has been studied in the past [21, 36, 9]. However, the proposed techniques severely limit the type of changes that are allowed. For example, a common assumption is that vehicles visit the stops in the planned order but may arrive later than scheduled. With this model it is not possible to support vehicles that have a modified stop sequence or arrive earlier. A modified stop sequence appears for example, if a train station unexpectedly closes down. Both CSA and CSAccel support realtime changes. CSA has no restriction in how the timetable changes. It is possible with CSA to change the arrival times and stop sequences of every vehicle. CSAccel is only efficient if the number of vehicles that pass over a track remains roughly comparable, i.e., it is slightly less flexible than CSA. The advantage of CSAccel is that it achieves lower query running times on country-wide networks.

Timetable-based route planning inherently depends on the departure time of the passenger. The distinction between earliest arrival and profile problem variants is therefore meaningful even without predicted changes. It is usual to consider additional optimization criteria beside arrival time such as for example the number of transfers. There are multiple ways to combine these two criteria. A simple option is to compute a route with a minimum number of transfers among all routes with a minimum arrival time. A more sophisticated approach consists of optimizing both criteria in the Pareto-sense [103]. In this setting, the output of an algorithm then consists of a set R of routes such that no route in R is better with respect to both criteria than any other route of R.

Predicted changes have an inherent uncertainty. For example in the road-based setting, a predicted congestion can be less or more severe than predicted. This is similar in the timetable-based setting. Trains can be delayed and thus do not perfectly adhere to their schedule. A delay prediction contains the information about which train will be delayed with what probability. Similarly to the road-based setting, such a probability can be estimated by aggregating historic information. For example, a train that was delayed four days in the past week is more likely to be delayed today than a train that was always on time in the last week. We formalize this problem as Minimum Expected Arrival Time (MEAT) problem. We describe an algorithm based on CSA to solve the MEAT problem. Our proposed algorithm does not require additional preprocessing compared to CSA. CSA can easily incorporate realtime changes to the timetable. Our MEAT solver directly inherits this property and thus addresses the timetable-based routing problem with predicted and realtime changes.

1.2 Graph Bisection

A significant amount of speedup techniques partition or bisect road graphs. They exploit that small, balanced, recursive edge cuts or node separators exist. The Customizable Contraction Hierarchy algorithm that we develop belongs to this category. Unfortunately, the existence of such cuts or separators is usually not enough. We additionally require a method to detect them. Indeed, the most costly step of the CCH preprocessing phase consists of finding small, balanced node separators. Unfortunately, balanced graph bisection is an NP-hard problem [74]. Finding a solution that works well enough for our application is therefore a challenge of its own. We start our research using general purpose graph partitioning tools [92, 48, 117, 5, 120]. Our first experimental evaluation of CCH in Chapter 2 uses these.

A conclusion of this experiments is that finding better separators would immediately improve the performance of CCH with respect to all criteria. Further, the existing general purpose graph partitioning software seems to overly emphasize high balance, which is not necessarily helpful for our application. We therefore start researching graph partitioning in the context of road graphs. The result is FlowCutter and described in Chapter 3 together with a second experimental CCH evaluation demonstrating that the achieved bisections are, at least with respect to our application, superior. FlowCutter computes not a single cut but a set of cuts that heuristically optimize balance and cut size in the Pareto-sense. This allows us to effectively find cuts that optimize a combination of balance and cut size. This is not possible with existing software as these always require a balance as part of the input and use it as a side constraint.

We also apply FlowCutter to graph partitioning benchmark sets containing graphs from other applications. FlowCutter shows good performance across the board as long as small balanced separators exist. The larger the separators get the more running time FlowCutter requires and the less likely FlowCutter is to find a small cut. FlowCutter is therefore a good heuristic to find small cuts or to demonstrate their absence.

Chapter 4 describes a relation between CCH, multilevel graph partitioning, and tree decompositions. Tree decompositions are related to many areas, such as for example efficient Gaussian elimination on sparse matrices. This establishes a astonishingly deep relation between seemingly unrelated topics such as efficient route planning in roads and efficient Gaussian elimination of sparse matrices. Using this connection, we can use FlowCutter to efficiently compute tree decompositions. We therefore entered FlowCutter into the "PACE 2016 Track A" implementation challenge [39], whose objective was to compute tree-decompositions of small width. FlowCutter won the first place in the sequential setting and was second by a small margin in the parallel setting. The results of this competition clearly show that, even though FlowCutter was designed with CCH and road graphs in mind, it can by applied in a significantly broader context.

1.3 Algorithm Engineering

Algorithm Engineering is a methodology used throughout this thesis to design algorithms. For an in-depth discussion of the approach, we refer to [114] and [118].

The key realization that lead to the development of the methodology is that there is an interdependency between theoretical algorithm design and practical algorithm evaluation. This contrasts with many algorithmic studies that consider the development of an algorithm as finished once a non-trivial asymptotic worst case running time is known. The core idea of Algorithm Engineering is to go beyond asymptotic

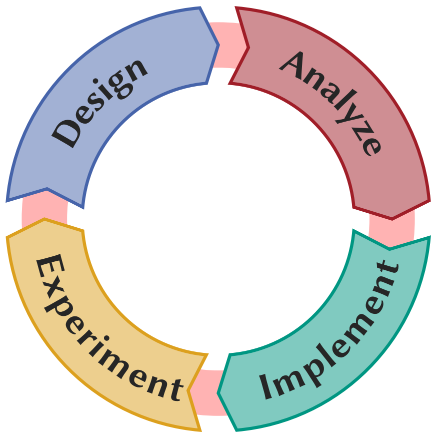


Figure 1.1: Algorithm Engineering Cycle.

running time analysis and to implement the proposed algorithms. These should then be experimentally evaluated on real world instances within an application context. Insights gained through these experiments should be used to improved on the design, analysis, and understanding of the algorithm. The improved algorithm should then be experimentally reevaluated. This approach gives rise to a repeating work flow that is usually depicted using cycle as in Figure 1.1.

The methodology is very visible in Chapters 2, 3, and 4. We started our research just after a purely theoretical study by [15]. This study suggested a link between small, nested separators and Contraction Hierarchies. We use this insight to improve the CH algorithm design yielding the CCH algorithm and perform an experimental evaluation in Chapter 2. This chapter therefore starts in the middle of the algorithm design stage and ends after the experiment phase. Using the insights gained in the evaluation, we revisit the graph bisection subroutine of the CCH algorithm resulting in the development of the FlowCutter algorithm. We improve the existing design and reevaluate the modified CCH algorithm in Chapter 3. This chapter therefore starts in the context of the Algorithm Engineering cycle right where Chapter 2 left in the algorithm design phase and finishes with the experiments. From the experimental evaluation of the FlowCutter and CCH algorithms, we gain the insight that tree decompositions seem to be a very related concept. This observation lead to the relations described in Chapter 4, which therefore can be seen as the first step towards the design of an improved algorithm.

1.4 Contribution

In this thesis, we introduce several algorithms: CCH, FlowCutter, TD-S, CSA, and CSAccel. CCH is a solution to the road-based routing problem with realtime congestions. TD-S makes use of CCH to provide a heuristic approach to the road-based routing problem with predicted and optionally realtime congestions. We demonstrate that there is a deep connection between CCH and tree decompositions and analyze the performance of CCH in terms of tree decomposition terminology. We design the graph bisection algorithm FlowCutter to improve the CCH preprocessing step. As a side result, we obtain with FlowCutter an efficient and scalable tree decomposition computation heuristic. CSA is a simple but very flexible framework to solve a variety of timetable related routing problem settings. CSAccel is a combination of CSA with a multilevel partitioning scheme to decrease query running times.

1.5 Outline

This thesis is organized into two parts. In the first part, we discuss road-based networks. The subject of the three chapters 2, 3, and 4 is realtime congestion. In Chapter 2, we introduce Customizable Contraction Hierarchies (CCH), an algorithm to solve the road-based routing problem with only realtime congestions. Chapter 3 introduces FlowCutter, a graph bisection algorithm. FlowCutter can be used in the CCH preprocessing. In Chapter 4, we focus on the relation between CCH, tree decompositions, and multilevel partitions. We further derive worst case running time bounds for CCH in this chapter and compare these to bounds achieved using highway dimension theory. In Chapter 5, we present TD-S, a simple heuristic to handle predicted congestions in road-based networks.

We discuss timetable-based networks in the second part of this thesis. All algorithms that we describe are based on the Connection Scan Algorithm (CSA), which is introduced in Chapter 6. We expand CSA to compute profiles and Pareto optimization in Chapter 7. In Chapter 8, we accelerate CSA using a multilevel partitioning scheme. The resulting algorithm is called CSAccel. Finally, in Chapter 9 we introduce the Minimum Expected Arrival Time (MEAT) problem and describe a solution algorithm that is based on the CSA profile algorithm.

Part I

Routing in Road Networks

2 Customizable Contraction Hierarchies

Chapters 2, 3, and 4 are coupled. In these chapters, we develop and discuss Customizable Contraction Hierarchies (CCH), an algorithm to solve the road-based routing problem with realtime congestions. In Chapter 2, we introduce and evaluate the technique itself. Chapter 3 focuses on graph bisection, which is a subroutine needed in the preprocessing phase of CCH. We introduce a novel graph bisection algorithm named FlowCutter and demonstrate that using it we can achieve the best CCH performance. The data structures used by CCH are coupled with tree decomposition theory. This connection is explained in Chapter 4. In it, we also bound the worst-case CCH performance in terms of tree decomposition related quantities. Further, we demonstrate in this chapter that multilevel partitions and tree decompositions are related.

Tree decomposition theory can be a complex topic. Luckily, all CCH correctness proofs can be performed using more elementary arguments than those involving tree decompositions. In Chapters 2 and 3, we therefore mostly ignore tree decompositions.

The experiments of Chapter 2 were performed before the experiments of Chapter 3 and therefore do not reference FlowCutter. We repeat the most important CCH experiments using FlowCutter in the evaluation of Chapter 3. It is the discussion of Table 3.7 in Section 3.6.2 that ties the two algorithms together.

We first described Customizable Contraction Hierarchies in an ArXiv paper [59]. A short version of this paper was presented at the SEA conference [60]. A more in-depth study was later published in the JEA journal [61]. This chapter is based on the JEA text. Customizable Contraction Hierarchies are also the subject of a book chapter [134]. This chapter is based upon joint work with Julian Dibbelt and Dorothea Wagner. We provide an open source CCH implementation in RoutingKit [129].

2.1 Introduction

In this chapter, we introduce the Customizable Contraction Hierarchies (CCH) algorithm to solve the road-based routing problem with realtime congestions. It works by preprocessing the input graph but keeping the edge weights flexible. The edge weights can be exchanged in a quick customization phase. In total there are three phases: the preprocessing phase, the customization phase, and the query phase.

The *preprocessing phase* is computationally expensive. During the preprocessing phase, only the road graph is known. The edge weights and the source and target nodes are unknown. The preprocessing phase must only be executed if the road graph changes, i.e., new roads are built. The assumption is that this happens sufficiently rarely that investing several hours of computation time is acceptable.

The *customization phase* should be reasonably fast. This phase is handed the results of the preprocessing phase and the edge weights. The source and target nodes are still unknown. The objective is to integrate the edge weights into the preprocessing phase. It must be run each time that the congestion situation changes. We envision a setup, where new congestion data is fed to the system at regular intervals such as for example every 10 seconds. The customization phase must therefore be run every 10 seconds. Its running time should thus be significantly smaller. A running time of 1 second or less seems ideal.

The *query phase* must be very fast. It is run each time that a path is requested from the system. It knows the results of the preprocessing phase and of the customization phase. Further, the source and target nodes are handed to the query phase as input. The output consists of a shortest path. We envision a system where there are many queries per second. For throughput reasons it is therefore important that the running time of a single query is fast. There are further reasons why a low query running time is desirable. For example, reactivity is important. If computing a path requires several seconds, as is the case with Dijkstra's algorithm and long distance queries, then the system will feel laggy to the user.

In our envisioned setup, we run the preprocessing phase very rarely. For many applications, it is enough to update the road graph data every day or even every week. The customization phase should be regularly executed with the current realtime information. We expect that every 10 seconds is a useful setup. Finally, we expect there to be a large amount of path queries. It is therefore of uttermost importance that the query phase is fast.

Many speedup techniques for road-based networks work by adding extra edges called *shortcuts* to the graph that allow query algorithms to bypass large regions of the graph efficiently. While variants of the optimal shortcut selection problem have been proven to be NP-hard [16], determining good shortcuts is feasible in practice even on large road graphs. Among the most successful speedup techniques using this building block are Contraction Hierarchies (CH) by [76, 77]. At its core the technique consists of a systematic way of adding shortcuts by iteratively contracting vertices along a given order. Even though ordering heuristics exist that work well in practice [77], the problem of computing an optimal ordering is NP-hard in general [14]. A central restriction of CHs as proposed by [77] is that their preprocessing is *metric-dependent*, that is edge weights, also called *metric*, need to be known. Substantial changes to the metric, e.g., due to traffic congestion, may require expensive recomputations. CH is therefore an excellent solution to the routing problem in road-based networks without realtime nor predicted congestions. We extend and improve upon CH to allow it to handle realtime congestions.

Game Grid Scenario. Most existing CH papers focus solely on road graphs [7], with [125] being a notable exception, but there are many other applications with differently structured graphs in which fast shortest path computations are important. It is therefore unclear whether good performance on road graphs translates into a good performance on graphs from a different application. To diversify our experimental evaluation, we do not just evaluate the performance on road graphs. We further investigate an application that originates from path finding in computer games. Our main objective is to demonstrate how CCH behaves on data that it was not designed for. We do not intend to engineer an algorithm to handle this particular application.

Consider a real-time strategy game where units quickly have to navigate across a large map with many choke points. The basic topology of the map is fixed, however, when buildings are constructed or destroyed, fields are rendered impassable or freed up. Furthermore, every player has his own knowledge of the map. Many games include a feature called *fog of war*: Initially only the fields around the player's starting location are revealed. As his units explore the map, new fields are revealed. Since a unit must not route around a building that the player has not yet seen, every player needs his own metric. Furthermore, units such as hovercrafts may traverse water and land, while other units are bound to land. This results in vastly different, evolving metrics for different unit types per player, making metric-dependent preprocessing difficult to apply. Contrary to road graphs one-way streets tend to be extremely rare, and thus being able to exploit the symmetry of the underlying graph is a useful feature.

Metric-Independent Orders for CHs. One of the central building blocks of this chapter is the use of metric-independent *nested dissection orders* (ND-orders) [78] for CH precomputation instead of the metric-dependent order of [77]. This approach was proposed by [15], and a preliminary case study can be found in [146]. A similar idea was followed by [54], where the authors employ partial CHs to engineer subroutines of their CRP [45] customization phase. They also refer to preliminary experiments on full CH but did not publish results. Similar ideas have also appeared in [109]: They consider graphs of low *tree width* and leverage this property to compute CH-like structures, without explicitly using the term CH. Related techniques by [142, 35] work directly on the tree decomposition. Interestingly, our experiments show that even large road networks have relatively low tree width: Real-world road networks with vertex counts in the 10^7 have tree width in the 10^2.

Directed and Undirected Graphs. Real-world road networks contain one-way streets and highways. Such networks are usually modeled as directed graphs. Our algorithms fully support direction of traffic—however, we introduce it at a different stage of the toolchain than most related techniques, which should not be confused with only supporting undirected networks. Our first preprocessing phase works exclusively on the underlying undirected and unweighted graph, obtained by dropping all edge directions and edge weights. Direction of traffic as well as traversal weights are only introduced in the second customization phase, where every edge can have two weights: an upward and a downward weight. If an edge corresponds to a one-way street, then one of these weights is set to ∞. This setup is a strength of our algorithm: Throughout large parts of the toolchain we are not confronted with additional algorithmic complexity induced by directed edges.

Our Contribution. The main contribution of our work in this chapter is to show that Customizable Contraction Hierarchies (CCH) solely based on the ND-principle are feasible and practical. Compared to CRP [44], we achieve a similar preprocessing–query trade-off, albeit with slightly better query performance at slightly slower customization speed and we need somewhat more space. For less well-behaved metrics such as travel distance, we achieve query times below the original metric-dependent CH of [77]. Besides this main result, we show that given a fixed contraction order, a metric-independent CH can be constructed in time essentially linear in the size of the Contraction Hierarchy with working memory consumption linear in the input graph. Our specialized algorithm has a better theoretic worst-case running time and performs significantly better empirically than the dynamic adjacency arrays used in [77]. Another contribution of our work are perfect witness searches. We show that for a fixed metric-independent vertex order it is possible to construct CHs with a provably minimum number of arcs in a few seconds on continental road graphs. Our construction algorithm has a running time performance completely independent of the weights used. For a class of graphs with very regular recursive vertex separators and metric-independent CHs, we can show the following: There exists a nested dissection order which achieves a constant factor approximation of the maximum and average search space sizes in terms of the number of arcs and vertices. Experimentally, we show that road graphs have such a recursive separator structure.

Outline. Section 2.2 sets necessary notation. Section 2.3 discusses metric-dependent orders as used by [77], highlighting specifics of our implementation. Next, we discuss metric-independent orders in Section 2.4. In Section 2.5, we describe how to efficiently construct the arcs of the CH. The next Section 2.6 discusses how to efficiently enumerate triangles in the CH — an operation needed throughout the customization process detailed in Section 2.7. We further describe the details of the perfect witness search in Section 2.7. Finally, Section 2.8 concludes the algorithm description by introducing the algorithms used in the query phase to compute shortest path distances and compute the corresponding paths in the input graph. We then present an extensive experimental study that thoroughly evaluates the proposed algorithm.

2.2 Basics

We denote by $G = (V, E)$ an *undirected n-vertex graph* where V is the set of *vertices* and E the set of *edges*. Furthermore, $G = (V, A)$ denotes a *directed graph*, where A is the set of *arcs*. A graph is *simple* if it has no loops or multi-edges. Graphs in this chapter are simple unless noted otherwise, e.g., in parts of Section 2.5. Furthermore, we assume that input graphs are strongly connected. We denote by $N(v)$ the neighborhood of vertex $v \in G$, i.e., the set of vertices adjacent to v; for directed graphs the neighborhood ignores arc direction. A *vertex separator* is a vertex subset $S \subseteq V$ whose removal separates G into two disconnected subgraphs induced by the vertex sets A and B. The sets S, A and B are disjoint and their union forms V. The subgraphs induced by A and B are not necessarily connected and may be empty. A separator S is *balanced* if $\max\{|A|, |B|\} \leq 2n/3$.

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

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

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

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

The unweighted *vertex contraction* of v in G consists of removing v and all incident edges and inserting edges between all neighbors $N(v)$ if not already present. The inserted edges are referred to as *shortcuts* and the other edges are *original edges*. Given

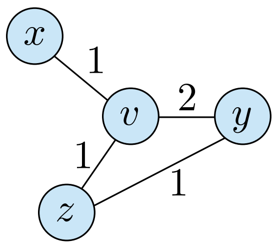


Figure 2.1: Contraction of v. If the pair x, y is considered first, a shortcut $\{x, y\}$ with weight 3 is inserted. If the pair x, z is considered first, an edge $\{x, z\}$ with weight 2 is inserted. This shortcut is part of a witness $x \to z \to y$ for the pair x, y. The shortcut $\{x, y\}$ is thus *not* added if the pair x, z is considered first.

an order π the *core graph* $G_{\pi,i}$ is obtained by contracting all vertices $\pi(1) \dots \pi(i-1)$ in order of their rank. We call the original graph G augmented by the set of shortcuts a *contraction hierarchy* $G^*_\pi = \bigcup_i G_{\pi,i}$. Furthermore, we denote by $G^\wedge_\pi$ the corresponding upward directed graph.

Given a fixed weight w, one can exploit that in many applications it is sufficient to only preserve all shortest path distances [77]. *Weighted vertex contraction* of a vertex v in the graph G is defined as the operation of removing v and inserting (a minimum number) of shortcuts among the neighbors of v to obtain a graph G' such that $\text{dist}_G(x, y) = \text{dist}_{G'}(x, y)$ for all vertices $x \neq v$ and $y \neq v$. To compute G', one iterates over all pairs of neighbors x, y of v increasing by $\text{dist}_G(x, y)$. For each pair one checks whether a xy-path of length $\text{dist}_G(x, y)$ exists in $G \backslash \{v\}$, i.e., one checks whether removing v destroys the xy-shortest path. This check is called *witness search* [77] and the xy-path is called *witness*, if it exists. If a witness is found, the considered vertex pair is skipped and no shortcut added. Otherwise, if an edge $\{x, y\}$ already exists, its weight is decreased to $\text{dist}_G(x, y)$, or a new shortcut edge with that weight is added to G. This new shortcut edge is considered in witness searches for subsequent neighbor pairs as part of G. If shortest paths are not unique, it is important to iterate over the pairs increasing by $\text{dist}_G(x, y)$, because otherwise more edges than strictly necessary can be inserted: Shorter shortcuts can make longer shortcuts superfluous. However, if we insert the shorter shortcut after the longer ones, the witness search will not consider them. Figure 2.1 shows an example. This effect was independently observed by [113] in a different setting. The witness searches are expensive and therefore the witness search is usually aborted after a certain number of steps [77]. If no witness was found, we assume that none exists and add a shortcut. This does not affect the correctness of the technique but might result in slightly more shortcuts than necessary. To distinguish, *perfect witness search* is without such a one-sided error.

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

The search space $\mathrm{SS}(v)$ of a vertex v is the subgraph of $G^\wedge_\pi$ (respectively $G^\wedge_{w,\pi}$) reachable from v. For every vertex pair s and t, it has been shown that a shortest up-down path must exist. This up-down path can be found by running a bidirectional search from s restricted to $\mathrm{SS}(s)$ and from t restricted to $\mathrm{SS}(t)$ [77]. A graph is *chordal* if for every cycle of at least four vertices there exists a pair of vertices that are non-adjacent in the cycle but are connected by an edge. An alternative characterization is that a vertex order π exists such that for every i the neighbors of $\pi(i)$ in $G_{\pi,i}$, i.e., the core graph before the contraction of $\pi(i)$, form a clique [73]. Such an order is called a *perfect elimination order*. Another way to formulate this characterization in CH terminology is as follows: A graph is chordal if and only if a contraction order exists such that the CH construction without witness search does not insert any shortcuts. A chordal super graph can be obtained by adding the CH shortcuts.

The elimination tree $T_{G,\pi}$ is a tree directed towards its root $\pi(n)$. The parent of vertex $\pi(i)$ is its upward neighbor $v \in N_u(\pi(i))$ of minimal rank $\pi^{-1}(v)$. This definition already yields a straightforward algorithm for constructing the elimination tree. As shown in [15], the set of vertices on the path from v to $\pi(n)$ is the set of vertices in $\mathrm{SS}(v)$. Computing a contraction hierarchy without witness search of graph G consists of computing a chordal super graph G^*_π with perfect elimination order π. The height of the elimination tree corresponds to the maximum number of vertices in the search space. The elimination tree is only defined for undirected unweighted graphs.

2.3 Metric-Dependent Orders

Most publications on applications and extensions of Contraction Hierarchies use greedy orders in the spirit of [77], but details of vertex order computation and witness search vary. For reproducibility, we describe our precise approach in this section, extending on the general description of metric-dependent CH preprocessing given in Section 2.2. Our witness search aborts once it finds some path shorter than the shortcut—or when both forward and backward search each have settled at most p vertices. For most experiments we choose $p = 50$. The only exception is the distance metric on road graphs, where we set $p = 1500$. We found that a higher value of p increases the time per witness-search but leads to sparser cores. For the distance metric we needed a high value because otherwise our cores get too dense. This effect did not occur for the other weights considered in the experiments. Our weighting heuristic is similar to the one of [2]. We denote by $L(x)$ a value that approximates the level of vertex x. Initially all $L(x)$ are 0. If x is contracted, then for every incident edge $\{x,y\}$ we perform $\ell(y) \leftarrow \max\{\ell(y), \ell(x)+1\}$. We further store for every arc

a a hop length $h(a)$. This is the number of arcs that the shortcut represents if fully unpacked. Denote by $D(x)$ the set of arcs removed if x is contracted and by $A(x)$ the set of arcs that are inserted. $A(x)$ is not necessarily a full clique because of the witness search and because some edges may already exist. We greedily contract a vertex x that minimizes its *importance* $I(x)$ defined by

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

We maintain a priority queue that contains all vertices weighted by I. Initially all vertices are inserted with their exact importance. As long as the queue is not empty, we remove a vertex x with minimum importance $I(x)$ and contract it. This modifies the importance of other vertices. However, our weighting function is chosen such that only the importance of adjacent vertices is influenced, if the witness search was perfect. We therefore only update the importance values of all vertices y in the queue that are adjacent to x. In practice, with a limited witness search, we sometimes choose a vertex x with a sightly suboptimal $I(x)$. However, preliminary experiments have shown that this effect can be safely ignored. Hence, for the experiments presented in Section 2.9, we do not use lazy updates or periodic queue rebuilding as proposed in [77].

2.4 Metric-Independent Orders

The metric-dependent orders presented in the previous section lead to very good results on road graphs with travel time metric. However, the results for the distance metric are not as good and the orders are completely impracticable to compute Contraction Hierarchies without witness search as our experiments in Section 2.9 show. To support metric-independence, we therefore use *nested dissection* orders as suggested in [15] or ND-orders for short. An order π for G is computed recursively by determining a balanced separator S of minimum cardinality that splits G into two parts induced by the vertex sets A and B. The vertices of S are assigned to $\pi(n-|S|+1) \dots \pi(n)$ in an arbitrary order. Orders π_A and π_B are computed recursively and assigned to $\pi(1) \dots \pi(|A|)$ and $\pi(|A| + 1) \dots \pi(|A| + |B|)$, respectively. The base case of the recursion is reached when the subgraphs are empty. Computing ND-orders requires good graph bisectors, which in theory is NP-hard. However, recent years have seen heuristics that solve the problem very well even for continental road graphs [117, 47, 48]. This justifies assuming in our particular context that an efficient bisection oracle exists. We experimentally examine the performance of nested dissection orders computed by NDMetis [92] and KaHIP [117] in Section 2.9. After having obtained the nested dissection order we reorder the in-memory vertex IDs of the input graph accordingly, i.e., the contraction order of the reordered graph is the identity function. This improves cache locality and we have seen a resulting acceleration of a factor 2 to 3 in query times. In the remainder of this section we prepare and provide a theoretical approximation result.

For $\alpha \in (0,1)$, let K_α, be a class of graphs that is closed under subgraph construction and admits balanced separators S of cardinality $O(n^\alpha)$.

Lemma 1. For every $G \in K_\alpha$ an ND-order results in $O(n^\alpha)$ vertices in the maximum search space.

The proof of this lemma is a straightforward argument using a geometric series as described in [15]. As a direct consequence, the average number of vertices is also in $O(n^\alpha)$ and the number of arcs in $O(n^{2\alpha})$.

Lemma 2. For every connected graph G with minimum balanced separator S and every order π, the chordal super graph G^*_π contains a clique of $|S|$ vertices. Furthermore, there are at least $n/3$ vertices such that this clique is a subgraph of their search space in $G^\wedge_\pi$.

This lemma is a minor adaptation and extension of [99]. The authors of [99] only prove that such a clique exists but not that it lies within enough search spaces. We provide the full proof for self-containedness.

Proof. Consider the subgraph G_i of G^*_π induced by the vertices $\pi(1) \dots \pi(i)$. Do not confuse with the core graph $G_{\pi,i}$. Choose the smallest i, such that a connected component A exists in G_i such that $|A| \geq n/3$. As G is connected, such an A must exist. We distinguish two cases:

1. $|A| \leq 2n/3$: Consider the set of vertices S' adjacent to A in G^*_π but not in A. Let B be the set of all remaining vertices. S' is by definition a separator. It is balanced because $|A| \leq 2n/3$ and $|B| = n - \underbrace{|A|}_{\geq n/3} - \underbrace{|S'|}_{\geq 0} \leq 2n/3$. As S is minimum, we have that $|S'| \geq |S|$. For every pair of vertices $u \in S'$ and $v \in S'$ there exists a path through A as A is connected. The vertices u and v are not in G_i as otherwise they could be added to A. The ranks of u and v are thus strictly larger than i. On the other hand, the ranks of the vertices in A are at most i as they are part of G_i. The vertices u and v thus have the highest ranks on the path. They are therefore contracted last and therefore an edge $\{u,v\}$ in G^* must exist. S' is therefore a clique. Furthermore, from every $u \in A$ to every $v \in S'$ there exists a path such that v has the highest rank. Hence, v is in the search space of u, i.e., there are at least$|A| \geq n/3$ vertices whose search space contains the full S'-clique.

2. $|A| > 2n/3$: As i is minimum, we know that $\pi(i) \in A$ and that removing it disconnects A into connected subgraphs $C_1 \dots C_k$. We know that $\left|C_j\right| < n/3$ for all j because i is minimum. We further know that $|A| = 1 + \sum \left|C_j\right| > 2n/3$. We can therefore select a subset of components C_k such that the number of their vertices is at most $2n/3$ but at least $n/3$. Denote by A' their union. A' does not

contain $\pi(i)$. Consider the vertices S' adjacent to A' in G^*_π. The set S' contains $\pi(i)$. Using an argument similar to Case 1, one can show that $|S'| \geq |S|$. But since A' is not connected, we cannot directly use the same argument to show that S' forms a clique in G^*. Observe that $A' \cup \{\pi(i)\}$ is connected and thus the argument can be applied to $S' \backslash \{\pi(i)\}$ showing that it forms a clique. This clique can be enlarged by adding $\pi(i)$ as for every $v \in S' \backslash \{\pi(i)\}$ a path through one of the components C_k exists where v and $\pi(i)$ have the highest ranks and thus an edge $\{v, \pi(i)\}$ must exist. The vertex set S' therefore forms a clique of at least the required size. It remains to show that enough vertices exist whose search space contains the S' clique. As $\pi(i)$ has the lowest rank in the S' clique, the whole clique is contained within the search space of $\pi(i)$. It is thus sufficient to show that $\pi(i)$ is contained in enough search spaces. As $\pi(i)$ is adjacent to each component C_k, a path from each vertex $v \in A'$ to $\pi(i)$ exists such that $\pi(i)$ has maximum rank showing that S' is contained in the search space of v. This completes the proof as $|A'| \geq n/3$.

□

Theorem 1. Let G be a graph from K_α with a minimum balanced separator with $\Theta(n^\alpha)$ vertices. Then an ND-order gives an $O(1)$-approximation of the average and maximum search spaces of an optimal metric-independent contraction hierarchy in terms of vertices and arcs.

Proof. The key observation of this proof is that the top level separator solely dominates the performance. Denote by π_{nd} the ND-order and by π_{opt} an optimal order. First, we show a lower bound on the performance of π_{opt}. We then demonstrate that π_{nd} achieves this lower bound showing that π_{nd} is an $O(1)$-approximation.

As the minimum balanced separator has cardinality $\Theta(n^\alpha)$, we know by Lemma 2 that a clique with $\Theta(n^\alpha)$ vertices exists in $G^*_{\pi_{opt}}$. As this clique is in the search space of at least one vertex with respect to π_{opt}, we know that the maximum number of vertices in the search space is at least $\Omega(n^\alpha)$. Further, as this clique contains $\Theta(n^{2\alpha})$ arcs we also have a lower bound of $\Omega(n^{2\alpha})$ on the maximum number of arcs in a search space. From these bounds for the worst case search space, we cannot directly derive bounds for the average search space. Fortunately, Lemma 2 does not only tell us that this clique exists but that it must also be inside the search space of at least $n/3$ vertices. For the remaining $2n/3$ vertices we use a very pessimistic lower bound: We assume that their search space is empty. The resulting lower bound for the average number of vertices is $2/3 \cdot \Omega(0) + 1/3 \cdot \Omega(n^\alpha) = \Omega(n^\alpha)$ and the lower bound for the average number of arcs is $2/3 \cdot \Omega(0) + 1/3 \cdot \Omega(n^{2\alpha}) = \Omega(n^{2\alpha})$.

We required that $G \in K_\alpha$, i.e., that recursive $O(n^\alpha)$ balanced separators exist. This allows us to apply Lemma 1. We therefore know that the number of vertices in the maximum search space of $G^\wedge_{\pi_{nd}}$ is in $O(n^\alpha)$. In the worst-case this search space contains

$O(n^{2\alpha})$ arcs. As the average case can never be better than the worst case, these upper bounds directly translate to upper bounds for the average search space.

As the given upper and lower bounds match, we can conclude that π_{nd} is a $O(1)$-approximation in terms of average and maximum search space in terms of vertices and arcs. □

2.5 Constructing the Contraction Hierarchy

In this section, we describe how to efficiently compute the hierarchy $G_\pi^\wedge$ for a given graph G and order π. Weighted contraction hierarchies are commonly constructed using a dynamic adjacency array representation of the core graph. Our experiments show that this approach also works for the unweighted case, however, requiring more computational and memory resources because of the higher growth in shortcuts. It has been proposed by [146] to use hash-tables on top of the dynamic graph structure to improve speed but at the cost of significantly increased memory consumption. We show that the contraction hierarchy construction can be done significantly faster on unweighted and undirected graphs. In our toolchain, graph weights and arc directions are handled in the customization phase.

Denote by n the number of vertices in G (and $G_\pi^\wedge$), by m the number of edges in G, by $\hat{m}$ the number of arcs in $G_\pi^\wedge$, and by $\alpha(n)$ the inverse $A(n,n)$ Ackermann function. For simplicity we assume that G is connected. Our approach enumerates all arcs of $G_\pi^\wedge$ in $O(\hat{m}\,\alpha(n))$ running time and has a memory consumption in $O(m)$. To store the arcs of $G_\pi^\wedge$, additional space in $O(\hat{m})$ is needed. The approach is heavily based upon the method of the quotient graph [79]. To the best of our knowledge it has not yet been applied in the context of route planning and there exists no complexity analysis for the specific variant employed by us. Therefore we discuss both the approach and present a running time analysis in the remainder of the section.

Recall that to compute the contraction hierarchy $G_\pi^\wedge$ from a given input graph G and order π, one iteratively contracts each vertex, adding shortcuts between its neighbors. Let $G' = G_{\pi,i}$ be the core graph in iteration i. We do not store G' explicitly but employ a special data structure called *contraction graph* for efficient contraction and neighborhood enumeration. The contraction graph H contains both yet uncontracted core vertices as well as an independent set of virtually contracted *super vertices*, see Figure 2.2 for an illustration. These super vertices enable us to avoid the overhead of dynamically adding shortcuts to G'. For each vertex in H we store a marker bit indicating whether it is a super vertex. G' can be obtained by contracting all super vertices in H.

2.5.1 Contracting Vertices

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

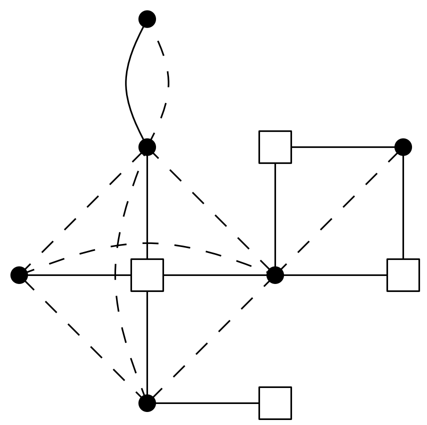


Figure 2.2: Dots represent vertices in G' and H. Squares are additional super vertices in H. Solid edges are in H and dashed ones in G'. The neighbors of each super vertex in H form a clique in G'. Furthermore, there are no two adjacent super vertices in H, i.e., they form an independent set.

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

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

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

2.5.2 Enumerating Neighbors

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

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

2.5.3 Performance Analysis

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

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

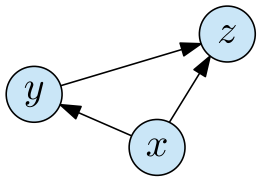


Figure 2.3: A triangle in $G_\pi^\wedge$. The triple $\{x,y,z\}$ is a lower triangle of the arc (y,z), an intermediate triangle of the arc (x,z), and an upper triangle of the arc (x,y).

of x is within $O(d(x)\alpha(n))$. Merging two vertices consists of redirecting a constant number of references within a linked list. The merge operation is thus in $O(1)$.

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

2.5.4 Adjacency Array

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

2.6 Enumerating Triangles

A triangle $\{x,y,z\}$ is a set of three adjacent vertices. A triangle can be an upper, intermediate or lower triangle with respect to an arc (x,y), as illustrated in Figure 2.3. A triangle $\{x,y,z\}$ is a *lower triangle* of (y,z) if x has the lowest rank among the three vertices. Similarly, $\{x,y,z\}$ is a *upper triangle* of (x,y) if z has the highest rank and $\{x,y,z\}$ is a *intermediate triangle* of (x,z) if y's rank is between the ranks of x and z. The triangles of an edge (a,b) can be characterized using the upward N_u and downward N_d neighborhoods of a and b. There is a lower triangle $\{a,b,c\}$ of an arc (a,b) if and only if $c \in N_d(a) \cap N_d(b)$. Similarly, there is an intermediate triangle $\{a,b,c\}$ of an arc (a,b) with $\pi^{-1}(a) < \pi^{-1}(b)$ if and only if $c \in N_u(a) \cap N_d(b)$ and an upper triangle

$\{a,b,c\}$ of an arc (a,b) if and only if $c \in N_u(a) \cap N_u(b)$. The triangles of an arc can thus be enumerated by intersecting the neighborhoods of the arc's endpoints.

Efficiently enumerating all lower triangles of an arc is an important base operation of the customization (Section 2.7) and path unpacking algorithms (Section 2.8). It can be implemented using adjacency arrays or accelerated using extra preprocessing. In addition to the vertices of a triangle we are interested in the IDs of the participating arcs as we need these to access the metric of an arc.

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

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

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

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

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

Comparison with CRP. Triangle preprocessing has similarities with micro and macro code in CRP [54]. In the following, we compare the space consumption of these two approaches against our lower triangles preprocessing scheme. At this stage we do not yet consider travel direction on arcs. Hence, let t be the number of undirected triangles and m be the number of arcs in $G_\pi^\wedge$; further let t' be the number of directed triangles and m' be the number of arcs used in [54]. If every street is a one-way street, then $m' = m$ and $t' = t$; otherwise, without one-way streets, $m' = 2m$ and $t' = 2t$.

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

Hence, our approach always requires less space than micro code. It has similar space consumption as macro code if one-way streets are rare, otherwise it needs at most twice as much data. However, the main advantage of our approach over macro code is that it allows for random access, which is crucial in the algorithms presented in the following sections.

2.7 Customization

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

For correctness of the CH query algorithms (c.f. Section 2.8) it is necessary that between any pair of vertices s and t a shortest up-down st-path in $G_\pi^\wedge$ exists with the same distance as the shortest st-path in the input graph G. In other words, $\text{dist}_\text{I}(s, t) = \text{dist}_\text{A}(s, t) = \text{dist}_\text{UD}(s, t)$ must hold for all vertices s and t. We say that a metric that fulfills $\text{dist}_\text{I}(s,t) = \text{dist}_\text{A}(s,t)$ *respects* the input weights. If additionally $\text{dist}_\text{A}(s,t) = \text{dist}_\text{UD}(s,t)$ holds, we call the metric *customized*. Customized metrics are not necessarily unique. However, there is a special customized metric, called *perfect* metric m_P, where for every arc (x,y) in $G_\pi^\wedge$ the weight of this arc $m_P(x,y)$ is equal to the shortest path distance $\text{dist}_\text{I}(x,y)$. We optionally use the perfect metric to perform perfect witness search.

Constructing a respecting metric is trivial: Assign to all arcs of $G_\pi^\wedge$ that already exist in G their input weight and to all other arcs $+\infty$. Computing a customized metric is less trivial. We therefore describe in Section 2.7.1 the basic customization algorithm that computes a customized metric m_C given a respecting one. Afterwards, we describe the perfect customization algorithm that computes the perfect metric m_P given a customized one (such as for example m_C). Finally, we show how to employ the perfect metric to perform a perfect witness search.

2.7.1 Basic Customization

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

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

Theorem 2. Every respecting metric that additionally fulfills the lower triangle inequality is customized.

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

2.7.2 Perfect Customization

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

Theorem 3. After the perfect customization, $m_P(x,y)$ corresponds to the shortest xy-path distance for every arc (x,y), i.e., m_P is the perfect metric.

Proof. We have to show that after the algorithm has finished processing a vertex x, all of its outgoing arcs in $G_\pi^\wedge$ are weighted by the shortest path distance. We prove this by induction over the level of the processed vertices. The top-most vertex is the only vertex in the top level. It does not have any upward arcs and thus the algorithm does not have anything to do. This forms the base case of the induction. In the inductive step, we assume that all vertices with a strictly higher level have already been processed. As consequence, we know that the upward neighbors of x form a clique weighted by shortest path distances. Denote these neighbors by y_i. The situation is depicted in Figure 2.4. The weights of the y_i encode a complete shortest path distance table between the upward neighbors of x.

Pick some arbitrary arc (x, y_j). We show the correctness of our algorithm by proving that either $m_C(x, y_j)$ is already the shortest path distance or a neighbor $y_k \in N_u(x)$ must exist such that $x \rightarrow y_k \rightarrow y_j$ is a shortest up-down path. For the rest of this paragraph assume the existence of y_k, we prove its existence in the next paragraph. If $m_C(x, y_j)$ is already the shortest xy_j-path distance, then enumerating triangles will not

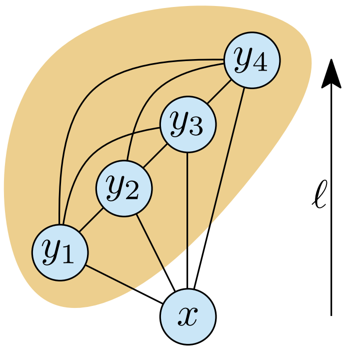


Figure 2.4: The vertices $y_1 \ldots y_4$ denote the upper neighborhood $N_u(x)$ of x. They form a clique (orange area) because x was contracted first. As $\ell(x) < \ell(y_j)$ for every j, we know by the induction hypothesis that the arcs in this clique are weighted by shortest path distances. We therefore have an all-pair shortest path distance table among all y_j. We have to show that using this information we can compute shortest path distances for all arcs outgoing of x.

change $m_C(x,y_j)$ and is thus correct. If $m_C(x,y_j)$ is not the shortest xy_j-path distance, then enumerating all intermediate and upper triangles of (x,y_j) is guaranteed to find the $x \to y_k \to y_j$ path and thus the algorithm is correct. The upper triangles correspond to paths with $\ell(y_k) > \ell(y_j)$ while the intermediate triangles correspond to paths with $\ell(y_k) < \ell(y_j)$.

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

2.7.3 Perfect Witness Search

Using the perfect customization algorithm, we can efficiently compute the weighted CH with a minimum number of arcs with respect to the same contraction order. We present two variants of our algorithm. The first variant consists of removing each arc (x,y) whose weight $m_C(x,y)$ after basic customization does not correspond to the shortest xy-path distance $m_P(x,y)$. While simple and correct, this variant does not remove as many arcs as possible, if a pair of vertices a and b exists in the input graph such that there are multiple shortest ab-paths. The second variant[2] also removes these additional arcs. An arc (x,y) is removed if and only if an upper or intermediate triangle $\{x,y,z\}$ exists such that the shortest path from x over z to y is no longer than the shortest xy-path. However, before we can prove the correctness of the second variant, we need to introduce some technical machinery, which will also be needed in the correctness proof of the stalling query algorithm. We define the "height" of a not-necessarily up-down path in G^*_π. We show that with respect to every customized metric, for every path that is not up-down, an up-down path must exist that is strictly higher and is not longer.

2.7.3.1 Variant for Graphs with Unique Shortest Paths

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

Theorem 4. If the input graph has unique shortest paths between all pairs of vertices, then we can remove an arc (x,y) from the CH if and only if $m_P(x,y) \neq m_C(x,y)$.

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

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

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

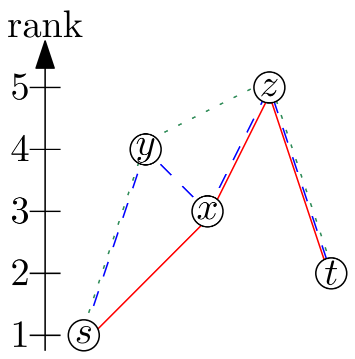


Figure 2.5: The rank sequence of the solid red path is $[3,2,1]$. 3 is the minimum of the ranks of the endpoints of the $\{x,z\}$ edge. Similarly, 2 is induced by the $\{z,t\}$ edge and 1 by the $\{s,x\}$ edge. The rank sequence of the blue dashed path is $[3,3,2,1]$ and the rank sequence of the green dotted path is $[4,2,1]$. The solid red path is the lowest followed by the blue dashed path and the green dotted path is the highest.

2.7.3.2 Variant for General Graphs

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

We want to order paths by "height". To achieve this, we first define for each path K in G^*_π its *rank sequence*. We order paths by comparing the rank sequences lexicographically. Denote by v_i the vertices in K. For each edge $\{v_i, v_{i+1}\}$ in K the rank sequence contains $\min\{\pi^{-1}(v_i), \pi^{-1}(v_{i+1})\}$. The numbers in the rank sequences are sorted in non-increasing order. Two paths have the same height if one rank sequence is a prefix of the other. Otherwise we compare the rank sequences lexicographically. This ordering is illustrated in Figure 2.5. We prove the following technical lemma:

Lemma 3. Let m_C be some customized metric. For every st-path K that is no up-down path, an up-down st-path Q exists such that Q is strictly higher than K and Q is not longer than K with respect to m_C.

Proof. Denote by v_i the vertices on the path K. As K is no up-down path, there must exist a vertex v_i on K that has lower ranks than its neighbors v_{i-1} and v_{i+1}. v_{i-1} and v_{i+1}

are different vertices because they are part of a shortest path and zero weights are not allowed. Further, as v_i is contracted before its neighbors, there must be a edge between v_{i-1} and v_{i+1}. As the metric is customized, $m_C(v_{i-1}, v_{i+1}) \leq m_C(v_{i-1}, v_i) + m_C(v_i, v_{i+1})$ must hold. We can bypass v_i by replacing the subpath (v_{i-1}, v_i, v_{i+1}) with the single arc (v_{i-1}, v_{i+1}) without making the path longer. Denote this new path by R. R is higher than K as we replaced $\pi^{-1}(v_i)$ in the rank sequence by $\min\{\pi^{-1}(v_{i-1}), \pi^{-1}(v_{i+1})\}$, which must be larger. Either R is an up-down path or we apply the argument iteratively. In each iteration, the path loses a vertex and therefore we can guarantee that eventually we obtain an up-down path that is higher than K and not longer. This is the desired up-down path Q that is not longer than K and strictly higher. □

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

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

Theorem 5. We can remove an arc (x, y) if and only if an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$.

Proof. We need to show that for every pair of vertices s and t a shortest up-down st-path exists, that uses no removed arc. We show that a highest shortest up-down st-path has this property. As the metric is customized, we know that a shortest up-down st-path K exists before removing any arcs. If K does not contain an arc (x, y) for which an upper or intermediate triangle $\{x, y, z\}$ exists with $m_P(x, y) = m_P(x, z) + m_P(z, y)$, then there is nothing to show. Otherwise, we modify K by inserting z between x and y. This does not modify the length of K, but we can no longer guarantee that K is an up-down path. If $\{x, y, z\}$ was an intermediate triangle, then K is still an up-down path. However, it is strictly higher, as we added $\pi^{-1}(z)$ into the rank sequence, which is guaranteed to be larger than $\pi^{-1}(x)$. If $\{x, y, z\}$ was an upper triangle, then K is no longer an up-down path. Fortunately, using Lemma 3 we can transform K into an up-down path, that is not longer and strictly higher. In both cases, the new K is an up-down path or we apply the argument iteratively. As K gets strictly higher in each iteration and the number of up-down paths is finite, we know that we will eventually obtain a shortest up-down st-path where no arc can be removed.

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

2.7.4 Parallelization

The basic customization can be parallelized by processing the arcs (x, y) that depart within a level in parallel. Between levels, we need to synchronize all threads using a barrier. As all threads only write to the arc they are currently assigned to and only read from arcs processed in a strictly lower level, we can thus guarantee that no read/write conflict occurs. Hence, no locks or atomic operations are needed.

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

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

In the proof of Theorem 3 we have shown that for every vertex x and arc (x, y_i) either the arc (x, y_i) already has the shortest path distance or an upper or intermediate triangle $\{x, y_i, y_j\}$ exists such that $x \to y_j \to y_i$ is a shortest path. No matter in which order the threads process the arcs, they do not modify shortest path weights. This implies that the shortest path $x \to y_j \to y_i$ is thus retained, regardless of the execution order. This shortest path is not modified and is guaranteed to exist before any arcs outgoing from the current level are processed. Every thread is thus guaranteed to see it. However, other weights can be modified. Fortunately, this is not a problem as long as we can guarantee that no thread sees a value that is below the corresponding shortest path distance. Therefore, if we can guarantee that thread B either sees the old value or the new value, as is the case on x86 processors, then the algorithm is correct.

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

2.7.5 Directed Graphs

Up to now we have focused on customizing undirected graphs. If the input graph G is *directed*, our toolchain works as follows: Based on the *undirected unweighted* graph induced by G we compute a vertex ordering π (Section 2.4), build the upward directed Contraction Hierarchy $G_\pi^\wedge$ (Section 2.5), and optionally perform triangle preprocessing (Section 2.6). For customization, however, we consider two weights per arc in $G_\pi^\wedge$, one for each direction of travel. One-way streets are modeled by setting the weight corresponding to the forbidden traversal direction to ∞. With respect to π we define an upward metric m_u and a downward metric m_d on $G_\pi^\wedge$. For each arc $(x,y) \in G$ in the directed input graph with input weight $w(x,y)$, we set $m_u(x,y) = w(x,y)$ if $\pi^{-1}(x) < \pi^{-1}(y)$, i.e., if x is ordered before y; otherwise, we set $m_d(x,y) = w(x,y)$. All other values of m_u and m_d are set to ∞. In other words, each arc $(x,y) \in G_\pi^\wedge$ of the Contraction Hierarchy has upward weight $m_u(x,y) = w(x,y)$ if $(x,y) \in G$, downward weight $m_d(x,y) = w(y,x)$ if $(y,x) \in G$, and ∞ otherwise.

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

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

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

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

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

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

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

2.7.6 Single Instruction Multiple Data

The weights attached to each arc in the CH can be replaced by an interleaved set of k weights by storing for every arc a vector of k elements. Vectors allow us to customize all k metrics in one go, amortizing triangle enumeration time. Additionally, they allow us to use single instruction multiple data (SIMD) operations. As we use essentially two metrics to enable directed graphs, we can store both of them in a 2-dimensional vector. This allows us to handle both directions in a single processor instruction. Similarly, if we have k directed input weights we can store them in a $2k$-dimensional vector. Depending on the width of SIMD registers, we might require more than one SIMD instruction per vector. However, this approach still benefits from amortized triangle enumeration time, which is only done once per arc.

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

2.7.7 Partial Updates

Until now we have only considered computing metrics from scratch. However, in many scenarios this is overkill, as we know that only a few edge weights of the input graph were changed. It is unnecessary to redo all computations in this case. The ideas employed by our algorithm are somewhat similar to those presented in [77], but our situation differs as we know that we do not have to insert or remove arcs. Denote by $U = \left\{((x_i, y_i), w_i^{\text{new}})\right\}$ the set of arcs whose weights should be updated, where (x_i, y_i) is the arc ID and w_i^{new} the new weight. Modifying the weight of one arc can trigger further changes. However, these new changes have to be at higher levels. We therefore organize U as a priority queue ordered by the level of x_i. We iteratively remove arcs from the queue and apply the change. If new changes are triggered we insert these into the queue. The algorithm terminates once the queue is empty.

Denote by (x, y) the arc that was removed from the queue and by w^{new} its new weight and by w^{old} its old weight. We first have to check whether w^{new} can be bypassed using a lower triangle. For this reason, we iterate over all lower triangles $\{x,y,z\}$ of (x,y) and perform $w^{\text{new}} \leftarrow \min\{w^{\text{new}}, m(z,x)+m(z,y)\}$. Furthermore, if $\{x,y\}$ is an edge in the input graph G, we might have overwritten its weight with a shortcut weight, which after the update might not be shorter anymore. Hence, we additionally test that w^{new} is not larger than the input weight. If after both checks $w^{\text{new}} = m(x,y)$ holds, then no change is necessary and no further changes are triggered. If w^{old} and w^{new} differ we iterate over all upper triangles $\{x,y,z\}$ of (x,y) and test whether $m(x,z) + w^{\text{old}} = m(y,z)$ holds and if so the weight of the arc (y,z) must be set to $m(x,z) + w^{\text{new}}$. We add this change to the queue. Analogously we iterate over all intermediate triangles $\{x,y,z\}$ of (x,y) and queue up a change to (z,y) if $m(x,z) + w^{\text{old}} = m(z,y)$ holds.

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

2.8 Distance and Shortest Path Queries

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

2.8.1 Basic Query Algorithm

The basic query runs two instances of Dijkstra's algorithm on $G_\pi^\wedge$ from s and from t. If G is undirected, then both searches use the same metric. Otherwise if G is directed the search from s uses the upward metric m_u and the search from t the downward metric m_d. In either case, in contrast to [77], they operate on the same upward search graph $G_\pi^\wedge$. Once the radius of one of the two searches is larger than the shortest path found so far, we stop the search because we know that no shorter path can exist. We alternate between processing vertices in the forward search and processing vertices in the backward search.

2.8.2 Stalling

We implemented a basic version of an optimization presented in [77, 115] called stall-on-demand. The optimization exploits that the shortest strictly upward sv-path in $G_\pi^\wedge$ can be longer than the shortest sv-path in G_π^*, which can go up and down arbitrarily. The search from s only finds upward paths. If we observe that an up-down path exists that is not longer, then we can prune the upward search. Denote by x the vertex removed from the queue. We iterate over all outgoing arcs (x,y) and test whether $d(x) \geq m(x,y)+d(y)$ holds. If it holds for some arc we prune x by not relaxing its outgoing arcs.

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

Theorem 6. The upward search can be pruned when $d(x) \geq m(x,y) + d(y)$ holds.

Proof. We show that for every pair of vertices s and t an unprunable, shortest, up-down st-path exists. Our proof relies on Lemma 3 which orders paths by height and states that st-path that are no up-down paths can be transformed into up-down paths that are no longer and strictly higher. We know that some shortest st-path K exists. If K is not pruned, then there is nothing to show. If K is pruned, then there exists a vertex x on K at which the search is pruned. Without loss of generality we assume that x lies on the upward part of K. Further, there must exist a vertex y and a path Q from s to x going through y such that Q is not longer than the sx-prefix of K. Consider the path R obtained by concatenating Q with the xt-suffix of K. R is by construction not longer than K. If x is the highest vertex on K then R is an up-down path and R is strictly higher. Otherwise, R is no up-down path, but using Lemma 3 R can be transformed into an up-down path that is strictly higher and no longer. In both cases, R is no longer and strictly higher. Either, R is unprunable or we apply the argument iteratively. As there are only finitely many up-down paths and each iteration increases the height of R, we eventually end up at an unprunable, shortest, up-down st-path, which concludes the proof. □

2.8.3 Elimination Tree-based Query Algorithm

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

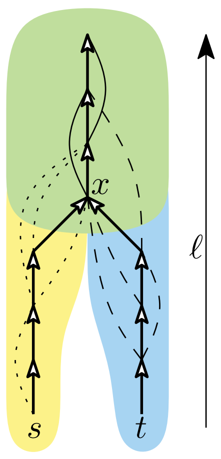


Figure 2.6: The union of the yellow and green areas is the search space of s. Analogously the union of the blue and green areas is the search space of t. The green area is the intersection of both search spaces. The dotted arcs start in the search space of s, but not in the search space of t. Analogously the dashed arcs start in the search space of t, but not in the search space of s. The solid arcs start in the intersection of the two search spaces. The vertex x is the least common ancestor of s and t.

We store two tentative distance arrays $d_f(v)$ and $d_b(v)$. Initially these are all set to ∞. In a first step we compute the lowest common ancestor (LCA) x of s and t in the elimination tree. We do this by simultaneously enumerating all ancestors of s and t by increasing rank until a common ancestor is found. In a second step we iterate over all vertices y on the tree-path from s to x and relax all forward arcs of such y. In a third step we do the same for all vertices y from t to x in the backward search. In a fourth step we iterate over all vertices y from x to the root r and relax all forward and backward arcs. Further, in the fourth step we also determine the vertex z that minimizes $d_f(z) + d_b(z)$. A shortest up-down path must exist that goes through z. Knowing z is necessary to determine the shortest path distance and to compute the sequence of arcs that compose the shortest path. In a fifth cleanup step we iterate over all vertices from s and t to the root r to reset all d_f and d_b to ∞. This fifth step avoids having to spend $O(n)$ running time to initialize all tentative distances to ∞ for each query. Consider the situation depicted in Figure 2.6. In the first step the algorithm determines x. In the second step it relaxes all dotted arcs and the tree arcs departing in the lightgray area. In the third step all dashed arcs and the tree arcs departing in the middlegray area and in the fourth step the solid arcs and the remaining tree arcs follow.

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

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

2.8.4 Path Unpacking

All presented shortest path queries only compute shortest up-down paths. This is enough to determine the distance of a shortest path in the original graph. However, if the sequence of edges that form a shortest path should be computed, then the up-down path must be unpacked. The original CH of [77] unpacks an up-down path by storing for every arc (x, y) the vertex z of the lower triangle $\{x, y, z\}$ that caused the weight at $m(x, y)$. This information depends on the metric and we want to avoid storing additional metric-dependent information. We therefore resort to a different strategy: Denote by $p_1 \ldots p_k$ the up-down path found by the query. As long as a lower triangle $\{p_i, p_{i+1}, x\}$ of an arc (p_i, p_{i+1}) exists with $m(p_i, p_{i+1}) = m(x, p_i) + m(x, p_{i+1})$, our algorithm inserts the vertex x between p_i and p_{i+1} into the path.

2.9 Experiments

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

Compiler and Machine. We implemented our algorithms in C++, using g++ 4.7.1 with -O3 for compilation. The customization and query experiments were run on a dual 8-core Intel Xeon E5-2670 processor, which is based on the Sandy Bridge architecture, clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. The order computation experiments reported in Table 2.9 were run on a single core of an Intel Core i7-2600K processor.

Instances. We evaluate three large instances of practical relevance in detail. In Section 2.11, we provide summarized experiments on further instances. The sizes of our main test instances are reported in Table 2.8: The DIMACS Europe graph was provided by PTV for the DIMACS challenge [55]. The vertex positions are depicted

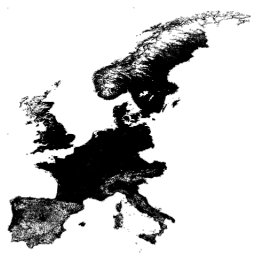


Figure 2.7: All vertices in the DIMACS-Europe graph.

in Figure 2.7. The instance is composed of several, mostly West European countries as can be seen from the figure. It is the standard benchmarking instance used by road routing papers over the past few years. Besides roads it also contains a few ferries to connect Great Britain and some other islands with the continent. The Europe graph analyzed here is its largest strongly connected component, which is a common method to remove bogus vertices. The numbers in Table 2.8 display statistics after computing the strongly connected component. The graph is directed and we consider two different weights. The first weight is the travel time and the second weight is the straight-line distance between two vertices on a perfect Earth sphere. In the input data highways are often modeled using only a small number of vertices compared to the streets going through the cities. This differs from other data sources, such as OpenStreetMap that have a high number of vertices on highways to model road bends. As demonstrated in Section 2.11.1, degree-2 vertices do not hamper the performance of CHs. The Karlsruhe graph is a subgraph of the PTV graph for a larger region around Karlsruhe. We consider the largest connected component of the graph induced by all vertices with a latitude between 48.3 and 49.2, and a longitude between 8 and 9.

The TheFrozenSea graph is based on the largest Star Craft map presented in [135]. The map is composed of square tiles having at most eight neighbors and distinguishes between walkable and non-walkable tiles. These are not distributed uniformly, but rather form differently-sized pockets of freely walkable space alternating with *choke points* of very limited walkable space. The corresponding graph contains for every walkable tile a vertex and for every pair of adjacent walkable tiles an edge. Diagonal edges are weighted by $\sqrt{2}$, while horizontal and vertical edges have weight 1. The graph is symmetric, i.e., for each forward arc there is a backward arc and contains large grid subgraphs.

For comparability with other works, we report in Table 2.8 the time needed by Dijkstra's algorithm. Our implementation uses a 4-ary heap. As usual, it is unidirectional and employs a stopping criterion for point-to-point queries. Performance was obtained before re-ordering vertices in memory.

Instance	# Vertices	# Arcs	# Edges	Sym-metric?	Dijkstra [ms]
Karlsruhe	120 412	302 605	154 869	○	6
TheFrozenSea	754 195	5 815 688	2 907 844	●	58
Europe	18 010 173	42 188 664	22 211 721	○	1 560

Table 2.8: Benchmark instances. We report the number of vertices and directed arcs, as well as the number of edges in the induced undirected graph. For comparison, we also report the running time of Dijkstra's algorithm (with stop criterion) averaged over 10 000 st-queries, where s and t are chosen uniformly at random.

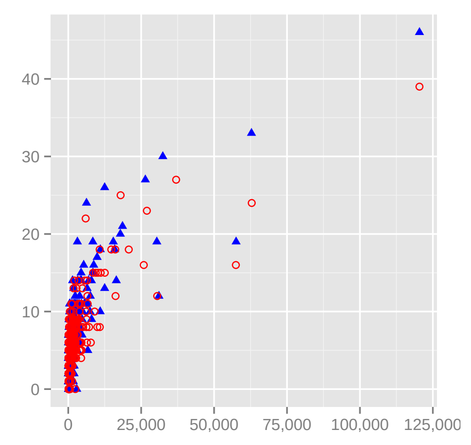
Instance	MetDep	Metis	KaHIP
Karlsruhe	4.1	0.5	$<$ 1 532
TheFrozenSea	1 280.4	4.7	$<$ 22 828
Europe	813.5	131.3	$<$ 249 082

Table 2.9: Duration of order computation in seconds without parallelization.

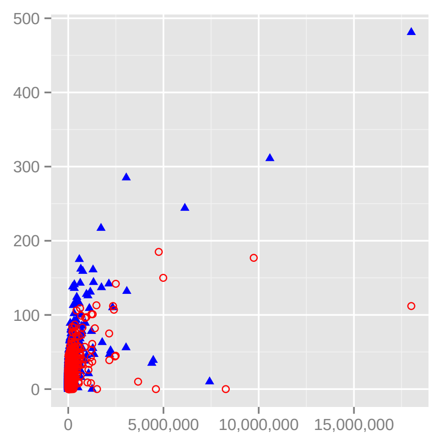
2.9.1 Computing Orders

We analyze three different vertex orders: 1) The greedy metric-dependent order in the spirit of [77]. We refer to it as "MetDep" in the tables. 2) The Metis 5.0.1 graph partitioning package contains a tool called `ndmetis` to create ND-orders. 3) KaHIP 0.61 provides just graph partitioning tools. We therefore implemented a very basic nested dissection computation on top of it: For every graph we iteratively compute bisections with different random seeds, using the "strong" configuration of KaHIP, until for ten consecutive runs no better cut is found. We recursively bisect the graph until the parts are too small for KaHIP to handle and assign the order arbitrarily in these small parts. We set the imbalance for KaHIP to 20%. Our program is solely tuned for quality completely disregarding running time. We report the running times only as upper bounds. The running times reported in Table 2.9 cannot be used to conclude that the *original* KaHIP package is slow.

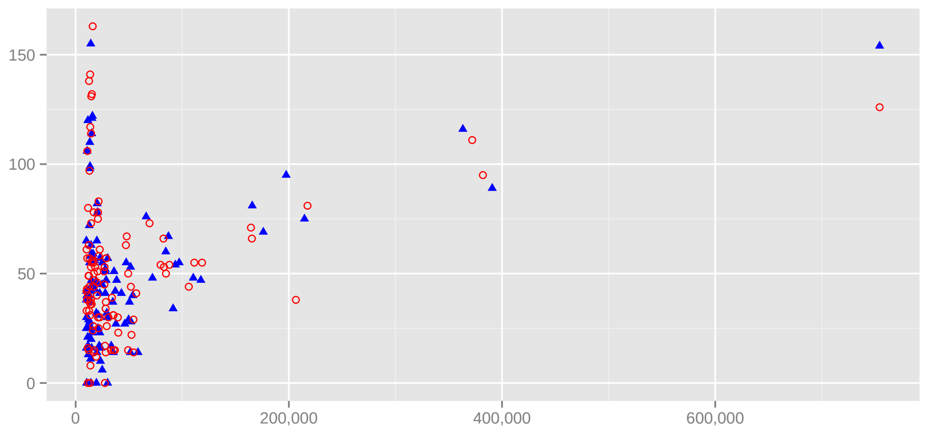
Table 2.9 reports the times needed to compute the orders. Interestingly, Metis is even faster than the metric-dependent greedy vertex ordering strategy. Figure 2.10 shows the sizes of the computed separators. As expected, KaHIP results in better quality. Moreover, our road graphs have separators that seem to follow a cubic-root law. A more rigorous complexity analysis as in [101] would be interesting but is not the focus of this work. On Karlsruhe, the separator sizes steadily decrease from the top level to



(a) Karlsruhe



(b) Europe



(c) TheFrozenSea

Figure 2.10: The amount of vertices in the separator (vertical) vs the number of vertices in the subgraph being bisected (horizontal). We only plot the separators for (sub)graphs of at least 1000 vertices. The red hollow circles is KaHIP and the blue filled triangles is Metis.

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

Table 2.11: Construction of the Contraction Hierarchy. We report the time in seconds required to compute the arcs in $G_\pi^\wedge$ given a KaHIP ND-order π. No witness search is performed. No weights are assigned.

the bottom level, making Theorem 1 directly applicable under the assumption that no significantly better separators exist. The KaHIP separators on the Europe graph have a different structure on the top level. The separators first increase before they get smaller. This is because of the special structure of the European continent. For example the cut separating Great Britain and Spain from France is far smaller than one would expect for a graph of that size. In the next step, KaHIP cuts Great Britain from Spain which results in one of the extremely thin cuts observed in the plot. Interestingly, Metis is not able to find these cuts that exploit the continental topology. The game map has a structure that differs from road graphs as the plots have two peaks. This effect results from the large grid subgraphs. Grids have $\Theta(\sqrt{n})$ separators, whereas at the higher levels the choke points result in separators that approximately follow a cubic-root law. At some point the bisector has cut all choke points and has to start cutting through the grids. The second peak is at the point where this switch happens.

2.9.2 Contraction Hierarchy Construction

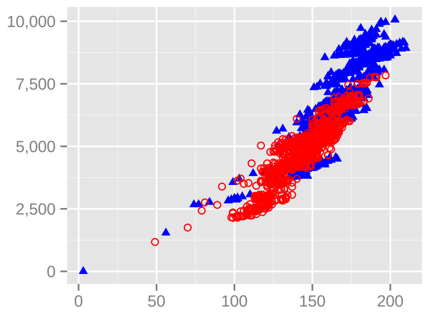
Table 2.11 compares the performance of our specialized Contraction Graph data structure, described in Section 2.5, to the dynamic adjacency structure, as used in [77] to compute undirected and unweighted CHs. We do not report numbers for the hash-based approach of [146] as it is fully dominated. Our data structure dramatically improves performance. However to be fair, our approach cannot immediately be extended to directed or weighted graphs. Fortunately, this is no problem as we can introduced weights and directions during the customization phase.

2.9.3 Contraction Hierarchy Size

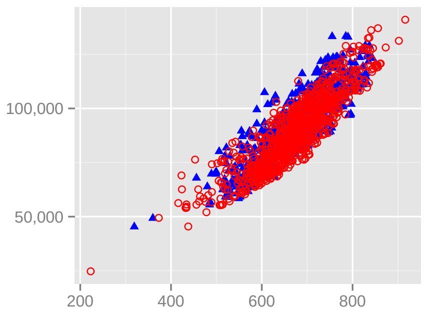
In Table 2.12, we report the resulting CH sizes for various approaches. Computing a CH on Europe *without witness search* with the greedy, metric-dependent order is infeasible even using the Contraction Graph data structure. This is also true if we only want to count the number of arcs: We aborted calculations after several days. However, we can state with certainty that there are at least 1.3×10^{12} arcs in the CH, and the maximum

	Order	Witness	# Arcs [·10^3]		# Tri-angles [·10^6]	Average upward search space size			
						unweighted		weighted	
			undir.	upward		# Nodes	# Arcs	# Nodes	# Arcs
Karlsruhe	D	N	21 926	17 661	37 440	5 870	15 786 622	5 246	11 281 564
		H	—	244	—	—	—	108	503
		P	—	239	—	—	—	107	498
	M	N	478	463	2.6	164	6 579	163	6 411
		P	—	340	—	—	—	152	2 903
	K	N	528	511	2.2	143	4 723	142	4 544
		P	—	400	—	—	—	136	2 218
TheFrozenSea	D	H	—	6 400	—	—	—	1 281	13 330
	M	N	21 067	21 067	602	676	92 144	676	92 144
		P	—	10 296	—	—	—	644	32 106
	K	N	25 100	25 100	864	674	89 567	674	89 567
		P	—	10 162	—	—	—	645	24 782
Europe	D	H	—	33 912	—	—	—	709	4 808
	M	N	70 070	65 546	1 409	1 291	464 956	1 289	453 366
		P	—	47 783	—	—	—	1 182	127 588
	K	N	73 920	69 040	578	652	117 406	651	108 121
		P	—	55 657	—	—	—	616	44 677

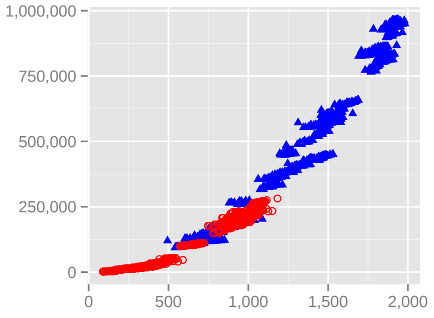
Table 2.12: Size of the Contraction Hierarchies for different instances and orders. We use following abbreviations: “D” for metric-dependent order, “M” for Metis order, “K” for KaHIP order, “N” for no witness search, “H” for heuristic, and “P” for perfect witness search. We report the average number of vertices and arcs reachable in the upward search space of a vertex. This number varies depending on whether a witness search is performed or not. It also varies depending on whether we follow one-way streets in both directions or not. We also report the number of triangles. As an indication for query performance, we report the average search space size in vertices and arcs, by sampling the search space of 1000 random vertices. Metis and KaHIP orders are metric-independent. We report resulting figures after applying different variants of witness search. A heuristic witness search is one that exploits the metric in the preprocessing phase. A perfect witness search is described in Section 2.7.



(a) Karlsruhe



(b) TheFrozenSea



(c) Europe

Figure 2.13: The number of vertices (horizontal) vs. the number of arcs (vertical) in the search space of 1000 random vertices. The red hollow circles is KaHIP and the blue filled triangles is Metis.

upward vertex degree is at least 1.4×10^6. As the original graph has only 4.2×10^7 arcs, it is safe to assume that, using this order, it is impossible to achieve a speedup over Dijkstra's algorithm on the input graph. However, at least on the Karlsruhe graph we can compute the CH without witness search and perform a perfect witness search. The numbers show that the heuristic witness search employed by [77] is nearly optimal. Furthermore, the numbers clearly show that using metric-dependent orders in a metric-independent setting, i.e., without witness search, results in unpractical CH sizes. However, they also show that a metric-dependent order exploiting the weight structure dominates ND-orders. In Figure 2.13 we plot the number of arcs in the search space vs. the number of vertices. The plots show that the KaHIP order significantly outperforms the Metis order on the road graphs whereas the situation is a lot less clear on the game map where the plots suggest nearly a tie. KaHIP only slightly outperforms Metis, when using a perfect customization. Table 2.14 examines the elimination tree.

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

Table 2.14: Elimination tree characteristics. The reported numbers are exact while the numbers in Table 2.12 are sampled over a random subset of vertices. We also report upper bounds on the tree width of the input graphs, after dropping the directions of arcs.

Most noticeably, it has a relatively small height compared to the number of vertices in G. The height of the elimination tree corresponds[3] to the number of vertices in the (undirected) search space. As the ratio between the maximum and the average height is only about 2, we know that no special vertex exists that has a search space significantly differing from the numbers shown in Table 2.14.

The tree width of a graph is a measure widely used in theoretical computer science and thus interesting on its own. The notion of tree width is deeply coupled with the notion of chordal super graphs and vertex separators. See [28] for details. The authors show in their Theorem 6 that the maximum upward degree $d_u(v)$ over all vertices v in $G_\pi^\wedge$ is an upper bound to the tree width of a graph G. This theorem yields a straightforward algorithm that gives us the upper bounds presented in Table 2.14.

Interestingly, these numbers correlate with our other findings: The difference between the bounds on the road graphs reflect that the KaHIP orders are better than Metis orders. On the game map there is nearly no difference between Metis and KaHIP, which is in accordance with all other performance indicators. The fact that the tree width grows with the graph size reflects that the running times are not independent of the graph size. These numbers strongly suggest that road graphs are not part of a graph class of constant tree width. Fortunately, the tree width seems to grow sub-linearly: Our findings from Figure 2.10 suggest that assuming a tree width of $O(\sqrt[3]{n})$ for road graphs of n vertices might come close to reality, although more rigorous analysis as in [101] would be necessary to substantiate this claim.

In Table 2.15, we evaluate the witness search performances for different metrics.

[3]The numbers in Table 2.12 and Table 2.14 deviate a little because the search spaces in the former table are sampled while in the latter we compute precise values.

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

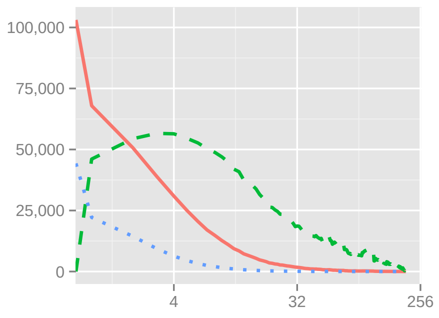
Table 2.15: Detailed analysis of the size of CHs after perfect witness search. We evaluate uniform, random and distance weights on the Karlsruhe input graph. Random weights are sampled from $[0, 10000]$. The distance weight is the straight distance along a perfect Earth sphere's surface. All weights respect one-way streets of the input graph.

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

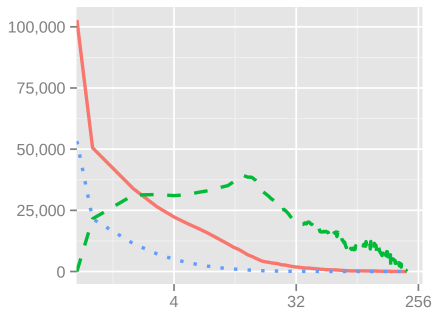
Table 2.18: Precomputed triangles. As show in Section 2.6, the memory needed is proportional to $2t + m + 1$, where t is the triangle count and m the number of arcs in the CH. We use 4 byte integers. We report t and m for precomputing all levels ("full") and all levels below a reasonable threshold level ("partial"). We further indicate how much percent of the total unaccelerated enumeration time is spent below the given threshold level. We chose the threshold level such that this factor is about 33 %.

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

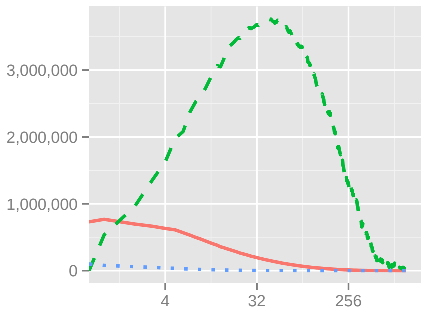
Recall that a CH is a DAG, and in DAGs each vertex can be assigned a level. If a vertex can be placed in several levels we put it in the lowest level. Figure 2.16 illustrates the amount of vertices and arcs in each level of a CH. The many highly ranked extremely thin levels are a result of the top level separator clique: Inside a clique, every vertex must be on its own level. A few big separators therefore significantly increase the level count.



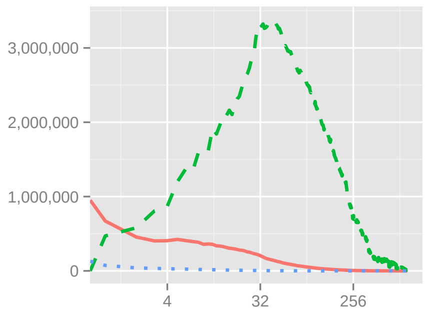
(a) Karlsruhe/KaHIP



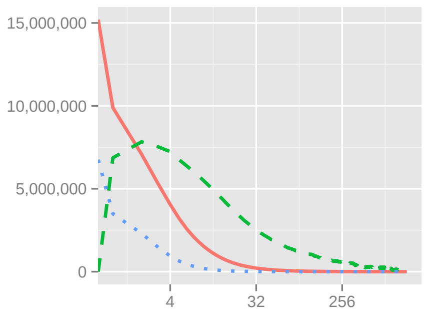
(b) Karlsruhe/Metis



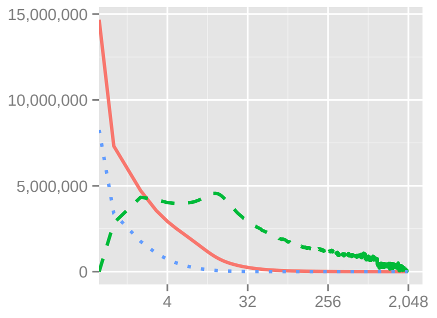
(c) TheFrozenSea/KaHIP



(d) TheFrozenSea/Metis

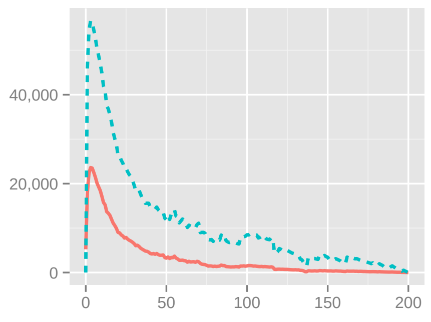


(e) Europe/KaHIP

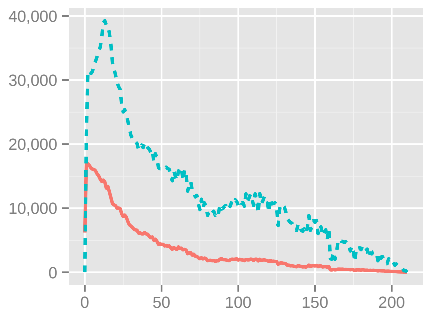


(f) Europe/Metis

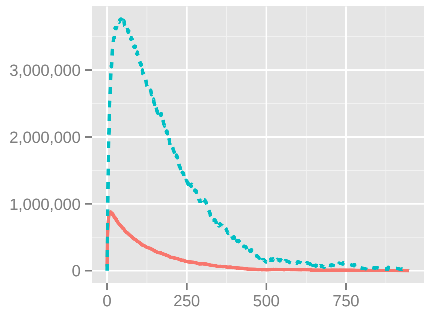
Figure 2.16: The number of vertices (y-axis) per level (x-axis) is the blue dotted line. The number of arcs departing in each level is the red solid line, and the number of lower triangles in each level is the green dashed line. In contrast to Figure 2.17 these figures have a logarithmic x-scale.



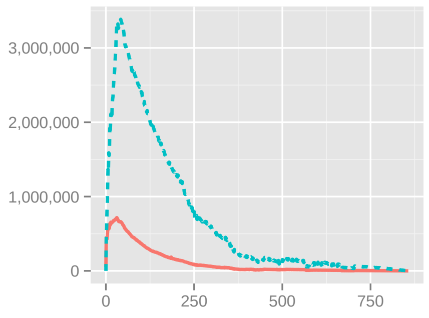
(a) Karlsruhe/KaHIP



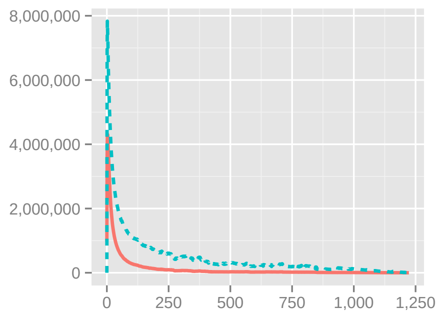
(b) Karlsruhe/Metis



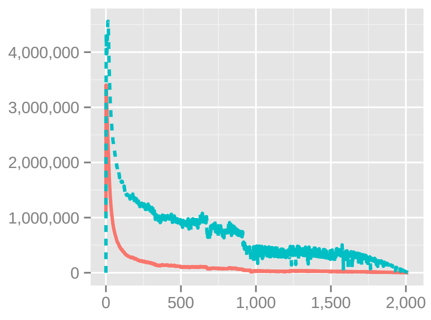
(c) TheFrozenSea/KaHIP



(d) TheFrozenSea/Metis



(e) Europe/KaHIP



(f) Europe/Metis

Figure 2.17: The number of lower triangles (y-axis) per level (x-axis) is the blue dashed line, and the time needed to enumerate all of them per level is the red solid line. The time unit is 100 nanoseconds. If the time curve thus rises to 1 000 000 on the plot the algorithm needs 0.1 seconds. In contrast to Figure 2.16 these figures do not have a logarithmic x-scale.

SSE	Prepro.t.	Threads	Metric Pairs	Karlsruhe		TheFrozenSea		Europe	
				Metis time [s]	KaHIP time [s]	Metis time [s]	KaHIP time [s]	Metis time [s]	KaHIP time [s]
○	○	1	1	0.0567	0.0468	7.88	10.08	21.90	10.88
●	○	1	1	0.0513	0.0427	7.33	9.34	19.91	9.55
●	●	1	1	0.0094	0.0091	3.74	3.75	7.32	3.22
●	●	16	1	0.0034	0.0035	0.45	0.61	1.03	0.74
●	●	16	2	0.0035	0.0033	0.66	0.76	1.34	1.05
●	●	16	4	0.0040	0.0048	1.19	1.50	2.80	1.66

Table 2.19: Basic customization performance. The input graphs are assumed to be directed, i.e., separate upward and downward metrics are used. We show the impact of enabling SSE, precomputing triangles (Prepro.t.), multi-threading, and customizing several metric pairs at once.

2.9.4 Triangle Enumeration

We first evaluate the running time of the adjacency-array-based triangle enumeration algorithm. Figure 2.17 clearly shows that most time is spent enumerating the triangles of the lower levels. This justifies our suggestion to only precompute the triangles for the lower levels as these are the levels were the optimization is most effective. However, precomputing more levels does not hurt if enough memory is available. We propose to determine the threshold level up to which triangles are precomputed based on the size of the available unoccupied memory. On modern server machines, such as our benchmarking machine, there is enough memory to precompute all levels. The memory consumption is summarized in Table 2.18. However, precomputing all triangles is prohibitive in the game scenario as less available memory should be expected.

2.9.5 Customization

In Table 2.19, we report the times needed to compute a customized metric using the basic customization algorithm. A first observation is that on the road graphs, the KaHIP order leads to a faster customization whereas on the game map Metis dominates. Using all optimizations presented, we customize Europe in below one second. When amortized[4], we even achieve 415 ms which is only slightly above the non-amortized 347 ms reported in [54] for CRP. Their experiments were run on a different machine

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

Undir-ected	Metrics	Bits per Metric	SSE	Threads	Prepro all tri.?	Customiz-ation time [s]	Amor-tized time [s]
○	2	32	○	1	○	7.88	7.88
●	1	32	○	1	○	6.65	6.65
●	4	32	○	1	○	9.36	2.34
●	4	32	●	1	○	8.51	2.13
●	8	16	●	1	○	8.52	1.06
●	8	16	●	2	○	5.00	0.63
●	8	16	●	2	●	2.16	0.27
●	8	16	●	16	●	0.63	0.08

Table 2.20: Detailed basic customization performance on TheFrozenSea. We show the impact of exploiting undirectedness, customizing several metrics at once, reducing the bitwidth of the metric, enabling SSE, multi-threading (# Thr.), and precomputing triangles (Pre. trian.). The order in which improvements are investigated is different from Table 2.19. The results are based on the Metis order as Table 2.19 shows that KaHIP is outperformed.

with a faster clock but 2 × 6 instead of 2 × 8 cores, while using a turn-aware data structure, making an exact comparison difficult.

Previous works [77] have tried to accelerate the preprocessing phase of the original two-phase CH to the point that it can be used in a similar scenario as our technique. A fast preprocessing phase can be viewed as form of customization phase. In [77] a sequential preprocessing time of 451 s was reported. This compares best to our 9.5 s sequential customization time. Note that the machine on which the 451 s were measured is slower than our machine. However, the gap in performance is large enough to conclude that we achieve a significant speedup. Furthermore, [2] report a CH preprocessing time of 2 min when parallelized on 12 cores. This compares best against our 415 ms parallelized amortized customization time. While the machine used in [2] is slightly older and slower than our machine and the number of cores differs (12 vs. 16), again, the performance gap is large enough to safely conclude that a significant speedup is present. Besides these differences, both CH preprocessing experiments were only performed for travel time weights. To the best of our knowledge, nobody has been able to match performance achieved for travel time weights for less well-behaved weights, such as travel distance. For example, CH preprocessing times reported in [77] show at least a factor of 2 difference in performance on distance over travel time metric on any of the considered benchmarks. This contrasts with CCH, for which we can prove that basic customization and elimination-tree query performance are completely independent of the metric considered.

Unfortunately, the optimizations illustrated in Table 2.19 are pretty far from what is possible with the hardware normally available in a game scenario. Regular PCs do not have 16 cores and one cannot clutter up the whole RAM with several GB of precomputed triangles. We therefore ran additional experiments with different parameters and report the results in Table 2.20. The experiments show that it is possible to fully customize TheFrozenSea in an amortized[5] time of 1.06s without precomputing triangles or using multiple cores. However a whole second is still too slow to be usable, as graphics, network and game logic also require resources.

We therefore evaluated the time needed by partial updates as described in Section 2.7.7. We report our results in Table 2.21, also for the road networks. The median, average and maximum running times significantly differ. There are a few arcs that trigger a lot of subsequent changes, whereas for most arcs a weight change has nearly no effect. The explanation is that highway arcs and choke point arcs are part of many shortest paths, and thus updating such an arc triggers significantly more changes. On the Europe road network, the maximum observed time for a partial CCH update (81.0 ms) is similiar to CRP (73.77 ms), but the average time is much lower for CCH (0.045 ms) than for CRP (17.94 ms), c.f. [45].

Finally, we report the running times of the perfect customization algorithm in Table 2.22. The required running time is about 3 times the running time needed by the basic customization. Recall that the basic customization in essence enumerates all lower triangles, i.e., it visits every triangle once, while the perfect customization also enumerates all intermediate and upper triangles, i.e., it visits every triangle three times.

2.9.6 Query Performance

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

In Table 2.23, 2.24, and 2.25 we compare query performance. The “D+w” variant uses a metric-dependent order and a non-perfect witness search in the spirit of [77]. The details are described in Section 2.3. The “M-w” and “K-w” variants use a metric-independent order computed by Metis or KaHIP. Only a basic customization was performed, i.e., no witness search was performed. The “M+w” and “K+w” variants use a metric-independent order and combine it with a perfect witness search. We evaluate three query variants. The “basic” variant uses a bidirectional variant of Dijkstra’s algorithm with stopping criterion. The “stalling” variant additionally uses the stall-on-demand optimization as described in Section 2.8.2. Finally, we also evaluate the elimination tree query and refer to it as “tree”. This query requires the existence of an

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

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

Table 2.21: Partial update performance for Metis orders "M" and KaHIP orders "K". We report the running time and number of arcs changed for partial metric updates. We report median, average and maximum over 10 000 runs. In each run we change the upward and the downward weight of a single random arc in G to random values in $[0, 10^5]$. The metric is reset to initial state between runs. Timings are sequential without SSE. No triangles were precomputed.

		Karlsruhe		TheFrozenSea		Europe	
# Threads	Prepro.t.	Metis	KaHIP	Metis	KaHIP	Metis	KaHIP
1	○	0.15	0.13	30.54	33.76	67.01	32.96
16	○	0.03	0.02	3.26	4.37	14.41	5.47
1	●	0.05	0.05	8.95	12.51	23.93	10.75
16	●	0.01	0.01	1.93	2.29	3.50	2.35

Table 2.22: Perfect Customization. We report the time required to turn an initial metric into a perfect metric. Runtime is given in seconds, without use of SSE.

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

Table 2.23: Query performance on the Karlsruhe instance. We use the following abbreviations: “D” refers to a metric dependent order, “M” to a Metis order, “K” to a KaHIP order, “+w” and “-w” indicate whether a witness search is used. Numbers are averaged over 10^6 random uniform queries. The reported node and arc counts refer only to the forward search.

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

Table 2.24: Query performance on the Europe instance. Continuation of Table 2.23.

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

Table 2.25: Query performance on the FrozenSea instance. Continuation of Table 2.23.

elimination tree of low depth and is therefore not available for metric-dependent orders. We ran our experiments on all three of our main benchmark instances. Experiments on additional instances are available in Section 2.11. For both road graphs, we evaluate the travel-time and distance variants. We report the average running time needed to perform a distance query, i.e., we do not unpack the paths. We further report the average number of "visited" vertices in the forward search. For the "basic" and "stalling" queries, these are the vertices removed from the queue. For the "tree" query, we regard every ancestor as "visited". The numbers for the backward search are analogous and therefore not reported. We report the average number of arcs relaxed in forward search of each query variant. Finally, we report the average number of vertices stalled and the average number of arcs that need to be tested in the stalling test. A stalled vertex is not counted as "visited".

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

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

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

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

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

It is well-known that metric-dependent CHs work significantly better with the travel-time metric than with other less well behaved metrics such as the geographic distance. For such metrics, the KaHIP order outperforms the metric-dependent orders. For example the basic query with perfect customization visits less vertices and less arcs. This is very surprising, especially considering, that the metric-dependent orders that we computed are better than those reported in [77], i.e., the gap with respect to the original implementation is even larger. However, combining the stalling query with metric-dependent orders yields the smallest number of visited vertices and relaxed arcs. Unfortunately, combining the stalling query with metric-independent orders does not yield the same benefit and even makes the query running times worse. Fortunately, the metric-independent orders can be combined with the elimination tree query. As result, the fastest variant is the combination of KaHIP order, perfect witness search, and elimination tree query, which is over a factor of two faster than stalling with the metric-dependent order. Interestingly, the latter is even beaten when no perfect witness search is performed, but with a significantly lower margin.

A huge advantage of metric-independent orders compared to metric-dependent orders is that the resulting CH performs equally well regardless of the weights of the input graph. The combination of metric-independent order, elimination tree query and basic customization results in a setup where the order in which the vertices are visited and the order in which the arcs are relaxed during the query execution does not even depend on the weights of the input graph. It is thus impossible to construct a metric, where this setup performs badly. This contrasts with the CH of [77], whose performances varies significantly depending on the input metric.

In Table 2.26, we give a more in-depth experimental analysis of the elimination tree query algorithm without perfect witness search. We break the running times down into the time needed to compute the least common ancestor (LCA), the time needed to reset the tentative distances and the time needed to relax all arcs. We further report the total distance query time, which is in essence the sum of the former three. We additionally report the time needed to unpack the full path. It is therefore not useful to further optimize the LCA computation or to accelerate tentative distance resetting using, e.g., timestamps. We only report path unpacking performance without precomputed lower triangles. Using them would result in a further speedup with a similar speed-memory trade-off as already discussed for customization.

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

Table 2.26: Detailed elimination tree query running time performance without perfect witness search. We use the following abbreviations: “K” for KaHIP order, “M” for Metis order, “TT” for travel time metric, “GD” for geographic distance in a road network, “MD” for map distance on a game map, and “FSea” for TheFrozenSea.

2.9.7 Comparison with Related Work

We conclude our experimental analysis on the DIMACS Europe road network with a final comparison of related techniques, as shown in Table 2.27. For Contraction Hierarchies (CH), we report results based on implementations by [77, 45] and ourselves, covering different trade-offs in terms of preprocessing versus query speed. More precisely, we observe that our own CH implementation (used for detailed analysis and comparison in Section 2.9.1–2.9.6) has slightly slower queries on travel time metric but factor of 2.1 faster queries on distance metric, at the cost of higher preprocessing time. Recall from Section 2.3 that we employ a different vertex priority function and no lazy updates. For Customizable Route Planning (CRP), we report results from [45, 44].

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

In contrast to traditional CH, Customizable Contraction Hierarchies (CCH) by design achieve a performance trade-off with much lower metric-dependent preprocessing

Algorithm	Implementation	Test Prozessor	Metric	Turn-aware	Metric-Dep. Prepro. Time [s]	(# Threads)	Queries Search Space [Nodes]	Running Time [μs]	(# Threads)
CH	[77]	Opt 270	TT	○	1 809	(1)	356	152	(1)
CH	[77]	Opt 270	GD	○	5 723	(1)	1 582	1 940	(1)
CH	[77]	E5-2670	TT	○	1 075.88	(1)	353	91	(1)
CH	[77]	E5-2670	GD	○	3 547.44	(1)	1 714	1 135	(1)
CH	our	E5-2670	TT	○	813.53	(1)	375	107	(1)
CH	our	E5-2670	GD	○	9 390.32	(1)	1422	540	(1)
CH	[45]	X5680	TT	○	109	(12)	280	110	(1)
CH	[45]	X5680	GD	○	726	(12)	858	870	(1)
CRP	[45]	X5680	TT	●	0.37	(12)	2 766	1 650	(1)
CRP	[45]	X5680	GD	●	0.37	(12)	2 942	1 910	(1)
CRP	[44]	i7-920	TT	○	4.7	(4)	3 828	720	(2)
CRP	[44]	i7-920	GD	○	4.7	(4)	4 033	790	(2)
CCH	our	E5-2670	TT	○	0.74	(16)	1 303	413	(1)
CCH	our	E5-2670	GD	○	0.74	(16)	1 303	426	(1)
CCH+a	our	E5-2670	TT	○	0.42	(16)	1 303	416	(1)
CCH+a	our	E5-2670	GD	○	0.42	(16)	1 303	421	(1)
CCH+w	our	E5-2670	TT	○	2.35	(16)	1 303	161	(1)
CCH+w	our	E5-2670	GD	○	2.35	(16)	1 303	214	(1)

Table 2.27: Comparison with related work on the DIMACS Europe instance with travel time (TT) and geographic distance metric (GD). We compare our approaches, CCH, CCH with amortized customization (CCH+a), and CCH with perfect witness search (CCH+w), with different CRP and CH implementations from the literature. We report performance of the metric-*dependent* fraction of overall preprocessing, i.e., vertex ordering and contraction time for CH, customization time for CRP and CCH. We further report average query search space, including stalled vertices for CH (which might not be included in the CH figures taken from [45]). We finally report running time in microseconds. If parallelized, the number of threads used is noted in parenthesis. Since the CH performance in [77] was evaluated on a ten year old machine (AMD Opteron 270), we obtained the source code and re-ran experiments on our hardware (Intel Xeon E5-2670) for better comparability. Also note that the latest CRP implementation by [45], evaluated on an Intel Xeon X5680, is turn-aware (●), i.e., it uses turn tables (set to zero in the reported experiments); We therefore additionally take results from [44] obtained on an Intel Core-i7 920, which uses a turn-unaware implementation but parallelizes queries.

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

The CRP implementation of [45] uses SSE to achieve its customization time of 0.37 s. In a server scenario where customization is run for many users concurrently, e.g., to customize a stream of traffic updates for all active users at once, we propose to amortize triangle enumeration time as described in Section 2.7.6. By using SSE and processing metrics for four users at once, this amortized customization (CCH+a, 0.42 s) can almost close the gap to CRP customization performance. Refer to Table 2.19 for other configurations.

Most interestingly, in terms of query performance on travel distance, CCH outperforms even the best CH result. For even better CCH query performance, we may employ perfect customization and witness search (CCH+w). It increases customization time by factor of 3.2 (enumerating all lower, intermediate and upper triangles), but enables a CCH query variant that, while still visiting all vertices in the elimination tree, needs to consider far fewer arcs (c.f. Table 2.24). Thereby, CCH+w further improves CCH query performance by factor of 1.9 for distance metric and factor of 2.6 for time metric. With 161 μs for travel time, CCH+w query performance is almost as good as the best CH result of 91 μs.

2.10 Further Metric-Independent Ordering Strategies

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

In the context of sparse matrix factorization a common approach is the *minimum degree heuristic* (MinDeg). To the best of our knowledge, the first variant of this ordering heuristic was described in [138], but we refer to [80] for more details. The basic idea is simple: Iteratively contract a vertex with a minimum degree in the core. This approach differs from sorting vertices by degree in the input graph, which does not work well on road networks [46].

A variant of the minimum degree heuristic was proposed in [45]. The idea is also simple: Iteratively contract a vertex that adds the least number of arcs to the chordal supergraph (MinArc), using degree in the core to break ties. However, [45] already observed that MinArc orders can be improved when augmented with partitioning

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

Table 2.28: Time in seconds to compute minimum degree (MinDeg) and minimum shortcut (MinArc) orders.

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

Table 2.29: Further ordering strategies: minimum degree (MinDeg) and minimum shortcut (MinArc).

information from what they refer to as guidance levels. Their reported experimental results are only with respect to these hybrid orders.

We implemented both MinDeg and MinArc in the straightforward way using a priority queue of vertices ordered by the respective weighting function. Table 2.28 shows the resulting order computation times on our three main instances. At least for MinDeg, more sophisticated strategies exist [80] that might be faster. Nonetheless, MinArc is significantly slower than MinDeg because it simulates the contraction of every vertex in the graph, including those that yield a high number of shortcuts but are only contracted in the end, once their degree has already decreased due to the graph shrinking.

More importantly, Table 2.29[6] reports performance indicators for MinDeg and MinArc in comparison to Metis and KaHIP. Recall that the number of triangles determines customization running times, the number of arcs in the CH is proportional to the memory consumption if precomputed triangles are not used, and the number of arcs in the search space gives a good indication of query performance. MinArc nearly always dominates MinDeg with respect to every criterion except computation time. Yet, while both can be computed faster than the KaHIP-based orders, Metis is still fastest. Similarly, both MinArc and MinDeg result in lower memory consumption than KaHIP-based orders, but not than Metis on the TheFrozenSea instance. Upper tree width bound from KaHIP-based orders are consistently better than from MinDeg or MinArc.

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

2.11 Further Instances

2.11.1 OpenStreetMap-based Road Graphs

OpenStreetMap (OSM) is a very popular collaborative effort to create a map of the world. From this huge data source, very large road graphs can be extracted, that are very detailed depending on the exact region considered. Using the data provided by GeoFabrik[7] and the tools provided by OSRM[8], we extracted a road graph of Europe and report its size in Table 2.30. The exact graph is available in DIMACS format on our website[9]. The geographic region corresponding to the graph is depicted in Figure 2.31. As depicted in the figure, our OSM Europe graph is significantly larger than the DIMACS Europe graph. Our OSM Europe graph also contains Eastern Europe and Turkey. The graph's east border ends at the east border of Turkey and then goes upward cutting through Russia. On the other hand, the DIMACS Europe graph stops at the German-Polish border.

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

[6]The KaHIP and Metis numbers slightly differ from those in Table 2.12, where they were only sampled over 1000 random search spaces.

[7]`http://download.geofabrik.de/`

[8]`http://project-osrm.org/`

[9]`http://i11www.iti.kit.edu/resources/roadgraphs.php`

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

Table 2.30: Size of DIMACS Europe compared to OSM Europe.

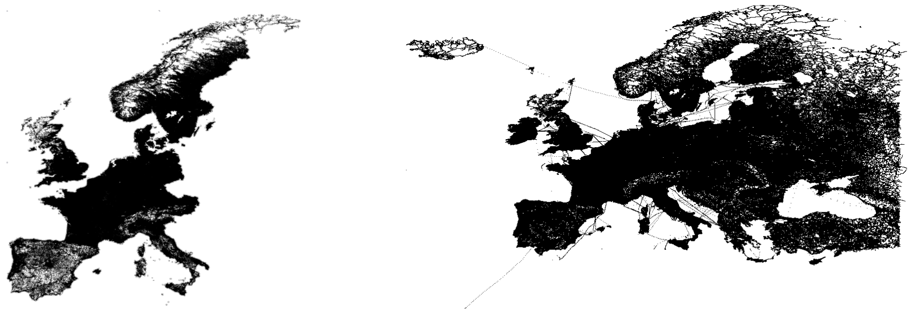


Figure 2.31: Comparison of Europe instances from DIMACS and OSM.

and thus have fewer degree-2 vertices. Accelerating shortest path computations on graphs with a huge number of vertices of degree 1 or 2 is significantly easier relative to the graph size. One reason is that Dijkstra's algorithm cannot exploit the abundance of low-degree vertices. Dijkstra's algorithm with stopping criterion needs on average 27 s for a st-query with s and t picked uniformly at random on the OSM-Europe graph. This contrasts with the DIMACS Europe graph, where only 1.6 s are needed. A slower baseline obviously leads to larger speedups. Table 2.30 shows that the difference between the two Europe graphs in terms of vertex count is significantly smaller, when discarding degree-1 and degree-2 vertices. In fact, relative to their geographical region's area, the two graphs seem to be approximately comparable in size.

We computed contraction orders for OSM-Europe. The sizes of the resulting CHs are reported in Table 2.32. These sizes can be compared with the "undirected" numbers of Table 2.12. We did not perform experiments with a perfect witness search. Metis ordered the vertices within 29 minutes, whereas the KaHIP-based ordering algorithm needed slightly less than 3 weeks. However, as already discussed in detail, we did not optimize the latter for speed and therefore one must *not* conclude from this experiment that KaHIP is slow. The CHs for OSM-based graphs are significantly larger. The DIMACS-Europe CH only contains 70M arcs for Metis whereas the OSM-Europe CH contains 400M arcs for Metis. However, this is due to the huge amount of low-degree vertices in the input. On the DIMACS graph the size increase compared to the number of input arcs is 70 M/42 M = 1.67 whereas for the OSM-based graph the size increase is only 400 M/348 M = 1.15. This effect can be explained by considering what happens when contracting a graph consisting of a single path. In a path graph every vertex, except the endpoints, has 2 outgoing arcs. There is one arc in each direction. As long as the endpoints are contracted last, every vertex, except the endpoints, in the resulting CH search graph also have degree 2. There is thus no size increase. As the OSM-based graph has many degree-2 vertices, this effect dominates and explains the comparatively small size increase. The search space sizes are nearly identically. For example the KaHIP search space contains 117K arcs for the DIMACS Europe graph, whereas it contains 119K arcs for the OSM Europe graph. This effect is explained by the fact, that both data sources correspond to almost the same geographical region. The mountains and rivers are thus in the same locations and the number of roads through these geographic obstacles are the same in both graphs, i.e., both graphs have very similar recursive separators. The small size increase is explained by the fact that the OSM-based graph also includes Eastern Europe. The customization times are reported in Table 2.33. As the OSM-based graph has more arcs, the customization times are higher on that graph. On the DIMACS Europe graph 0.61 s are needed whereas 1.7 s are needed on OSM Europe for the KaHIP order and 16 threads, which is a surprisingly small gap considering the differences in input sizes. Eliminating the degree-2 vertices from the input should further narrow this gap. As the search space sizes are very similar, it is not surprising that the query running times reported in Table 2.34 are nearly identical.

2.11.2 Further DIMACS-Instances

During the DIMACS challenge on shortest path [55], several benchmark instances were made available. Among them is the Europe instance used throughout our in-depth experiments in previous chapters. Besides this instance, also a set of graphs

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

Table 2.32: CCH sizes for OSM Europe. The search space sizes were obtained by randomly sampling 10 000 vertices uniformly at random.

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

Table 2.33: CCH Customization performance on OSM Europe. We vary the number of threads and whether precomputed triangles are used. SSE is enabled, running times are non-amortized and no perfect customization was performed.

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

Table 2.34: CCH Elimination Tree Query performance on OSM Europe, averaged over 10 000 random st-pairs chosen uniformly at random.

representing the road network of the USA was published. In Table 2.35, we report experiments for these additional DIMACS road graphs. Besides CCH performance, we also report the running time needed by Dijkstra's algorithm. CCH queries use the elimination-tree query algorithm. Other than the DIMACS-Europe instance, these USA instances originate from the U.S. Census Bureau. Note that the USA instances have some known data quality issues: The graphs are generally undirected (no one-way streets) and highways are sometimes not connected at state borders. Furthermore, the arcs are not directed, i.e., there are no one-way streets and the weight of an arc corresponds to its backward arc's weight. In the "Uni" configuration, our algorithm exploits that the arcs are undirected and only stores one weight per CCH arc. In the "Bi" configuration, which is the one we use for the Europe graphs, our algorithm stores two weights per CCH arc. However, on these instances both weights will be identical for every arc. The DIMACS-Europe comes from another data source and

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

Table 2.35: Instance sizes and experimental results for the additional DIMACS road graphs graphs with basic, non-amortized customization averaged over 10 000 random uniform queries. The instances are weighted by travel time.

does not have these limitations. This is the reason why we focus on the Europe instance in the main part of our evaluation.

Fortunately, these graphs are undirected. We can therefore evaluate the impact that using a single undirected metric has on the customization running times. We compare it against the performance of a setup with with two directed metrics, which is needed with directed input graphs. Experiments using a single metric are marked with “Uni” in the table, whereas the experiments with two metrics are marked with “Bi”. The query running times are very similar. This is not surprising as the number of relaxed arcs does not depend on whether one or two weights are used. For larger graphs there is a slightly larger difference in running times. We believe that this is a cache effect. As the “Bi” variant has twice as many weights, less arcs fit into the L3 cache. For the smaller graphs this effect does not occur because the higher CH levels occupy less memory than the cache’s size and thus doubling the memory consumption is non-problematic.

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

Graph	Nodes $[\cdot 10^3]$	Edges $[\cdot 10^3]$	CCH Arcs $[\cdot 10^3]$	Metis [s]	Custom. [ms] 1thr.	16thr.	Dijkstra [μs]	CCH Query [μs]
16room_005	231	838	2 959	1.196	359	41	10 913	15
AcrossCape	392	1 534	12 632	2.452	4 789	563	23 609	239
blastedlands	131	507	3 740	0.896	1 250	144	6 075	304
maze512-4-3	209	686	1 641	0.996	138	35	7 810	6
ost100d	137	531	3 722	0.880	1 076	124	5 607	116
rand512-35-8	161	421	1 805	0.988	469	39	7 422	223
rand512-40-8	114	280	797	0.684	115	16	4 856	41

Table 2.36: Instance sizes and experimental results for the additional game-based graphs. We report the number of vertices, undirected edges, arcs in the CH search graph, the running time Metis needed to compute the order, the time needed to do a full non-amortized customization with 1 and 16 threads using an undirected weight, the average running time of Dijkstra's algorithm with stopping criterion and the average running time of an elimination-tree distance query. We averaged over 10 000 queries where s and t were picked randomly at uniform.

2.11.3 Further Game Instances

Besides our main game benchmark instance TheFrozenSea, the benchmark data set of [135] contains a large variety of different game maps. To demonstrate that our technique also works on other game maps we ran our experiments on a selection of different graphs from the set. "16room_005" is a synthetic map with many rooms in grid shape that are connected through small doors. "AcrosstheCape" is another Star Craft map that is sometimes used as benchmark instance. "blastedlands" originates from WarCraft 3 and is the largest map in that set in terms of vertices. "maze512-4-3" is a synthetic map that consists of a random maze with corridors that are 4 fields wide. "ost100d" is the largest map from the Dragon Age Origins map set. "random512-35-8" and "random512-40-8" are synthetic maps that contain random obstacles. The difference between them is the amount of space covered by obstacles. The website [136] from which the data originates includes pictures depicting each instance. In Table 2.36, we report experiments on these graphs. All experiments were run using a single undirected metric with 32bits per weight. The customization running times are non-amortized. We did not perform experiments with a perfect witness search.

All additional game-based instances have fewer vertices than TheFrozenSea. Further, the CH query is the slowest on TheFrozenSea with 316 μs. Interestingly, a full customization is slower on AcrosstheCape than on TheFrozenSea by about a factor of 2. This is most likely due to slight differences in the structures of the maps.

However, we believe that it is safe to conclude from the experiments that our technique works across a wide range of maps.

2.12 Chapter Conclusion

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

3 FlowCutter

In this chapter, we introduce FlowCutter, a novel algorithm to compute a set of edge cuts or node separators that optimize cut size and balance in the Pareto-sense. Our core algorithm heuristically solves the balanced connected st-edge-cut problem, where two given nodes s and t must be separated by removing edges to obtain two connected parts. Using the core algorithm as subroutine, we build variants that compute node separators which are independent of s and t. From the computed Pareto-set, we can identify cuts with a particularly good trade-off between cut size and balance that can be used to compute contraction orders, which can be used in the Customizable Contraction Hierarchy algorithm described in the previous chapter. Our core algorithm runs in $O(c|E|)$ time where E is the set of edges and c is the size of the largest outputted cut. This makes it well-suited for separating large graphs with small cuts, such as road graphs, which is the primary application motivating our research. For road graphs, we present an extensive experimental study demonstrating that FlowCutter outperforms the current state-of-the-art both in terms of cut sizes and CCH performance. By evaluating FlowCutter on a standard graph partitioning benchmark, we further show that FlowCutter also finds small, balanced cuts on non-road graphs. Another application is the computation of small tree-decompositions. To evaluate the quality of our algorithm in this context, we entered the PACE 2016 challenge [39] and won the first place in the corresponding sequential competition track. We can therefore conlude that our FlowCutter algorithm finds small, balanced cuts on a wide variety of graphs.

Our work was presented at the ALENEX conference [86]. A preliminary ArXiv version is available [85]. This chapter is based on a submitted but not yet accepted extended journal version. This chapter is joint work with Michael Hamman. The source code of the PACE 2016 submission is available at [128].

3.1 Introduction

A graph cut is a set of edges, whose removal separates a graph into two sides. Similarly, a node separator is a set of nodes whose removal separates a graph into two sides. A cut or separator is balanced if the number of nodes in both sides is roughly the same. Balanced graph bisection is the problem of finding a balanced cut or separator. This is a fundamental and NP-hard [74] graph problem that has received a lot of attention [92, 48, 117, 5, 120] and has many applications. We present FlowCutter, a novel algorithm to compute edge cuts and node separators. It computes a set of cuts or separators that trade-off cut respectively separator size for imbalance in the Pareto-sense. For edge cuts, FlowCutter guarantees that the two sides of the cut are connected subgraphs.

Outline. Section 3.2 presents an overview over related work and the core ideas of the shortest path application driving our research as well as some other applications including tree-decompositons. Section 3.3 presents our notation. Section 3.4 introduces the core FlowCutter *st*-bisection algorithm. Section 3.5 extends the core algorithm to general bisection, node bisection, and describes how to compute CCH contraction orders. Section 3.6 presents an extensive experimental evaluation on road graphs against the current state of the art.

3.2 Applications and Related Work

We start by presenting a very high level overview of some of the core ideas employed to accelerate shortest path computations. The Customizable Contraction Hierarchy algorithm as described in Chapter 2 is one way to use these ideas. Fortunately, other techniques work similarly and thus FlowCutter is more broadly applicable and not limited to CCH. Our experimental evaluation will, however, focus on the perfmance in a CCH context.

In many shortest path acceleration techniques, the preprocessing phase involves computing balanced graph edge cuts or node separators. The central idea can be formulated in terms of edge cuts as well as node separators. In this section, we present the node separators variant as CCH uses this variant. The idea can be described as follows: Given a graph G and a node separator S, the algorithms precompute for every node in the graph how to get to every node in S. Further, they precompute the shortest paths among all nodes of S. Consider a query that asks to compute a shortest path from a node s to a node t. Either s and t are on the same or on opposite sides of S. If they are on opposite sides, a shortest path can be assembled by iterating over all nodes v in S and combining the precomputed paths from s to v and from v to t and picking the shortest path. The running time of a distance query in this case is thus in $O(|S|)$, which is assumed to be small for road graphs. However, if s and t are on the same side then a graph search is necessary using, for example, Dijkstra's algorithm. On the side of s and t the search is unrestricted. However, it does not cross S and instead makes use of the shortest paths precomputed between the nodes of S. If the sides are of the same size and s and t are chosen uniformly at random then there is a 50% probability that they are on opposite sides. Half of the queries can therefore be answered quickly. For the other half of queries, this approach restricts the search to half of the graph. However, as half of a continent is still large, one usually applies this idea recursively.

The effectiveness of these techniques crucially depends on the size of the separators found. The balance is less important. Only a slight balance is necessary to assure that the recursion has a logarithmic depth. This application does, however, not benefit from a perfect balance. In practice, the contrary is true: Requiring perfect balance results in many small, slightly imbalanced separators not being found. The perfectly balanced separators found can be therefore larger. This larger size is detrimental to the running time of the query phase, compared to using the smaller slightly imbalanced separators. Fortunately, road graphs have small separators and cuts because of geographical features such as rivers or mountains. Previous work has coined the term *natural cuts* for this phenomenon [48]. However, identifying these natural cuts is a difficult problem.

Graph partitioning software used for road graphs include *KaHip* [117], *Metis* [92], *InertialFlow* [120], and *PUNCH* [48]. We experimentally compare FlowCutter with the first three. As we unfortunately have no implementation of PUNCH, we omitted an experimental comparison with it. All of these works formalize the graph bisection problem as a bicriteria problem optimizing the cut size and the imbalance. The *imbalance* measures how much the sizes of both sides differ and is small if the sides are balanced. The standard approach is to bound the imbalance and minimize the cut size. However, this approach has several shortcomings. Consider a graph with a million nodes and set the maximum imbalance to 1%. Suppose an algorithm finds a cut C_1 with 180 edges and 0.9% imbalance. This is all the information of the cut's quality that is provided. Can you decide solely based on this information, whether this is a good cut? It seems good as 180 is small compared to the node count. However, we would come to a different conclusion, if we knew that a cut C_2 with 90 edges and 1.1% imbalance existed. In our application — shortest paths — moving a few nodes to the other side of a cut is no problem. However, halving the cut size has a huge impact. The cut C_2 is thus clearly superior. Further, assume that a third cut C_3 with 180 edges and 0.7% imbalance existed. C_3 dominates C_1 in both criteria. However, both are equivalent with respect to the standard problem formulation and thus a programm is not required to output C_3 instead of C_1. To overcome these problems, our approach computes a set of cuts that optimize cut size and imbalance in the Pareto sense, i.e., it tries to compute solutions that are Pareto-optimal. As this problem is NP-hard, one cannot expect to always succeed perfectly. A further significant shortcoming of the state-of-the-art partitioners, with the exception of InertialFlow, is that they were designed for small imbalances. Common benchmarks, such as the one maintained by Chris Walshaw [124], only include test cases with imbalances up to 5%. However, for our application imbalances of 50% can be fine. For such high imbalances unexpected things happen with the standard software, such as increasing the allowed imbalance can increase the achieved cut sizes. Indeed, KaHip, one of the competitors, has been updated, as reaction to the conference version of our work [86], to overcome some of these shortcomings. The newer version produces better results for higher imbalances than the old version.

Other Applications. A vast amount of algorithms for NP-hard graph problems exist that are fixed-parameter tractable in the tree-width of a graph [38, 26]. It is therefore an interesting question whether algorithms being able to compute good tree-decompositions in practice leads to practicable variants of these algorithms. To investigate this question, the PACE competition [39] was held in 2016 at IPEC, a conference with a focus on fixed-parameter tractable algorithms. The objective of two competition tracks was to compute a small tree decomposition within a limited time frame. The tracks differed in whether parallelism was allowed or not. To evaluate the performance of FlowCutter in this context, we submitted our algorithm. Our implementation runs FlowCutter iteratively with varying parameters until the time limit is reached. The parallel version runs several FlowCutter instances in parallel. The code submitted to the PACE challenge is open source and available at [128]. In the sequential track our implementation won the first place out of six submissions and in the parallel track it won the second place out of 3 submissions. Given these results, it is safe to say, that our algorithm is at least highly competitive, if not the state-of-the-art, in terms of computing tree decompositions in practice.

The contraction orders used by CCH, which as based upon nested dissection [78, 99], are also called minimum fill-in orders in the context of sparse matrices. This establishes a connection to the theory of quickly solving sparse systems of linear equations. Indeed, METIS was developed with this application in mind [92]. METIS was not developed to bisect road graphs. The fact that we use METIS in the context of road graphs is therefore an example of this theoretical connection being exploited in practice. Using the same connection, it is also possible to use FlowCutter to solve sparse system of linear equations. However, even though these two applications are so closely related, the precise trade-off between the various optimization criteria differs. For example in the context of sparse equation systems, cut size is less important than in the road setting whereas having a small bisection algorithm running time is more important.

Another application is information propagation in belief networks [90]. In this setting, a set of random variables is given. It is known how these random variables depend on each other and their interactions are modeled as a graph whose nodes are the random variables. The question is how the distributions change throughout the graph if the distribution of a subset of the variables changes, i.e., some but not all variables are measured. To solve this problem, so called junction-trees are employed. Junction-trees are essentially another name for tree-decompositions. As we can use FlowCutter to compute tree-decompositions, we can also use it to compute junction-trees.

3.3 Preliminaries

A *directed graph* is denoted by $G = (V, A)$ with *node set* V and *arc set* $A \subseteq V \times V$. Similarly, an *undirected* graph is denoted by $G = (V, E)$ with node set V and *edge set*

$E \subseteq 2^V$. Arcs have an implicit direction, whereas edges are undirected. A directed graph is *symmetric*, if for every arc (y,x) there exists an arc (x,y). In a slight abuse of notation, we do not discern between undirected and directed, symmetric graphs. We identify an edge $\{x,y\}$ with the corresponding pair of arcs (x,y) and (y,x). We set $n := |V|$ and $m := |A|$. As input, we only consider undirected graphs without multi-edges and without reflexive loops, i.e., without arcs of the form (x,x). Road graphs that do not fit this description are modified by removing multi-edges, removing reflexive loops, and adding backarcs in the case of one-way streets. In intermediate steps of our algorithm, we also consider non-symmetric directed graphs. The *out-degree* $d_o(x)$ of a node x is the number of outgoing arcs. Similarly, the *in-degree* $d_i(x)$ is the number of incoming arcs. In symmetric graphs, we refer to the value as *degree* $d(x)$ of x, as $d_i(x) = d_o(x)$. A *degree-2-chain* is a sequence of adjacent nodes $x, y_1 \ldots y_k, z$ in a symmetric graph such that $k \geq 1$, $d(x) \neq 2$, $d(z) \neq 2$, and $\forall i : d(y_i) = 2$. An *xy-path* P is a list $(x,p_1),(p_1,p_2)\ldots(p_i,y)$ of adjacent arcs and i is P's length. The *distance* $\text{dist}(x,y)$ is defined as the minimum length over all xy-paths.

3.3.1 Cuts and Separators

A *cut* (V_1,V_2) is a partition of V into two disjoint sets V_1 and V_2 such that $V = V_1 \cup V_2$. An arc (x,y) with $x \in V_1$ and $y \in V_2$ is called *cut-arc*. In another slight abuse of notation, we do not discern between the node partition and the set of cut-arcs. The *size of a cut* is the number of cut-arcs. A min-cut is a cut of minimum size. A *separator* (V_1,V_2,Q) is a partition of V into three disjoint sets V_1, V_2 and Q such that $V = V_1 \cup V_2 \cup Q$. There must be no arc between V_1 and V_2. The cardinality of Q is the *separator's size*. The *imbalance* ϵ of a cut or separator is defined as the smallest number such that $\max\{|V_1|,|V_2|\} \leq \lceil (1+\epsilon)n/2 \rceil$. The imbalance of a separator is defined analogously. For edge cuts $0 \leq \epsilon \leq 1$ holds. This is not necessarily the case for node separators. The separator itself may contain nodes, making it possible that the minimum ϵ is smaller than 0, as both sides can have fewer than $n/2$ nodes. An *ST-cut/separator* is a cut/separator between two disjoint node sets S and T such that $S \subseteq V_1$ and $T \subseteq V_2$. If $S = \{s\}$ and $T = \{t\}$, we write *st-cut/separator*. The *expansion* of a cut/separator is the cut's size divided by $\min\{|V_1|,|V_2|\}$.

Pareto-Optimization and NP-hardness. Computing cuts (and separators) is inherently a bicriteria problem: We want to minimize the cut size and minimize the imbalance. A cut C_1 dominates a cut C_2 if C_1 is strictly better with respect to one criterion and no worse with respect to the other criterion. A cut that is not dominated by any other cut is *Pareto-optimal*. We refer to the pair of imbalance and cut size of a Pareto-optimal cut as *Pareto-trade-off*. It is possible that several Pareto-optimal cuts exist with the same trade-off. The problem we consider asks to compute one cut for every Pareto-trade-off. If there are several, then the algorithm is free to pick any one of them.

This is a departure from existing experimental papers [92, 124, 117, 5, 43, 141, 120] that consider the problem of finding a smallest cut subject to an imbalance bounded by an input parameter. Given a cut for every Pareto-trade-off, it is easy to find a smallest cut with a bounded imbalance. However, a cut with minimum size with an imbalance bounded by an input parameter is not necessarily Pareto-optimal: It is possible that a more balanced cut with the same size exists. Our problem setting is therefore a strict generalization of the problem setting considered in previous works.

The minimum cut problem disregarding the imbalance is polynomially solvable [70]. However, nearly all cut-problems that combine optimizing imbalance and cut size are NP-hard. Examples include:

- Finding a perfectly balanced minimum cut, i.e., one with $\epsilon = 0$, is NP-hard [74].
- A *sparsest cut* C is a cut that minimizes $\frac{|C|}{|V_1|\cdot|V_2|}$. A sparsest cut is Pareto-optimal. Finding a sparsest cut is NP-hard [98].
- Even computing, for a fixed st-pair, a most balanced cut among all st-cuts of minimum size is already NP-hard [30].
- In [139], it was shown that computing a minimum cut that respects a given imbalance is NP-hard.

Being able to compute a cut for every Pareto-trade-off efficiently would yield an efficient algorithm for all these NP-hard problems. Unless P=NP, we can therefore not hope to find an efficient algorithm that provably computes an optimal cut for every Pareto-trade-off. Our algorithm tries to heuristically compute in a single run a cut for every Pareto-trade-off.

3.3.2 Flows

In this chapter, we only consider *unit flows*. These are a restricted variant of the flow problem: Every arc has capacity 1 and an integral flow intensity of either 0 or 1. Formally, a flow is a function $f : A \to \{0,1\}$. An arc a with $f(a) = 1$ is *saturated*. Denote by the *surplus of a node x*. A flow is valid with respect to a source set S and target set T if and only if:

- Flow may be created at sources, i.e., $\forall s \in S : p(s) \geq 0$,
- flow may be removed at targets, i.e., $\forall t \in T : p(t) \leq 0$,
- flow is conserved at all other nodes, i.e., $\forall x \in V\backslash(S \cup T) : p(x) = 0$,
- and flow does not flow in both directions, i.e., for all $(x,y) \in A$ such that $(y,x) \in A$ exists, it holds that $f(x,y) = 0 \lor f(y,x) = 0$.

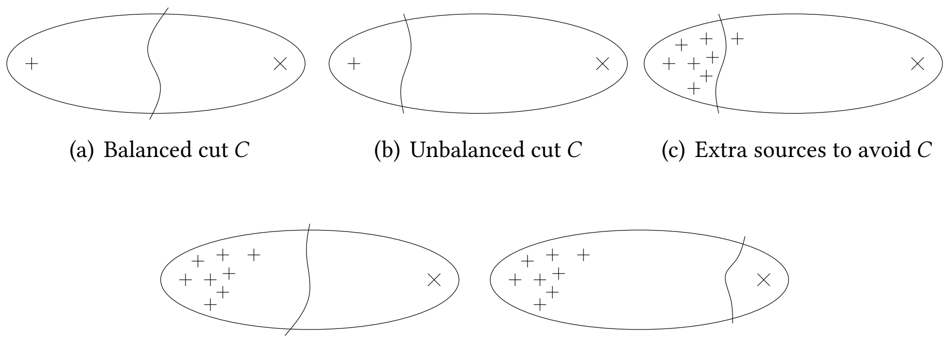
The flow *intensity* is defined as the sum over all $f(x,y)$ for arcs (x,y) with $x \in S$ and $y \notin S$. In other works, the flow intensity is sometimes also called flow value. A path $a_1, a_2 \ldots, a_i$ is *saturated* if there exists an i with $f(a_i) = 1$. A node x is *source-reachable* if a non-saturated sx-path exists with $s \in S$. Similarly, a node x is called *target-reachable* if a non-saturated xt-path exists with $t \in T$. We denote by S_R the set of all *source-reachable nodes* and by T_R the set of all *target reachable nodes*. In [70] it was shown that a flow is maximum, if and only if no non-saturated st-path with $s \in S$ and $t \in T$ exists. If such a path exists, then it is called *augmenting path*. The classic approach to compute max-flows consists of iteratively searching for augmenting paths. Our algorithm builds upon this classic approach. The minimum ST-cut size corresponds to the maximum ST-flow intensity. We define the *source-side cut* as $(S_R, V \backslash S_R)$ and the *target-side cut* as $(T_R, V \backslash T_R)$. Note that in general max-flows and min-cuts are not unique. However, the source-side and target-side cuts are. The source-side and target-side cuts are the same for every max-flow.

3.4 Core FlowCutter Algorithm

In the previous two sections, we described how finding good graph cuts and separators is beneficial to many applications. In this section, we propose our novel algorithm to compute graph cuts, named FlowCutter.

FlowCutter works by computing a sequence of st-min-cuts of increasing size. The more imbalanced cuts are computed first and are followed by more balanced ones. The cuts in this sequence form, after removing dominated ones, the heuristically approached Pareto-set. During its execution our algorithm maintains a maximum flow. With respect to this flow there is a source-side cut C_S and a target-side cut C_T. Our algorithm picks one of the two as the next cut C that it inserts into the set. After choosing C it modifies the set of source and target nodes and potentially augments the maintained flow. This results in a new pair of source-side and target-side cuts. FlowCutter picks C_S as C if there are less or equally many nodes on the source side of C_S than there are on the target side of C_T.

Consider the situation depicted in Figure 3.1. Initially s is the only source node and t is the only target node. Our algorithm starts by computing a maximum st-flow. If we are lucky and the cut C is perfectly balanced as in Figure 3.1(a) then our algorithm is finished. However, most of the time we are unlucky and we either have the situation depicted in Figure 3.1(b) where the source's side of C is too small or the analogous situation where the target's side of C is too small. Assume without loss of generality that the source's side is too small. Our algorithm now transforms non-source nodes into additional source nodes to invalidate C and computes a new more balanced st-min-cut C', the second cut in the sequence. To invalidate C, our algorithm does two things: It marks all nodes on the source's side of C as source nodes and marks one



(a) Balanced cut C (b) Unbalanced cut C (c) Extra sources to avoid C

(d) Source side cut C' (e) Target side cut C'

Figure 3.1: An ellipse represents a graph and the curved lines are cuts. The "+"-signs represent source nodes and "×"-signs represent target nodes.

node as source node on the target's side of C that is incident to a cut edge. This node on the target's side is called the *piercing node* and the corresponding cut arc is called *piercing arc*. The situation is illustrated in Figure 3.1(c). All nodes on the source's side are marked as source node to assure that C' does not cut through the source's side. The piercing node is necessary to assure that $C' \neq C$. Choosing a good piercing arc is crucial for good quality. In this section, we assume that we have a *piercing oracle* that determines the piercing arc given C in time linear in the size of C. In Section 3.4.2, we describe heuristics to implement such a piercing oracle. For the algorithm to make progress we need that C' is non-dominated. As its size is at least the size of C, this is equivalent with C' being more balanced than C. However, we can only guarantee this if C' is, just as C, a source-side cut as in Figure 3.1(d). If C' is a target-side cut as in Figure 3.1(e) then C' might have a worse balance than C. Luckily, as our algorithm progresses, either the target side will catch up with the balance of the source side or another source side cut is found. In both cases our algorithm eventually finds a cut with a better balance than C.

Our algorithms grows the sides around the source and target nodes. By doing so it can guarantee that both sides are connected. In some applications, this is a desired property. In others, it might be an obstacle to finding the smallest possible cuts. Depending on the application this property is therefore either a feature or a drawback of our algorithm.

Our algorithm computes the st-min-cuts by finding max-flows and using the max-flow-min-cut duality [70]. It assigns unit capacities to every edge and compute the flow by successively searching for augmenting paths. A core observation of our algorithm is that turning nodes into sources or targets never invalidates the flow. It is only possible that new augmenting paths are created increasing the maximum flow intensity. Given

```
1  $S \leftarrow \{s\}; T \leftarrow \{t\}$;
2  $S_R \leftarrow S; T_R \leftarrow T$;
3  forward-grow $S_R$; backward-grow $T_R$;
4  while $S \cap T = \emptyset$ do
5      if $S_R \cap T_R \neq \emptyset$ then
6          augment flow by one;
7          $S_R \leftarrow S; T_R \leftarrow T$;
8          forward-grow $S_R$; backward-grow $T_R$;
9      else
10         if $|S_R| \leq |T_R|$ then
11             forward-grow $S$;
               // now $S = S_R$
12             output source side cut edges;
13             $x \leftarrow$ pierce node;
14             $S \leftarrow S \cup \{x\}; S_R \leftarrow S_R \cup \{x\}$;
15             forward-grow $S_R$;
16         else
               // Analogous for target side
```

Algorithm 3.2: Pseudo-Code illustrating the core *st*-bisection algorithm.

a set of nodes X we say that *forward growing* X consists of adding all nodes y to X for which a node $x \in X$ and a non-saturated xy-path exist. Analogously, *backward growing* X consists of adding all nodes y for which a non-saturated yx-path exists. The growing operations are implemented using a graph traversal algorithm (such as a DFS or BFS) that only follows non-saturated arcs. The algorithm maintains besides the flow values four node sets: the set of sources S, the set of targets T, the set source-reachable nodes S_R, and the set of target-reachable nodes T_R. An augmenting path exists if and only if $S_R \cap T_R \neq \emptyset$. Initially, we set $S = \{s\}$ and $T = \{t\}$. Our algorithm works in rounds. In every round it tests whether an augmenting path exists. If one exists, the flow is augmented and S_R and T_R are recomputed. If no augmenting path exists, then it must enlarge either S or T. This operation also yields the next cut. It then selects a piercing arc and grows S_R and T_R accordingly. The pseudo-code is presented as Algorithm 3.2.

3.4.1 Running Time.

Assuming a piercing oracle with a running time linear in the current cut size, we can show that the algorithm has a running time in $O(cm)$ where c is the size of the most balanced cut found and m is the number of edges in the graph. The exact details are slightly more involved but, fortunately, the core argument is simple. All sets only grow unless we find an augmenting path. As each node can only be added once to each set, the running time between finding two augmenting paths is linear. In total, we find c augmenting paths. The total running time is thus in $O(cm)$. The remainder of this section contains the details necessary to formally show the $O(cm)$ worst case running time.

The lines 1-3 of Algorithm 3.2, which initialize the data structures, have a running time in $O(m)$ and are therefore unproblematic. The condition in line 4 can be implemented in $O(1)$ as follows: S and T only grow. Using two bit-arrays with n elements we can store which node is in S and which in T. When adding a node we raise the corresponding bit and check whether the bit in the array is set. As S and T only grow, the loop will abort the next time line 4 is reached, once there is one node for which both bits are set.

We can use a similar structure for the test between S_R and T_R in line 5. S_R and T_R only grow as long as the true-branch in lines 6-8 is not executed. Outside of the true-branch we can therefore use the same bit-vector trick to achieve an $O(1)$-test in line 5. The lines 6-8 consist of the code that augments the flow, i.e., they have a running time of $O(m)$ each time that the branch is executed. In $O(m)$ running time we can reset the bit-arrays, i.e., entering the true-branch is unproblematic for the running time of the test in line 5. We can therefore account for the running time needed to manage the bit-arrays in the lines 6-8 and have an $O(1)$-test in line 5.

As already stated, the lines 6-8 augment the flow and need $O(m)$ running time each time that they are executed. Fortunately, there can be at most c path augmentations. The total time spent in the lines 6-8 over the algorithm's execution is therefore in $O(cm)$.

In addition to maintaining the bit-arrays for S_R and T_R, we can keep track of the number of elements in the sets. This allows us to implement the test in line 10 in $O(1)$.

Showing that the algorithm spends no more than $O(cm)$ running time in the lines 11-15 and in the analogous lines 16-17 is the tricky part of the algorithm's analysis. The lines 16-17 follow directly by symmetry and therefore we focus on the lines 11-15.

We will first establish that the lines 10-17 are only executed at most m times. In each iteration an arc is chosen as piercing arc. After being chosen, an arc cannot participate in another cut and can therefore not be chosen a second time as piercing arc. As there are only m arcs, the number of iterations is bounded by m.

For each of S, T, S_R, and T_R we maintain the data structures of a breath-first search[1], i.e., a queue and a bit-array of n elements. Growing a set as seen in the lines 11 and 15 consists of removing nodes from the corresponding queue and visiting neighboring nodes until the queue is emptied, i.e., executing the regular breath-first search algorithm. Adding a node to the set as seen in line 14 consists of adding the node to queue and raising the corresponding in the bit-array. It is thus clear the operation in line 14 is in $O(1)$ and as there are at most m iterations, the total time spent in line 14 is in $O(m)$

[1]A depth-first search would work too.

which is below the claimed running time of $O(cm)$. The growing of the sets S and T is also in $O(m)$ as we never remove an element from S nor T and they therefore consist of standard breath-first searches. These searches are interrupted from time to time but this does not change the fact that the total running time spent in them is in $O(m)$. Analyzing the running time required to grow the sets S_R and T_R is more difficult as the states of the associated searches can be reset in line 7. Fortunately, as we have already established, line 7 can only be executed at most c times. There are therefore only $O(c)$ state resets. Between two resets the search consists of a normal breath-first-search with a running time in $O(m)$. The total running time is therefore bounded by $O(cm)$.

We assumed that the piercing oracle requires a running time proportional to the number of arcs in the cut from which it must chose. The number of cut arcs never decreases. Further, there are c cut arcs at the end. We therefore know that c is an upper bound to the size of every intermediate cut. Further, as there are at most m iterations, we have bounded the total running time in line 13 by $O(cm)$.

It remains to show that line 12, which outputs the cuts, does not require more than $O(cm)$ running time. This seems trivial at first but the details are significantly more involved than one would naively expect. Following the argumentation for line 13, we know that the operation must run in $O(c)$ running time to achieve a total running time of $O(cm)$. The algorithm must therefore output the cuts as list of cut arcs and not as bit-array that maps each node to a side, as is often done in competitor algorithms. Outputting bit-arrays would be too slow. Another problem consists of identifying the cut-arcs efficiently. In $O(c)$ running time, the algorithm cannot iterate over all nodes in S or T to determine all outgoing arcs, which is needed to find the cut-arcs in the straight-forward way. The trick to achieve the required running time consists of maintaining two lists of saturated arcs. The first list consists of saturated arcs that depart in S and could be part of the cut. The second list consists of saturated arcs that enter T and works analogously. If the algorithm encounters a saturated arc when growing S in line 11, it adds the arc to the list of S. It does this regardless of whether the arc is a cut arc. When reaching line 12, this list contains all cut arcs but also possibly additional saturated arcs that are not part of the cut. The algorithm therefore iterates over all arcs before outputting them and removes those that are no cut arcs. This step can have a running time larger than $O(c)$. Fortunately, as every arc can only be added once to the list, it can also only be removed once. The total running time needed for the removal is therefore in $O(m)$ and we do not need to account for it in line 12. Further, after removing superfluous arcs at most $O(c)$ remain, which is within the required bounds. This concludes the proof that the running time of our core algorithm is within $O(cm)$.

3.4.2 Piercing Heuristic

In this section, we describe how we implement the piercing oracle used in the previous section. Given an unbalanced arc cut C, the piercing oracle should select a piercing

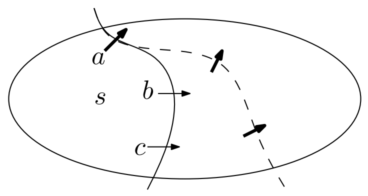


Figure 3.3: The curves represent cuts, the current one is solid. The arrows are cut-arcs, bold ones result in augmenting paths. The dashed cut is the next cut where piercing any arc results in an augmenting path.

arc that is not part of the final balanced cut in at most $O(|C|)$ time. Piercing the source side and target side cuts are analogous and we therefore only describe the procedure for the source side. Denote by $a = (q,p)$ the piercing arc with piercing node $p \notin S$.

Our piercing heuristic is composed of two parts: The primary and the secondary heuristic. The primary heuristic first narrows down the set of potential piercing arcs and the secondary heuristic then chooses from this smaller set.

3.4.3 Primary Heuristic: Avoid Augmenting Paths

The first heuristic consists of avoiding augmenting paths whenever possible. Piercing an arc a leads to an augmenting path, if and only if $p \in T_R$, i.e., a non-saturated path from p to a target node exists. As our algorithm has computed T_R, it can determine in constant time whether piercing an arc would increase the size of the next cut. The proposed heuristic consists of preferring edges with $p \notin T_R$ if possible. It is possible that none or multiple $p \notin T_R$ exist. In this case our algorithm employs a further heuristic to choose the piercing arc among them.

However, the secondary heuristic is often only relevant in the case that an augmenting path in unavoidable. Consider the situation depicted in Figure 3.3. Our algorithm can choose between three piercing arcs a, b, and c. It will not pick a as this would increase the cut size. The question that remains is whether b or c is better. The answer is that it nearly never matters. Piercing b or c does not modify the flow and therefore not T_R. Which piercing arcs result in larger cuts is therefore left unchanged. No matter whether b or c is picked, picking a in the next iteration results again in an augmenting path. The algorithm will therefore eventually end up with the same cut composed solely of arcs that should be avoided unless perfect balance is achieved first. This cut is represented as dashed line in Figure 3.3. We know that the dashed cut has the same size as all cuts found between the current cut and the dashed cut. Further, the dashed cut has the best balance among them and therefore dominates all of them. It therefore does not matter which of these dominated cuts are enumerated and in which order they are found.

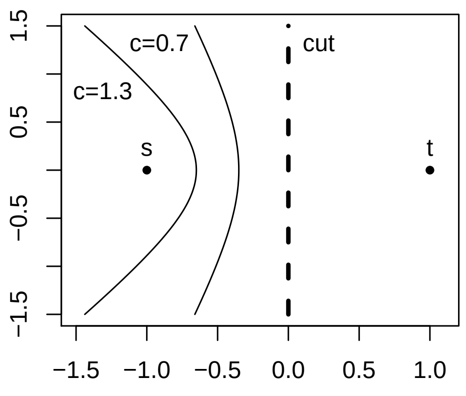


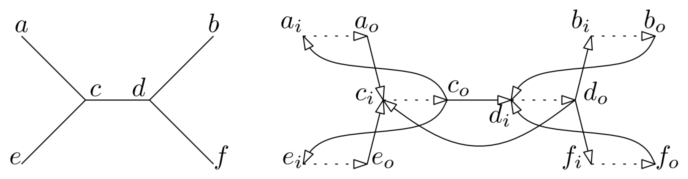
Figure 3.4: Geometric interpretation of the distance heuristic.

This means that most of the time our avoid-augmenting-paths heuristic does the right thing. However, it is less effective when cuts approach perfect balance. In this case it is possible that perfect balance is achieved before the dashed cut is found. The result consists of a race between source and target sides to claim the last nodes. Not the best side wins, but the first that gets there.

3.4.4 Secondary Heuristic: Distance-Based

If our primary avoid-augmenting-paths heuristic does not uniquely determine the piercing arc, we use a secondary distance heuristic to tie-break between the remaining choices. Our algorithm picks a piercing arc such that $\text{dist}(p,t) - \text{dist}(s,p)$ is maximized, where s and t are the original source and target nodes. The $\text{dist}(p,t)$-term avoids that the source side cut and the target side cut meet as nodes close to t are more likely to be close to the target side cut. Subtracting $\text{dist}(s,p)$ is motivated by the observation that s has a high likelihood of being positioned far away from the balanced cuts. A piercing node close to s is therefore likely on the same side as s. Our algorithm precomputes the distances from s and t to all nodes before the core algorithm is run. This allows it to evaluate $\text{dist}(p,t) - \text{dist}(s,p)$ in constant time inside the piercing oracle.

The distance heuristic has a geometric interpretation as depicted in Figure 3.4. We interpret the nodes as positions in the plane and the distances as being euclidean. The set of points p for which $\|p-s\|_2 - \|p-t\|_2 = c$ holds for some constant c is one branch of a hyperbola whose two foci are s and t. The figure depicts the branches for $c = 1.3$ and $c = 0.7$. The heuristic prefers piecing nodes on the $c = 1.3$-branch as it maximizes c. A consequence of this is that the heuristic works well if the desired cut follows roughly a line perpendicular to the line through s and t. This heuristic works on many graphs but there are instances where it breaks down. For example cuts with a circle-like shape are problematic. This geometric interpretation also works in higher-dimensional spaces.



(a) Input graph (b) Expanded graph

Figure 3.5: Expansion of an undirected graph G into a directed graph G'. The dotted arrows are internal arcs. The solid arrows are external arcs.

3.5 Extensions

Our base algorithm can be extended to compute general small cuts that are independent of an input st-pair, to compute node separators, and to compute contraction orders.

3.5.1 General Cuts

Our core algorithm computes balanced st-cuts. In many situations cuts independent of a specific st-pair are needed. This problem variant can be solved with high probability by running FlowCutter q times with st-pairs picked uniformly at random. Indeed, suppose that C is a Pareto-optimal cut such that the larger side has $\alpha \cdot n$ nodes (i.e. $\alpha = (\epsilon + 1)/2$) and q is the number of st-pairs. The probability that C separates a random st-pair is $2\alpha(1 - \alpha)$. The success probability over all q st-pairs is thus $1 - (1 - 2\alpha(1 - \alpha))^q$. For $\epsilon = 33\%$ and $q = 20$, the number of pairs we recommend in our experiments, the success probability is larger than 99.99%. For larger α this rate decreases. However, it is still large enough for all practical purposes, as for $\alpha = 0.9$ (i.e. $\epsilon = 80\%$) and $q = 20$ the rate is still slightly above 98.11%. The number of st-pairs needed does not depend on the size of the graph nor on the cut size. If the instances are run one after another then the running time depends on the worst cut's size which may be more than c. We therefore run the instances simultaneously and stop once one instance has found a cut of size c. The running time is thus in $O(qcm)$. As we set $q = 20$, it is a constant and therefore the running time is in $O(cm)$.

This argumentation relies on the assumption that it is enough to find an st-pair that is separated. However, in practice the positions of s and t in their respective sides influence the performance of our piercing heuristic. As a result it is possible that in practice more st-pairs are needed than predicted by the argument above because of effects induced by the properties of the piercing oracle.

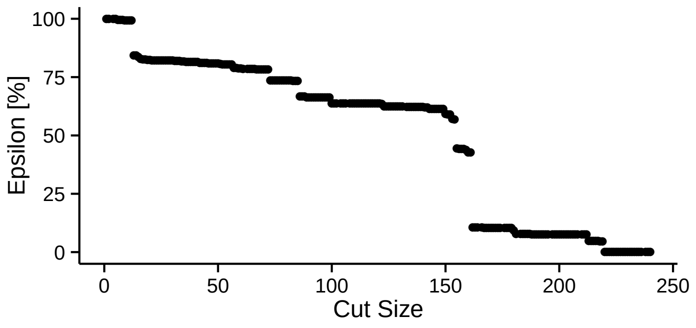


Figure 3.6: Edges cuts found by FlowCutter with 20 random source-target pairs for the Central Europe graph used in the experiments.

3.5.2 Node Separators

To compute contraction orders, node separators are needed and not edge cuts. To achieve this, we employ a standard construction to model node capacities in flow problems [4]. We transform the symmetric input graph $G = (V,A)$ into a directed expanded graph $G' = (V',A')$ and compute flows on G'. We expand G into G' as follows: For each node $x \in V$ there are two nodes x_i and x_o in V'. We refer to x_i as the *in-node* and to x_o as the *out-node* of x. There is an *internal arc* $(x_i,x_o) \in A'$ for every node $x \in V$. We further add for every arc $(x,y) \in A$ an *external arc* (x_o,y_i) to A'. The construction is illustrated in Figure 3.5. For a source-target pair s and t in G we run the core algorithm with source node s_o and target node t_i in G'. The algorithm computes a sequence of cuts in G'. Each of the cut arcs in G' corresponds to a separator node or a cut edge in G depending on whether the arc in G' is internal or external. From this mixed cut our algorithm derives a node separator by choosing for every cut edge in G the endpoint on the larger side. Unfortunately, using this construction, it is possible that the graph is separated into more than two components, i.e., we can no longer guarantee that both sides are connected.

3.5.3 Contraction Orders

Our algorithm constructs contraction orders using an approach based on nested dissection [78, 99]. It bisects G along a node separator Q into subgraphs G_1 and G_2. It recursively computes orders for G_1 and G_2. The order of G is the order of G_1 followed by the order of G_2 followed by the nodes in Q in an arbitrary order. Selecting Q is unfortunately highly non-trivial.

The cuts produced by FlowCutter can be represented using a plot such as in Figure 3.6. Each point represents a non-dominated cut. The question is which of the many points to choose. After some experimentation, we went with the following heuristic: Pick a separator with minimum expansion and at most 60% imbalance. This choice is not perfect as the experiments of Section 3.6.4 show but works well in most cases. Picking a cut of minimum expansion given a Pareto-cut-set is easy. However, we know of no easy way to do this using an algorithm that computes a single small cut of bounded imbalance, as all the competitor algorithms do. It is therefore not easy to swap FlowCutter out for any of the competitors without also dropping expansion as optimization criterion.

We continue the recursion until we obtain trees and cliques. On cliques any order is optimal. On trees an order can be derived from a so called optimal node ranking as introduced in [91]. A node ranking of a tree is a labeling of the nodes with integers $1, 2 \dots k$ such that on the unique path between two nodes x and y with the same label there exists a node z with a larger label. An optimal node ranking is one with minimum k. Contracting the nodes by increasing label yields an elimination tree of minimum depth. In [119] it has been shown that these ranking can be computed in linear running time.

Special Preprocessing for Road Graphs. Road graphs have many nodes of degree 1 or 2. We exploit this in a fast preprocessing step similar to [63] to significantly reduce the graph size.

Our algorithm first determines the largest biconnected component B using [89] in linear time. It then removes all edges from G that leave B. It continues independently on every connected component of G as described in the next paragraph. The set of connected components usually consists of B and many tiny often tree-like graphs. The resulting orders are concatenated such that the order of B is at the end. The other orders can be concatenated arbitrarily.

For each connected component our algorithm starts by marking the nodes with a degree of 3 or higher. For each degree-2-chain $x, y_1 \dots y_k, z$ between two marked nodes x and z with $x \neq z$, it adds an edge $\{x, z\}$. It then splits the graph into the graph $G_{\geq 3}$ induced by the marked nodes and the graph $G_{\leq 2}$ induced by the unmarked nodes. Edges between marked and unmarked nodes are dropped. $G_{\leq 2}$ consists of the disjoint union of paths. As paths are trees, we can therefore employ the node-ranking-based tree-ordering algorithm described above to determine a node order. For $G_{\geq 3}$ we determine an order using FlowCutter and nested dissection as described above We position the nodes of $G_{\leq 2}$ before those of $G_{\geq 3}$ and obtain the node order of the connected component.

3.6 Experiments

We compare Flowcutter to the state-of-the-art partitioners KaHip [117], Metis [92], and InertialFlow [120]. We present three experiments: (1) we compare the produced contraction orders in terms of CCH performance in Section 3.6.2 on road graphs made available during the DIMACS challenge on shortest paths [55], (2) compare the Pareto-cut-sets in more detail in Section 3.6.3 on the same road graphs, and (3) evaluate FlowCutter on non-road graphs using the Walshaw benchmark set [124] in Section 3.6.5. Section 3.6.1 describes the experimental setup common to all experiments. All experiments were run on a Xeon E5-1630 v3 @ 3.70GHz with 128GB DDR4-2133 RAM.

3.6.1 Algorithm Implementations Used and Their Configurations

Edge Cut Algorithm. We use FlowCutter in three variants denoted by F3, F20, and F100, with 3, 20, and 100 random source-target-pairs respectively. InertialFlow was introduced in [120] but no code was published. Fortunately, the idea is simple and we therefore were able to reimplement the algorithm. We refer to it using the letter I. Metis is a well-known general graph partitioner based on a multi-level scheme. The original authors published source code, which we used in our experiments. We compare against Metis 5.1.0 which is the newest version at the time of writing and refer to it as M. KaHip also uses a multi-level scheme but adds a lot of optimizations compared to Metis that can drastically decrease the cut sizes. The KaHip source code is also available and thus we use it for our experiments. At the time of writing, the current version of KaHip is 1.00. Unfortunately, we have observed several regressions compared to earlier versions. These regressions are due to a bug being fixed that caused certain expensive flow-based refinement steps not being run for higher imbalances. The newer version achieves smaller cuts at the expense of higher running times for higher imbalances. Because of these regressions and for comparability with previous works, we also include comparisons with the earlier versions KaHip 0.61 and KaHip 0.73 which were the current versions and therefore used when we performed the experiments for [61] and [86] respectively. We use KaHip in the strong preconfiguration and add `--enforce_balance` to the commandline for $\max \epsilon = 0$. We refer to the three variants as K0.61, K0.73, and K1.00.

Node Ordering Algorithms. Metis provides its own node ordering tool called `ndmetis`, which we use. Unfortunately, no other package provides a similar tool. We have therefore implemented a nested dissection algorithm on top of them. For KaHip 1.00 and InertialFlow we use the same straight-forward nested dissection implementation that computes one edge cut at each level and recurses until either cliques or trees are reached. Edge cuts are transformed into node separators by picking the nodes on one side incident to the cut edges. For KaHip we use a maximum imbalance of 20% and for InertialFlow we use 60%. For KaHip 0.61 we use an older nested dissection implementation originally written for [61]. It is not optimized for running time and only for quality. At each level, it invokes KaHip several times with different random seeds and picks the smallest cut found. It calls KaHip repeatedly on every level until for ten consecutive calls no smaller cut is found. We do this to reliably get rid of variations in achieved cut sizes that are due to randomization. However, this setup is unfavorable to KaHip as it results in large running times. We decided to stick with the old ordering routine for K0.61 for comparability with [61] and use an ordering scheme for K1.0 that only computes one cut per level. The FlowCutter nested dissection implementation is based on the same code as used for KaHip 1.00 and InertialFlow but uses the separator variant of FlowCutter and performs the low-degree node optimizations described in Section 3.5.3.

KaHip v1.00 includes a more sophisticated tool to transform edge cuts into node separators using the algorithm of [116]. We tried using it in combination with the newer nested dissection scheme with one separator per level, but we needed 19 hours to compute orders for the small California and Nevada graph used in our experiments. We were not able to compute orders on the larger instances in reasonable time and therefore omit this algorithm from our comparison.

3.6.2 Order Experiments

We computed contraction orders for 4 DIMACS roads graphs [55]. Our results are summarized in Table 3.7.

Instances. The smallest test instance is the DIMACS Colorado graph with $n = 436\text{K}$ and $m = 1\text{M}$. Next is California and Nevada with $n = 1.9\text{M}$ and $m = 4.6\text{M}$, followed by (Western) Europe with $n = 18\text{M}$ and $m = 44\text{M}$ and finally a graph encompassing the whole USA with $n = 24\text{M}$ and $m = 57\text{M}$.

Relations between Columns. Table 3.7 contains a lot of data. However, some columns are related. We therefore first point these relations out and then limit our discussion to the remaining non-related columns. We observe that, modulo small cache effects, the customization time is correlated with the number of triangles and the average query running time is correlated with the number of arcs in the CCH. These correlations are non-surprising and were predicted by CCH theory. Denote by n_s and m_s the number of nodes and arcs in the search space. For the average numbers we observe that $1.7 \le \frac{n_s(n_s-1)}{2}/m_s \le 2.6$ and for the maximum numbers we observe that $2.1 \le \frac{n_s(n_s-1)}{2}/m_s \le 3.9$, which indicates that the search spaces are nearly complete graphs. The number of nodes and the number of arcs are thus related. We can therefore say that search space is small or large without indicating whether we refer to nodes or arcs as one implies the other.

Search spaces. One of the FlowCutter variants always produces the smallest search spaces. KaHip produces the next smaller search spaces, followed by InertialFlow. Metis is last by a large margin. It is interesting that the USA graph has a smaller

		Search Space				#Arcs		Up.	Running times		
		Nodes		Arcs [$\cdot 10^3$]		in CCH	#Tri.	Tw.	Order	Cust.	Query
		Avg.	Max.	Avg.	Max.	[$\cdot 10^6$]	[$\cdot 10^6$]	Bd.	[s]	[ms]	[μs]
Col	M	155.6	354	6.1	22	1.4	6.4	102	**2.0**	18	26
	K0.61	135.1	357	4.6	22	1.7	7.2	103	3 837.1	21	20
	K1.00	136.4	357	4.8	22	1.5	6.9	99	1 052.4	20	20
	I	151.2	542	6.2	38	1.5	7.4	119	7.4	21	24
	F3	126.3	280	4.1	15	1.3	4.8	91	10.3	15	18
	F20	**122.4**	**262**	**3.8**	**14**	**1.3**	**4.4**	**87**	61.0	**14**	**17**
	F100	122.5	264	**3.8**	**14**	**1.3**	**4.4**	**87**	285.9	**14**	18
Cal	M	275.5	543	17.3	53	6.5	36.4	180	**9.9**	88	60
	K0.61	187.7	483	7.0	37	7.5	34.2	160	18 659.3	89	30
	K1.00	184.9	471	6.8	38	7.0	33.4	143	6 023.6	86	30
	I	191.4	605	7.1	53	6.9	34.1	161	42.6	84	31
	F3	177.5	**356**	6.2	**24**	5.9	23.4	**127**	64.1	69	27
	F20	170.0	380	**5.6**	26	**5.8**	**21.8**	132	386.8	68	**26**
	F100	**169.5**	380	**5.6**	26	**5.8**	**21.8**	132	1 831.8	**65**	**26**
Eur	M	1 223.4	1 983	441.4	933	69.9	1 390.4	926	**125.9**	2 241	1 164
	K0.61	638.6	1 224	114.3	284	73.9	578.2	482	213 091.1	971	303
	K1.00	652.5	1 279	113.4	287	68.3	574.5	451	242 680.5	934	297
	I	732.9	1 569	149.7	414	67.4	589.7	516	1 017.2	935	385
	F3	734.1	1 159	140.2	312	60.3	519.4	531	2 531.6	853	365
	F20	**616.0**	**1 102**	**102.8**	268	**58.8**	459.6	455	16 841.5	784	**270**
	F100	622.6	1 105	104.8	**239**	**58.8**	**459.4**	**449**	85 312.8	**766**	278
USA	M	990.9	1 685	249.1	633	86.0	1 241.1	676	**170.8**	2 111	651
	K0.61	575.5	1 041	71.3	185	97.9	737.1	366	265 567.3	1 250	202
	K1.00	540.3	1 063	62.3	208	88.7	648.3	439	315 942.6	1 097	179
	I	533.6	1 371	62.0	291	88.8	682.0	384	1 076.8	1 125	177
	F3	562.7	906	66.4	159	75.9	478.4	321	2 108.7	858	191
	F20	**490.6**	868	**52.7**	**154**	**74.3**	**440.5**	312	12 379.2	**812**	156
	F100	490.9	**863**	52.8	**154**	**74.3**	442.6	**311**	59 744.6	886	**155**

Table 3.7: Contraction Order Experiments. We report the average and maximum over all nodes v of the number of nodes and arcs in the CCH-search space of v, the number of arcs and triangles in the CCH, and the induced upper treewidth bound. We additionally report the order computation times, the customization times, and the average shortest path distance query times. Only the customization times are parallelized using four cores. The customization times are the median over nine runs to eliminate variance. The query running times are averaged over 10^6 st-queries with s and t picked uniformly at random. Several CCH customization variants exist. The times reported are for a non-amortized, non-perfect customization, with SSE and uses precomputed triangles.

search space than the Europe graph. The ratio between the average and the maximum search space sizes is very interesting. A high ratio indicates that a partitioner often finds good cuts, but at least one cut is comparatively bad. This ratio is never close to 1, indicating that road graphs are not perfectly homogeneous. In some regions, probably cities, the cuts are worse than in some other regions, probably the countryside. However, compared to the competitors, the ratio is higher for InertialFlow. This illustrates that its geography-based heuristic is effective most of the time but in few cases fails noticeably at finding a good cut.

Number of Arcs. A small search space size is not equivalent with the CCH containing only few arcs. It is possible that vertices are shared between many search spaces and thus the CCH can be significantly smaller than the sum of the search space sizes. This effect occurs and explains why the number of arcs in CCH is orders of magnitude smaller than the sum over the arcs in all search spaces. Further, minimizing the number of arcs in the CCH is not necessarily the same as minimizing the search space sizes. This explains why Metis beats KaHip in terms of CCH size but not in terms of search space size. InertialFlow seems to be comparable to Metis in terms of CCH size, as their CCH arc counts are never significantly different. However, FlowCutter beats all competitors and clearly achieves the smallest CCH sizes.

Number of Triangles. A third important order quality metric is the number of triangles in the CCH. Metis is competitive on the two smaller graphs, but is clearly dominated on the continental sized graphs. InertialFlow and KaHip seem to be very similar on all but the USA graph. On the USA graph K1.0 is ahead of both InertialFlow and K0.61. FlowCutter also wins with respect to this quality metric producing between 20% and 30% fewer triangles than the closest competitor.

Treewidth. As the CCH is essentially a chordal graph which are closely tied to tree-decompositions, we can easily obtain upper bounds on the tree-width of the input graphs as a side product. This quality metric is not directly related to CCH performance, but is of course indirectly related as most of the other criteria can be bounded in terms of it. As such it reflects the same trend: Metis is worst, followed by InertialFlow, followed by KaHip, and FlowCutter with the best bounds. Analogous to the search space sizes, we observe that the USA graph has a significantly lower tree-width than the Europe graph, assuming that our upper treewidth bounds are not completely off.

Running Time. Quality comes at a price and thus the computation times of the orders follow nearly the opposite trend: KaHip is the slowest, followed by FlowCutter, followed by InertialFlow, while Metis is astonishingly fast.

K1.00 vs K0.61. The two KaHip versions seem to be very similar. Sometimes the newer version K1.00 is ahead and sometimes the older version K0.61 wins in terms of order quality. We explain this effect by differences in implementation in our nested dissection code. Recall that K0.61 takes the best cut of at least 10 iterations on each level, whereas K1.00 only computes a single cut. This means that K1.00 is more sensible to random fluctuations coming from bad random seeds than K0.61. On average, one run of K1.00 is better than one run of K0.61. However, the best of at least 10 K0.61 runs wins against one K1.00 run with a bad seed. This effect explains the observed variance. Both, the running times of K1.0 and K0.61, are very high but for different reasons. K0.61 is slow because of the numerous repetitions on each level. However, K1.00 is slow because the newer KaHip version is significantly slower for $\epsilon = 20\%$ than the older versions. We will see this effect in greater detail in Section 3.6.3.

F3 vs F20 vs F100. It is not always clear which of F3, F20, or F100 gives the best results. F3 is most of the time slightly worse. This suggests that three source-target pairs are enough to get good separators most of the time but not enough to be fully reliable. A bad random seed can result in good separators being missed. The difference between F20 and F100 in terms of order quality is nearly negligible. This means that F20 and F100 find nearly always at least very similar separators. We conclude that there is no real advantage of going from 20 source-target pairs to 100 on road graphs. 20 source-target pairs are enough to be quality wise nearly independent of the random seed used.

3.6.3 Pareto Cut Set Experiments

In the previous experiment, we have demonstrated that FlowCutter produces the best contraction orders. In this section, we look at the Parteo-cut sets of five graphs in more detail. These are the DIMACS California and Nevada, Colorado, USA, and Europe graphs and a Central European subgraph.

Experimental Setup. For each of these graphs we report the results in a table similar to Table 3.8. With the exception of FlowCutter, we ran each of the algorithms for various maximum imbalance input parameters (max ϵ column), effectively sampling the Pareto-set computed by each partitioner. We report the imbalance of the produced cut. This achieved imbalance can be smaller than the input parameter which is only a maximum. We further report the size of each cut and indicate whether both sides of the cut form connected subgraphs. Finally, we report the running time needed to compute each cut. To compute all reported cuts, i.e., the sampled Pareto-set, all partitioners except FlowCutter need the sum over all reported running times.

For FlowCutter we use a slightly different setup. We compute a set of Pareto-cuts using FlowCutter and then pick the best cut from this set that has an imbalance below

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.001	0.000	136	119	1342	1344	245	1579
1	0.374	0.594	0.545	0.991	0.000	0.388	87	86	109	106	216	406
3	2.333	2.333	2.334	2.944	0.001	0.071	76	76	76	69	204	257
5	3.844	3.844	3.845	3.846	0.001	0.102	61	61	61	61	255	186
10	3.844	3.844	3.846	3.845	0.000	3.169	61	61	61	61	196	81
20	3.844	3.844	3.850	3.846	0.001	3.866	61	61	61	61	138	61
30	3.844	3.844	3.850	3.845	0.001	3.866	61	61	61	61	232	61
50	3.844	3.844	3.850	3.845	0.001	3.866	61	61	61	61	198	61
70	69.575	69.575	3.850	3.846	41.178	66.537	46	46	61	61	64414	61
90	89.350	89.350	3.850	69.598	47.370	70.315	42	42	61	46	60071	46

max ϵ	Running Time [s]						Are sides connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	297.7	1902.0	2489.1	2560.7	12.2	15.7	●	●	○	○	●	○
1	264.1	1717.6	274.7	279.0	12.1	23.6	●	●	●	○	○	○
3	240.9	1584.2	720.8	665.0	12.2	31.7	●	●	●	○	●	○
5	208.0	1377.5	1262.3	1251.1	12.4	35.5	●	●	●	●	○	○
10	208.0	1377.5	2073.7	2715.5	12.4	29.7	●	●	●	●	●	●
20	208.0	1377.5	249.0	3463.3	12.2	45.6	●	●	●	●	●	●
30	208.0	1377.5	249.1	4176.1	12.3	64.8	●	●	●	●	●	●
50	208.0	1377.5	248.8	3702.3	12.4	100.7	●	●	●	●	●	●
70	156.8	1056.2	249.6	4047.7	12.9	158.7	●	●	●	●	○	●
90	144.3	965.2	249.2	6359.3	12.8	201.1	●	●	●	●	○	●

Table 3.8: Results for the DIMACS USA graph.

the requested maximum. This means that for FlowCutter one can compute all reported cuts within the time needed to compute the cut for the input parameter $\max \epsilon = 0$.

Instance Selection. Selecting meaningful and representative testing instances is difficult as can be seen from Table 3.8. For the imbalance between 20% and 50% all partitioners with the exception of Metis find a cut of the same size. One can argue that this imbalance range is the most relevant for our application. It is therefore hard to argue, based on this experiment, whether one partitioner is better than another in terms of cut quality because they are all quasi the same. All cuts with 61 edges divide the USA along the Mississippi river into east and west. This cut is so pronounced that nearly all partitioners manage to find it. However, we cannot conclude from this experiment that all partitioners are interchangeable in terms of quality. This experiment only illustrates that the USA graph is in some sense an easy instance and therefore not a good testing instance. We therefore need to look at subgraphs of the USA to be able to observe the differences in quality, that definitely exist given the difference in contraction order qualities. We provide results for the DIMACS California and Nevada graph and the DIMACS Colorado graph in Tables 3.9 and 3.10. We also ran experiments on the DIMACS Europe graph. However, because of the special geographical topology of Europe, which we discuss in detail in Section 3.6.4, this graph is also non-representative. We therefore evaluate the algorithms on a Central European subgraph induced by nodes with a latitude $\in [45, 52]$ and a longitude $\in [-2, 11]$. This subgraph has about $n = 7\text{M}$ nodes and $m = 18\text{M}$ arcs.

3.6.3.1 Discussion for USA

As already outlined, we cannot deduce much from the experimental results for the USA graph. However, there are a few observations that are interesting nonetheless. Most of these observations are also valid for all other test instance. We will therefore refrain from repeating these observations when discussing the other graphs.

Limitations of Metis. Metis is clearly dominated as it is the only partitioner unable to find the Mississippi. We can further observe that for imbalances of 70% and above Metis finds huge cuts. This is most likely a bug in the implementation. Further, while Metis does find a highly balanced cut, it is not perfectly balanced and therefore formally not a valid output for the case $\max \epsilon = 0$.

Limitations of KaHip. The running times of K0.73 are comparatively small for imbalances of 20% and higher. This is not the behavior that one would expect from the algorithm description. The running time is expected to grow with increasing imbalance as it does for K1.00. The reason for this behavior is the bug that was fixed in version 1.00. Before this version, KaHip would not do the flow based refinement steps correctly. KaHip was therefore faster but the achieved cuts can be very strange. This fixed bug is also the reason why computing contraction orders with K1.00 is so slow.

Different Mississippi cuts. Another interesting observation is that while nearly all partitioners are able to find a Mississippi cut, they find different variants of it. All cuts have size 61 but the achieved imbalances vary. FlowCutter finds slightly smaller imbalances than KaHip and InertialFlow. The cuts found by FlowCutter are therefore marginally better.

Connected Sides. FlowCutter guarantees by construction that both sides of each reported cut are connected. The other partitioners give no such guarantees. This means that the exact problem variants that they solve are slightly different. We therefore report for each of the other partitioners whether the cut they find happens to have connected sides. It is interesting that this is nearly always the case. One of the exceptions is for example the 3% imbalance of K1.00 with 69 edges. This cut is also the only situation where FlowCutter is outperformed in terms of cut size on the USA graph. However, the sides of this cut are not connected. The cut is therefore not a valid solution with respect to the exact problem setting solved by FlowCutter. This explains why it is not found.

Perfectly balanced Cuts. Even though it is not useful for our application, it is interesting to compare the algorithms in terms of perfectly balanced cuts. This is the case when $\max \epsilon = 0$ or formulated differently: The number of nodes on each side must not differ by more than one node. Past research has partially focused on this special case. KaHip even includes a special postprocessing step called cycle-refinement to reduce the sizes of perfectly balanced cuts [117]. The results are surprising. Metis is not able to find perfectly balanced cuts as the balance of the achieved cut is larger than required. For this border case the achieved cuts are thus formally not valid. Even though KaHip includes special code, the achieved cut sizes are large. They even rival those of InertialFlow, a heuristic that in the case of perfect balance degenerates to sorting the nodes by longitude and cutting along the median. KaHip's cycle-refinement clearly does not work on this kind of graph. Even though FlowCutter was not designed to compute perfectly balanced cuts, it is capable of doing so. Further these cuts found turn out to be that smallest among all competitors by a significant margin.

3.6.3.2 Discussion for California and Nevada

Perfectly balanced cuts. We include the DIMACS California & Nevada graph in our benchmark because [43] were able to determine the optimal size of a perfectly balanced cut for this graph. The optimum is 32 edges. The best cut found by the partitioners evaluated in Table 3.9 contains 39 edges and was found by FlowCutter. It

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.000	0.000	42	39	157	174	51	306
1	0.169	0.169	0.184	1.000	0.000	0.566	31	31	31	36	52	93
3	2.293	2.293	2.300	2.303	0.001	1.112	29	29	29	29	61	64
5	2.293	2.293	2.293	2.329	0.005	1.571	29	29	29	29	42	62
10	2.293	2.293	2.304	2.294	0.001	0.642	29	29	29	29	43	37
20	2.293	16.706	2.756	2.293	0.000	2.656	29	28	30	29	41	29
30	2.293	16.706	2.768	2.293	13.936	5.484	29	28	29	29	51	29
50	2.293	49.058	2.768	2.296	0.000	40.833	29	24	29	29	39	27
70	64.522	49.058	2.768	2.296	41.178	42.591	27	24	29	29	4310	26
90	87.953	89.838	2.768	82.592	47.370	85.555	20	14	29	19	3711	18

max ϵ	Running Time [s]						Are sides connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	9.5	59.8	30.8	31.6	0.8	1.1	●	●	○	○	●	○
1	8.0	53.2	14.6	14.8	0.8	1.4	●	●	●	●	●	○
3	7.7	51.0	24.0	23.5	0.8	1.7	●	●	●	●	●	○
5	7.7	51.0	36.4	35.8	0.8	2.3	●	●	●	●	●	○
10	7.7	51.0	76.2	70.9	0.8	2.2	●	●	●	●	●	○
20	7.7	49.6	15.0	94.5	0.9	2.4	●	●	●	●	○	●
30	7.7	49.6	15.5	109.7	0.8	2.9	●	●	●	●	●	●
50	7.7	43.2	15.5	137.2	0.8	3.7	●	●	●	●	●	○
70	7.1	43.2	15.4	159.6	0.8	4.9	●	●	●	●	○	○
90	5.3	25.4	15.6	125.3	0.9	5.2	●	●	●	●	○	●

Table 3.9: Results for the DIMACS California and Nevada.

is therefore off by 7 edges. However, even with a slight imbalance, i.e., max $\epsilon = 1\%$, F3, F20, and K0.73 are able to find a cut with 31 edges. As this cut is smaller than the smallest balanced cut, it is possible that this 31 edge cut is optimal.

Cut sizes. The sizes of the cuts on California seem to be similar to those of the USA graph. There is one small and very pronounced cut, the one with 29 edges, which is found by all partitioniers. However, F20 is able to find a 28 and 24 edge cut for higher imbalances. KaHip misses these cuts and sticks with the 29 edge cut. It is also interesting that InertialFlow is able to find a good 29 edge cut with 2.7% imbalance. Unfortunately, it does not find it when the input parameter is at max $\epsilon = 3\%$ but at max $\epsilon = 20\%$. This means InertialFlow is capable of finding good cuts, but max ϵ parameters that significantly differ from the desired ϵ have to be tried.

3.6.3.3 Discussion for Colorado

Perfectly balanced cuts. The authors of [43] were also able to determine the minimum size of a perfectly balanced cut on the Colorado graph. It has 29 edges. Table 3.10 shows that while FlowCutter comes closest among all the evaluated partitioners, the cut found is again significantly larger by 8 edges. For imbalances in the range of 1% to 3% F20, K0.73, and K1.00 manage to achieve cut sizes of 29 edges but no cut is perfectly balanced. All of them are therefore suboptimal. For $\epsilon = 5\%$ cuts smaller than 29 edges are found.

Cut Sizes. In contrast to the USA graph, we observe different cut sizes for the different partitioners on this instance for the relevant imbalances. We can therefore better deduce from this experiment whether a partitioner is better than another for our specific application. We observe that F20 wins with respect to every imbalance except for max $\epsilon = 3\%$ and max $\epsilon = 5\%$ where K1.00 and K0.73 respectively win by one edge. This demonstrates that FlowCutter is indeed a heuristic and does not always achieve the optimum. Comparing K1.00 and K0.73 is interesting. One could expect K1.00 to always win because it is the newer version but this is not the case. For max $\epsilon = 1\%$ K1.00 is 5 edges ahead but for max $\epsilon = 10\%$ K0.73 wins by 5 edges. This can mean that K1.00 is not always superior to K0.73. Another explanation is that both do not make enough iterations in their standard configuration to produce results that are reliable, i.e, with high probability insensitive to the random seed used. Rerunning K1.00 and K0.73 with different random seeds could change the outcome. The cut sizes of Metis are far from the competitors. InertialFlow is better than Metis but also clearly dominated.

Running Times. Metis and InertialFlow are by an order of magnitude faster but also compute worse cuts. The comparison between FlowCutter and KaHip is interesting. FlowCutter gets slower with a decreasing maximum imbalance. However, KaHip gets

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.001	0.000	37	37	74	49	40	259
1	0.308	0.277	0.970	0.023	0.002	0.088	31	29	34	29	39	96
3	0.308	0.277	2.999	0.553	0.000	0.748	31	29	29	28	51	70
5	0.308	4.263	4.290	0.553	0.025	0.897	31	28	27	28	40	60
10	0.308	9.073	9.467	0.550	0.001	1.413	31	23	23	28	47	46
20	17.664	19.995	11.761	18.842	16.671	13.984	22	19	22	19	376	27
30	22.784	27.606	12.249	27.737	23.080	23.125	18	14	20	14	521	21
50	22.784	40.630	9.772	40.630	42.409	36.365	18	12	23	12	14	14
70	22.784	57.602	12.000	40.630	41.177	48.771	18	11	23	12	1124	12
90	88.080	87.330	12.084	81.224	47.362	81.495	17	8	20	9	856	9

max ϵ	Running Time [s]						Are sides connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	2.0	12.1	4.5	3.6	0.1	0.2	●	●	○	○	●	○
1	1.7	9.9	2.8	2.7	0.2	0.3	●	●	●	●	●	○
3	1.7	9.9	4.0	4.4	0.2	0.3	●	●	●	●	●	○
5	1.7	9.6	5.1	7.1	0.1	0.3	●	●	●	●	●	○
10	1.7	8.1	9.1	15.7	0.2	0.3	●	●	●	●	○	○
20	1.3	6.8	3.2	16.5	0.2	0.3	●	●	●	●	○	●
30	1.1	5.2	3.0	23.3	0.2	0.4	●	●	●	●	○	●
50	1.1	4.5	3.4	35.2	0.1	0.4	●	●	●	●	●	○
70	1.1	4.2	3.5	40.8	0.2	0.5	●	●	●	●	○	●
90	1.1	3.1	3.5	24.4	0.2	0.6	●	●	●	●	○	●

Table 3.10: Results for the DIMACS Colorado.

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.000	0.000	292	240	716	674	369	1180
1	0.232	0.132	0.998	0.916	0.000	0.089	275	220	245	216	360	391
3	0.232	0.132	0.457	2.086	0.000	0.008	275	220	227	207	372	319
5	4.963	4.894	0.464	1.470	0.000	0.857	271	213	227	208	369	276
10	6.914	9.330	0.043	8.862	0.000	0.375	243	180	228	207	375	241
20	19.419	10.542	3.139	10.546	0.000	0.132	225	162	250	162	375	220
30	19.419	10.542	3.139	10.543	0.017	7.384	225	162	250	162	369	203
50	19.419	44.386	3.139	10.547	33.336	10.542	225	155	250	162	9881	162
70	63.775	66.655	3.139	10.547	41.178	44.386	100	86	250	162	14375	155
90	84.199	84.199	3.139	10.544	83.087	84.257	13	13	250	162	28	17

max ϵ	Running Time [s]						Are sides connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	231.4	1390.3	369.1	315.9	3.3	4.3	●	●	○	○	●	○
1	230.2	1342.9	80.2	72.2	3.3	7.9	●	●	●	●	●	○
3	230.2	1342.9	112.5	111.7	3.1	10.2	●	●	●	●	●	○
5	229.9	1319.0	158.3	206.5	3.3	12.3	●	●	●	●	●	●
10	225.0	1181.5	338.1	455.1	3.1	16.8	●	●	●	●	●	○
20	215.7	1089.5	75.5	355.4	3.1	25.6	●	●	●	●	○	●
30	215.7	1089.5	75.4	395.9	3.1	34.9	●	●	●	●	●	●
50	215.7	1047.8	75.3	467.4	3.2	47.5	●	●	●	●	○	●
70	101.8	591.6	75.5	560.4	3.2	82.8	●	●	●	●	○	●
90	13.8	92.8	75.4	633.0	3.3	17.1	●	●	●	●	●	○

Table 3.11: Results for the DIMACS Central Europe graph.

slower with an increasing maximum imbalance, i.e., the other way round. A clear ranking is therefore not possible but the tendency for max imbalances above 10% is that F3 is the fastest, followed by K0.73, followed by F20, and finally K1.00.

3.6.3.4 Discussion for Central Europe

Cut sizes. In Table 3.11, we report the results of our experiments for the Central Europe graph. The most striking observation is that the cut sizes in this graph are larger than those in any of the USA graphs. This explains why the Europe graph has a higher tree-width and larger search spaces than the USA graph. It is not immediately clear which cut is the best for our application, however, the cuts with sizes 180, 162, and 155 seem to offer a good trade-off between cut size and imbalance. F20 manages

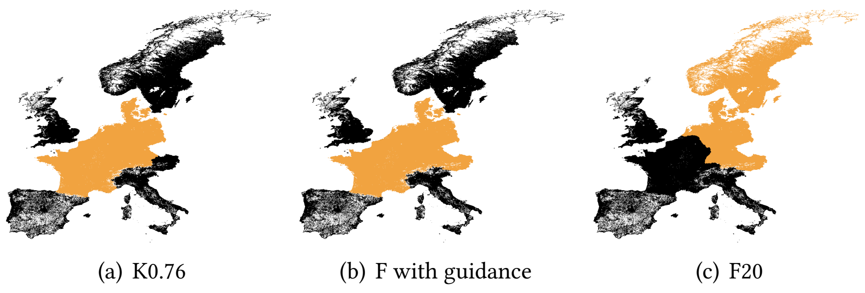


Figure 3.12: Various top-level Europe cuts.

to find all of them. K1.00 finds a variant of the 162 edge cut with a marginally higher imbalance. InertialFlow is able of finding the 162 and the 155 edge cuts. Unfortunately, as already previously observed we need to set the max ϵ parameter significantly higher than the imbalance of the cuts for InertialFlow to find them.

Running Times. On all of the USA graphs F20 was at least on par with K1.00 in terms of running time and often even faster. On this graph we see a significant gap of at least a factor 2 for all imbalances below 70%. The explanation is that the running time of FlowCutter does not only depend on the graph size but also on the cut size. As this graph has larger cuts than the USA graphs, FlowCutter is slower. KaHip's running time is not or at least less affected by cut size and therefore comes out ahead on this graph. However, for our particular application, i.e., nested dissection, a running time sensitive to the cut size is a good thing. We have only few top level cuts with large cuts but many more low level cuts that have tiny cuts. A partitioner that gets faster, the smaller the cuts become is therefore useful in this scenario as it gets faster on the lower levels. This observation also explains why F20 wins against K1.00 in terms of running time on the Europe graph in the CCH experiment.

3.6.4 Special Structure of the Europe Graph

We present the results for the Europe graph in Table 3.13. The reported cut sizes do not follow the pattern observed on the other graphs. The cuts of F20 are significantly larger than those of KaHip. Another observation is that most of the cuts found by partitioners except FlowCutter do not have connected sides. This already hints at the root of the problem. In Figure 3.12, we visualized the cuts found. Figure 3.12(a) depicts the cut found by KaHip with 112 edges and Figure 3.12(c) depicts the cut found by F20 with 188 edges. Visually these two cuts look very different. To explain the effect in detail we must first describe some properties of the Europe graph.

max ϵ	Achieved ϵ [%]						Cut Size					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	0.000	0.000	0.000	0.000	0.003	0.000	369	276	1296	1299	402	1579
1	0.930	0.930	1.000	0.984	0.003	0.337	234	234	169	154	398	417
3	2.244	2.244	2.717	2.654	0.003	0.357	221	221	130	130	306	340
5	2.244	4.918	2.976	2.985	0.003	0.171	221	216	129	129	276	299
10	9.453	9.453	8.092	7.875	0.003	0.174	188	188	112	112	460	284
20	9.453	9.453	9.405	7.888	0.003	7.539	188	188	126	112	483	229
30	9.453	9.453	9.232	8.216	0.003	9.060	188	188	128	111	465	202
50	9.453	42.080	9.232	8.214	33.336	9.453	188	58	128	111	31127	188
70	64.477	67.497	9.232	32.079	41.178	64.724	58	22	128	86	53365	38
90	72.753	72.753	9.232	72.753	70.741	72.753	2	2	128	2	44	2

max ϵ	Running Time [s]						Are sides connected?					
	F3	F20	K0.73	K1.00	M	I	F3	F20	K0.73	K1.00	M	I
0	508.4	3475.5	1887.5	1893.9	8.9	11.3	●	●	○	○	○	○
1	468.6	3292.7	224.7	196.9	8.9	19.0	●	●	○	●	●	○
3	455.7	3215.0	317.5	303.3	8.9	28.0	●	●	●	●	○	○
5	455.7	3181.7	510.2	524.8	8.9	33.6	●	●	●	●	○	○
10	411.5	2913.3	934.0	1419.1	9.0	49.3	●	●	○	○	○	●
20	411.5	2913.3	198.8	1646.2	8.9	69.9	●	●	○	○	○	○
30	411.5	2913.3	193.6	1569.3	8.9	94.0	●	●	○	○	●	●
50	411.5	949.4	194.1	1727.7	9.1	172.9	●	●	○	○	○	●
70	134.4	371.9	193.9	1642.2	9.4	79.0	●	●	○	○	○	●
90	7.6	51.9	194.1	3411.3	9.0	18.9	●	●	○	●	○	●

Table 3.13: Results for the DIMACS Europe.

Unique Geography. Top-level Europe has a unique geographic topology. There is a well connected center formed by France, Germany, Belgium, Luxembourg, and the Netherlands. Further, there are four peninsulas. Spain and Portugal are only connected by a comparatively small piece of land with France. Italy is separated by the rest of Europe by the Alps. Sweden and Norway are separated by the Baltic Sea from Central Europe. They are only connected to Denmark by a highway bridge in Kopenhagen. This bridge is also the cut with 2 edges with 72% imbalance found by several partitioners. Great Britain is separated by the North Sea and is only connected to the continent using ferries, which are treated as roads in the benchmark dataset. There are further ferries between Spain and England, and between Spain and Italy. However, there are no ferries from or to Scandinavia.

Structure of KaHip Cuts. The KaHip cut with 112 edges separates Central Europe from its peninsulas. The sides are not connected because there is no path from Great Britain to Scandinavia. The KaHip cut with 129 edges with connected sides further separates Denmark from Germany. The sides of the cut are connected, as there is a ferry from England to Denmark and a bridge from Denmark to Sweden.

Structure of FlowCutter Cuts. The FlowCutter cut is structurally very different. FlowCutter separates Central Europe along the Rhine river and the Alps. FlowCutter cannot find the 112 edge cut because its sides are not connected. Further, it does not find the 129 edge cut because the shape of this cut is very different from what the employed piercing heuristic expects.

KaHip vs FlowCutter. At first glance, KaHip seems to be better than FlowCutter on this instance. However, this is not consistent with our observation that FlowCutter produces better contraction orders. The explanation is that, as we consider a recursive bisection, the question is not whether Central Europe must be cut along the Rhine river, but at which recursion level we do it. FlowCutter does it at the top level, whereas KaHip does it at a lower level. It is unclear which approach is better. We will investigate this question in detail below. However, before we answer this question we explore how we can modify FlowCutter to find a cut similar to the one found by KaHip.

Adapting FlowCutter. One can regard the balanced 112 edge cut of KaHip as union of four smaller edge cuts with a higher imbalances. There is one cut for each peninsula. Repeatedly cutting of each peninsula on consecutive levels of the recursion is equivalent with cutting them all in one level. The question is therefore whether FlowCutter is able to find one of the peninsula cuts and this is indeed the case. FlowCutter finds the cut with 2 edges that separates Scandinavia from the rest. However,

	Lat	Lon	Place
Source	49.0	8.4	Karlsruhe
Target	41.0	16.9	Bari
	38.7	-9.1	Lisbon
	53.5	-2.8	Liverpool
	59.2	18.0	Stockholm

Table 3.14: Handpicked source and target nodes.

	Search Space				#Arcs		Up.
	Nodes		Arcs [$\cdot 10^3$]		in CCH	#Tri.	Tw.
	Avg.	Max.	Avg.	Max.	[$\cdot 10^6$]	[$\cdot 10^6$]	Bd.
F3	734.1	1 159	140.2	312	60.3	519.4	531
F20	616.0	1 102	102.8	268	**58.8**	459.6	455
F100	622.6	1 105	104.8	**239**	**58.8**	459.4	449
F3+H	625.1	1 151	106.2	262	60.2	509.2	**439**
F20+H	601.2	**1 064**	98.9	261	**58.8**	456.9	444
F100+H	**600.6**	1 065	**98.6**	250	**58.8**	**454.4**	444

Table 3.15: Contraction Order Experiments on the DIMACS Europe graph. F3, F20, F100 are the default FlowCutter variants that use a top-level cut along the Rhine river. F3+H, F20+H, F100+H use a handpicked top-level cut separating Central Europe from the peninsulas.

FlowCutter refrains from choosing this cut from the Pareto-set because we have a hard bound on a maximum imbalance of at most 60%.

Another option to help FlowCutter is to handpick source and target nodes. We selected the nodes which are closed to the coordinates given in Table 3.14 and used these as input to FlowCutter. These coordinates are not magic numbers. They represent positions chosen at the extremities of the peninsulas and in the center of Central Europe. Most humans are able to deduce this information from looking at a Europe map. With this setup we were not able to find the 112 edge cut with 7.9% imbalance found by KaHip. However, we were able to find another cut with a seemingly better trade-off. This new cut is depicted in 3.12(b). It has 87 edges and 15% imbalance. The smaller cut results from placing Austria on the other side of the cut compared to the 112 edge cut of KaHip and from some minor improvements along the other borders. KaHip is incapable of finding this cut.

The Best Top-level Cut. We have shown that with a bit of help it is possible to push FlowCutter towards computing a small cut that separates the peninsulas. Now, we will answer the question whether this a better top-level cut than the 188 edge cut found by the default FlowCutter configuration. We derive an 87 node separator from the 87 edge cut and place these nodes at the end of the contraction order manually. We then run FlowCutter on the resulting sides recursively without any further manual guidance. In Table 3.15, we report the characteristics of the so obtained orders. The new orders are marked with "+H", indicating human interaction. We compare them with the default FlowCutter orders. The new orders seem to be slightly superior with respect to every criteria except the maximum number of arcs in the search space where the default FlowCutter orders seem to win. Further, the orders seem to produce a similar number of edges in the CCH regardless of the top-level cut used. However, the differences in order quality are very minor. We observe with respect to no criterion a difference that is larger than 2%. This difference can be due to a peninsula top-level cut being slightly better. However, another explanation is that FlowCutter finds better cuts on the lower levels because a difficult to find peninsula cut was eliminated manually. In either case, the differences are so small that we decided that it is not worthwhile to automatize the selection of a top level peninsula cut.

3.6.5 Walshaw Benchmark Set

A popular set of graph partitioning benchmark instances is maintained by Walshaw [124]. The data contains 34 graphs and solutions to the edge-bisection problem with non-connected sides and maximum imbalance values of $\epsilon = 0\%$, $\epsilon = 1\%$, $\epsilon = 3\%$, and $\epsilon = 5\%$. These archived solutions are the best cuts that any partitioner has found so far. A few of them were even proven to be optimal [43]. Comparing against these archived solutions allows us to compare FlowCutter quality-wise against the state of the art. We want to stress that this state of the art was computed by a large mixture of algorithms with an even larger set of parameters that may have been chosen in instance-dependent ways. We compare this against a single algorithm with a single set of parameters. Further FlowCutter was designed for higher imbalances than 5%. It was not tuned for the cases with a lower imbalance. FlowCutter only computes cuts with connected sides. We therefore filter out all graphs that are either not connected or where the archived $\epsilon = 0$-solution has non-connected sides. Of the 34 graphs only 24 remain. The results are reported in Tables 3.16 and 3.17.

For $\epsilon = 5\%$ there are only six graphs where FlowCutter does not match the best known cut quality. These are: "144", "cs4", "m14b", "wave", "wing", and "wing_nodal". For three of these graphs, FlowCutter finds cuts that are larger by a negligible amount of at most 5 edges. For the other three, the cuts found are larger but are still close to the best known solutions. For lower imbalances, the results are not quite as good but still very close to the best known solutions.

		minimum edges in cut for				running
graph	algorithm	$\epsilon = 0\%$	$\epsilon = 1\%$	$\epsilon = 3\%$	$\epsilon = 5\%$	time [s]
144	F20	6 649	6 608	6 514	6 472	2 423.82
144K nodes	F100	6 515	6 479	6 456	6 366	10 437.91
1074K edges	Reference	**6 486**	**6 478**	**6 432**	**6 345**	
3elt	F20	**90***	**89**	**87**	**87**	0.36
4720 nodes	F100	**90***	**89**	**87**	**87**	1.87
13K edges	Reference	**90***	**89**	**87**	**87**	
4elt	F20	149	**138**	**137**	**137**	1.97
15K nodes	F100	**139***	**138**	**137**	**137**	9.50
45K edges	Reference	**139***	**138**	**137**	**137**	
598a	F20	2 417	2 390	**2 367**	**2 336**	545.69
110K nodes	F100	2 400	**2 388**	**2 367**	**2 336**	2 675.32
741K edges	Reference	**2 398**	**2 388**	**2 367**	**2 336**	
auto	F20	10 609	10 283	9 890	**9 450**	13 445.66
448K nodes	F100	10 549	10 283	9 823	**9 450**	66 249.82
3314K edges	Reference	**10 103**	**9 949**	**9 673**	**9 450**	
bcsstk30	F20	6 454	6 347	**6 251**	**6 251**	245.65
28K nodes	F100	6 408	6 347	**6 251**	**6 251**	1 230.27
1007K edges	Reference	**6 394**	**6 335**	**6 251**	**6 251**	
bcsstk33	F20	10 220	**10 097**	**10 064**	**9 914**	118.38
8738 nodes	F100	10 177	**10 097**	**10 064**	**9 914**	573.02
291K edges	Reference	**10 171**	**10 097**	**10 064**	**9 914**	
brack2	F20	742	**708**	**684**	**660**	58.13
62K nodes	F100	742	**708**	**684**	**660**	283.99
366K edges	Reference	**731***	**708**	**684**	**660**	
crack	F20	**184**	**183**	**182**	**182**	2.17
10K nodes	F100	**184**	**183**	**182**	**182**	10.97
30K edges	Reference	**184**	**183**	**182**	**182**	
cs4	F20	381	371	367	360	11.68
22K nodes	F100	372	370	365	357	58.11
43K edges	Reference	**369**	**366**	**360**	**353**	
cti	F20	342	**318**	**318**	**318**	6.10
16K nodes	F100	339	**318**	**318**	**318**	30.55
48K edges	Reference	334	**318**	**318**	**318**	
fe_4elt2	F20	**130***	**130**	**130**	**130**	1.86
11K nodes	F100	**130***	**130**	**130**	**130**	9.19
32K edges	Reference	**130***	**130**	**130**	**130**	

Table 3.16: Performance on the Walshaw benchmark set, table part 1. "Reference" is the best known bisection for the graph as maintained by Walshaw. A "*" marks solutions for which optimality has been shown.

graph	algorithm	minimum edges in cut for $\epsilon = 0\%$	$\epsilon = 1\%$	$\epsilon = 3\%$	$\epsilon = 5\%$	running time [s]
fe_ocean	FlowCutter 20	504	431	**311**	**311**	89.70
143K nodes	FlowCutter 100	483	408	**311**	**311**	418.60
409K edges	Reference	**464**	**387**	**311**	**311**	
fe_rotor	FlowCutter 20	2 115	2 091	**1 959**	1 948	334.58
99K nodes	FlowCutter 100	2 106	2 067	**1 959**	**1 940**	1 636.78
662K edges	Reference	**2 098**	**2 031**	**1 959**	**1 940**	
fe_sphere	FlowCutter 20	**386**	**386**	**384**	**384**	5.98
16K nodes	FlowCutter 100	**386**	**386**	**384**	**384**	30.84
49K edges	Reference	**386**	**386**	**384**	**384**	
fe_tooth	FlowCutter 20	3 852	3 841	3 814	**3 773**	413.48
78K nodes	FlowCutter 100	3 836	3 832	3 790	**3 773**	2 067.54
452K edges	Reference	**3 816**	**3 814**	**3 788**	**3 773**	
finan512	FlowCutter 20	**162***	**162**	**162**	**162**	8.11
74K nodes	FlowCutter 100	**162***	**162**	**162**	**162**	39.01
261K edges	Reference	**162***	**162**	**162**	**162**	
m14b	FlowCutter 20	3 858	**3 826**	**3 823**	3 805	2 115.07
214K nodes	FlowCutter 100	**3 836**	**3 826**	**3 823**	3 804	10 512.24
1679K edges	Reference	**3 836**	**3 826**	**3 823**	**3 802**	
t60k	FlowCutter 20	80	79	73	**65**	2.98
60K nodes	FlowCutter 100	80	77	**71**	**65**	14.55
89K edges	Reference	**79**	**75**	**71**	**65**	
vibrobox	FlowCutter 20	10 614	10 356	10 356	10 356	139.90
12K nodes	FlowCutter 100	10 365	**10 310**	**10 310**	**10 310**	680.76
165K edges	Reference	**10 343**	**10 310**	**10 310**	**10 310**	
wave	FlowCutter 20	8 734	8 734	8 734	8 724	2 723.12
156K nodes	FlowCutter 100	8 716	8 673	8 650	8 590	13 583.59
1059K edges	Reference	**8 677**	**8 657**	**8 591**	**8 524**	
whitaker3	FlowCutter 20	**127***	**126**	**126**	**126**	1.49
9800 nodes	FlowCutter 100	**127***	**126**	**126**	**126**	7.00
28K edges	Reference	**127***	**126**	**126**	**126**	
wing	FlowCutter 20	790	790	790	790	80.11
62K nodes	FlowCutter 100	790	790	781	773	401.82
121K edges	Reference	**789**	**784**	**773**	**770**	
wing_nodal	FlowCutter 20	1 767	1 764	1 715	1 691	27.02
10K nodes	FlowCutter 100	1 743	1 740	1 710	1 688	134.05
75K edges	Reference	**1 707**	**1 695**	**1 678**	**1 668**	

Table 3.17: Performance on the Walshaw benchmark set, table part 2.

In terms of running time the results are more mixed. Some cuts are found very quickly, while FlowCutter needs a significant amount of time on others. This is due to the fact that its running time is in $O(cm)$. If both, the cut size c and the edge count m, are large, then $O(cm)$ is large. However, for graphs with small cuts the algorithm scales nearly linearly in the graph size.

3.7 Chapter Conclusion

We introduce FlowCutter, a graph bisection algorithm that optimizes balance and cut size in the Pareto sense. The core algorithm computes small, balanced edge cuts separating two input nodes s and t. Upon this core algorithm, we build algorithms to compute overall small, balanced edges cuts independent of an st-pair specified in the input. We further extend our algorithm to compute small, balanced node separators. By combining FlowCutter with a nested dissection-based strategy, we compute contraction orders. We show that our orders beat the state-of-the-art in terms of quality on road graphs. We evaluate the quality of our orders by directly applying them in the context of Customizable Contraction Hierarchies, a speedup technique for shortest paths described in the previous chapter. Further, we show that FlowCutter manages to equate the best known cuts for many instances of the Walshaw benchmark set, demonstrating that FlowCutter is applicable beyond just bisecting road graphs. Finally, we use FlowCutter to compute tree-decompositions of small width. To evaluate the performance of our method, we submitted FlowCutter to the PACE2016 challenge [39], where it won the first place in the corresponding sequential track. This demonstrates that FlowCutter works well on a broad class of graphs. The source code of the PACE 2016 submission is available at [128].

4 Theoretical Results

In the previous two chapters, we focused on designing and evaluating algorithms. An in-depth theoretical understanding of the employed structures was not a primary goal. In this chapter, we therefore shift the focus and present several interesting theoretical insights and relationships.

In the first section, we present an introduction into tree decomposition theory. We show how contraction orders and by extension CCH relate to tree decompositions. [15]. We further show that there is a one-to-one correspondence between tree decompositions and multilevel partitions with node separators. The later result is especially interesting considering that MLD [121, 87, 45], the main CCH competitor, follows a multilevel partitioning scheme. Afterwards, we present worst case bounds for the CCH performance in terms of tree decomposition theory concepts. Finally, we compare our worst case analysis with the CH worst case analysis using highway dimension, presented in [3].

4.1 Relation between Contraction Orders, Chordal Graphs, Tree Decompositions and Multilevel Graph Partitions

Most insights in this section have been published elsewhere. For example the relation between contraction orders, chordal supergraphs and tree decompositions is well-known and we include it here only for completeness. We refer to the surveys [24], [25] and [26] for an introduction. However, the relation between tree decompositions and multilevel graph partitions is less well-known and was unknown to us before we started our research. While all three survey papers contain theorems about chordal graphs, not a single paper even mentions multilevel graph partitions.

In this section, we start by recapitulating the definitions of all relevant terms. In Subsection 4.1.2, we describe how the various concepts relate and can be interconverted. Afterwards, in Subsection 4.1.3, we illustrate the bags of road graph tree decomposition.

4.1.1 Definitions

A graph $G' = (V', E')$ is a *subgraph* of a graph $G = (V, E)$, if $V' = V$ and $E' \subseteq E$. We denote this relation by $G' \subseteq G$. Similarly, G is a *supergraph* of G'. Let S be a node subset, i.e., $S \subseteq V$. We denote by $G \setminus S$ the subgraph of G induced by the node set $V \setminus S$.

An undirected graph is *chordal* if for every cycle C with at least four nodes there exists an edge between two nodes of C that are not adjacent within C. A chordal supergraph G' of a graph G is a supergraph of G that is chordal. *Triangulated graph* is a synonym for chordal graph, also used in the literature.

A *contraction order* of an undirected graph G is an order O of the nodes of G. A supergraph is obtained by iteratively contracting the nodes of O. In the previous chapters, we referred to this supergraph as CCH. *Elimination order* is a common synonym for contraction order in the literature.

A *tree decomposition* of a graph $G = (V, E)$ is a pair (B, T), where B is the set of *bags* and T is the *tree backbone*. Every bag $b \in B$ is a set of nodes, i.e., $b \subseteq V$. T is a tree where the bags are the nodes, i.e., B is the set of nodes of T. A tree decomposition must fulfill three criteria to be valid:

1. Every node is in a bag, i.e., $\bigcup_{b \in B} b = V$.
2. For every edge $\{x, y\}$ of G, there must be a bag $b \in B$ such that both end points are in b, i.e., $x \in b$ and $y \in b$.
3. For every node x, the subgraph of the tree backbone T induced by all bags that contain x is a tree.

A *rooted tree decomposition* is a tree decomposition, where the tree backbone is directed towards a root node r. A tree decomposition is *degenerate* if it contains a bag b_1 that is a subset of another bag b_2, i.e., $b_1 \subset b_2$. The *width* of a tree decomposition is the maximum size of a bag plus one.

In this chapter, a *node separator* S of a graph $G = (V, E)$ is a non-empty node set, i.e., $S \subseteq V$. S decomposes G into the connected components of $G \setminus S$. Usually, one requires there to be at least two connected components in $G \setminus S$. However, in this chapter, we allow for the degenerate case of there only being one connected component. We say that a separator S *separates* two nodes of $V \setminus S$, if the two nodes are in different connected components in $G \setminus S$.

There exist a lot of papers that compute or employ multilevel graph partitions [7]. Unfortunately, the term refers to a general concept and not a mathematically precise construct. The details of many papers and even Chapter 6 differ from the definition of this chapter as different applications have different requirements. In this chapter, we work with the formal definition given in the next paragraph. A significant difference between it and many papers is that the definition given here separates cells using node separators instead of edge cuts. The reason we use node separators is that we obtain a tighter coupling with tree decompositions.

In this chapter, we define a *multilevel partition* P of a graph $G = (V, E)$ as a set of node subsets of V. Each element c of P induces a subgraph G_c of G. These subgraphs are called *cells*. The nodes in c are called the *interior* nodes of the cell. Nodes in

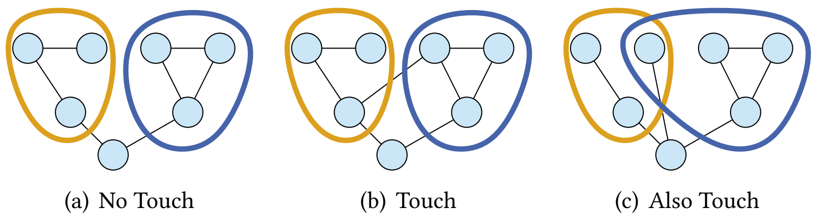


Figure 4.1: Graph with two cells illustrating the touch definition.

G that are adjacent to a node of c but not in c form the *boundary* of the cell. Two cells c_1 and c_2 *touch*, if there exists an edge in G with one endpoint in c_1 and the other endpoint in c_2 or c_1 and c_2 share a node. Figure 4.1 illustrates this definition. Requiring that two cells that share a node touch is only needed to handle degenerate cases where cells are not internally connected. We require that a multilevel partition P fulfills the following properties:

- There is a cell that encompasses the whole graph, i.e., $V \in P$. We refer to this cell as the *top level cell*.
- Touching cells are totally ordered by inclusion.

The *parent* p of a cell c is a cell such that $c \subset p$ and no other cell q exists such that $c \subset q \subset p$. Similarly, c is a *child* of p. Multilevel partitions are sometimes called *hierarchical decomposition* or *multilevel overlays* in the literature.

A multilevel partition is often obtained by recursively dividing cells. A common approach is to start with a multilevel partition P consisting of only the top level cell V. As long as there exists a cell c that has not yet been divided and contains more nodes than a given threshold, one computes a separator S of c and adds the connected components of $G_c \setminus S$ to the multilevel partition P.

4.1.2 Interconverting Structures

In this section, we describe how the structures introduced in the previous section can be interconverted. We use the toy graph depicted in Figure 4.2 to illustrate the various transformations.

From Contraction Order to Chordal Supergraph. Iteratively contract the nodes in a graph G along a contraction order O to obtain a set of shortcuts. The union of G with the shortcuts is the desired chordal supergraph G'.

Figure 4.3 illustrates a chordal supergraph of our toy graph. It can be obtained by contracting the nodes in the following order: k, l, a, b, d, n, j, o, m, f, g, h, e, q, r, i, p, c.

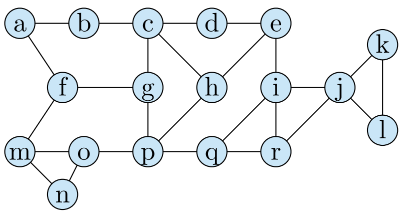


Figure 4.2: Example Toy Graph.

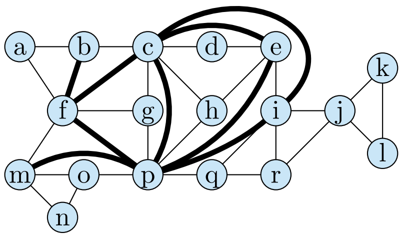


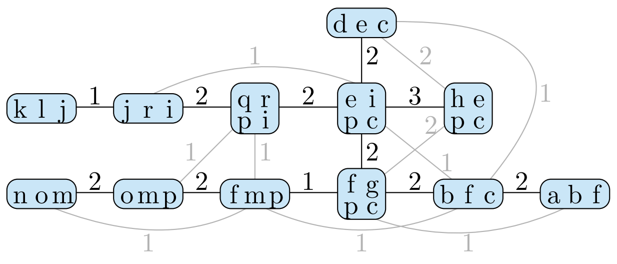
Figure 4.3: Chordal Supergraph. The thick edges were added to the base graph.

From Chordal Supergraph to Contraction Order. Every chordal graph G' possesses a *simplicial* node v, i.e., a node v such that the neighbors of v form a clique [73]. Put v into an order O as the first node and observe that $G' \setminus v$ is chordal. We can thus iteratively continue by removing simplicial nodes and putting them into O. The obtained order O is called *perfect elimination order* [73].

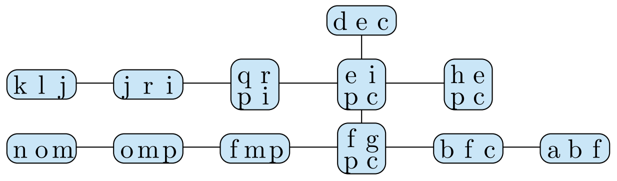
A chordal supergraph does not uniquely define a contraction order. However, the obtainable orders are equivalent in the following sense: Start with a graph G_1 and an order O_1 and construct the chordal supergraph G_2. From G_2 derive a perfect elimination order O_2. Next contract the nodes of G_1 along O_2 to obtain another chordal supergraph G_3. We have that $G_2 = G_3$, unless G_2 did not have a minimum number of edges. O_1 and O_2 thus encode essentially the same supergraph information as long as the orders were properly optimized. If the number of edges of G_2 is not minimum, then it is, in theory, possible that we tie-break the choice of the simplicial nodes in a way that reduces the number of edges, i.e., G_3 is in this case a subgraph of G_2.

The chordal supergraph from Figure 4.3 can also be obtained by contracting the nodes in the following order: a, b, n, o, m, f, g, h, d, c, e, p, q, r, i, k, j, l. This illustrates that contraction orders are not unique.

From Chordal Supergraph to Tree Decomposition. The maximal cliques of a chordal supergraph are the bags of a tree decomposition. A tree backbone can be computed using a maximum spanning tree algorithm as follows: Consider the



(a) Graph H without weight zero edges.



(b) Tree Decomposition.

Figure 4.4: Example Tree Decomposition.

weighted, complete graph H whose nodes are the maximal cliques, i.e., the bags. Edges are weighted by the size of the intersection. Every maximum spanning tree of H is a tree backbone [24]. Given a chordal supergraph, all corresponding tree decompositions have the same bag set. However, as several maximum spanning trees can exist, the tree backbone is not uniquely defined.

Figure 4.4(b) contains a tree decomposition that corresponds to the chordal supergraph of Figure 4.3. It is obtained by computing a maximum spanning tree in the graph H of Figure 4.4(a). We omitted edges with weight zero from the figure for readability.

From Tree Decomposition to Chordal Supergraph. Given a tree decomposition with bags B of a graph G, we can compute a chordal supergraph G' of G. For every pair of nodes x and y that are part of a common bag we add an edge between x and y to G if this edge does not already exist in G. The so-obtained graph is the desired chordal supergraph G' of G [24].

From Tree Decomposition to Contraction Order. To transform a tree decomposition into a contraction order, start by picking a bag b that is a leaf in the tree backbone. It has a unique neighbor p in the tree backbone. If b is a subset of p, then remove b from the tree decomposition. Otherwise, there exists a node $v \in b \setminus p$. Put v into the order and remove v from b. Iterate until the tree decomposition is gone. The resulting order is the desired contraction order.

Separators in Tree Decompositions. Every tree backbone edge e induces a separator S of G [24]. Let b and q be the endpoint bags of an edge in the tree backbone. $b \cap q$ is S. This separator can be interpreted as bisection as follows: Removing e from the tree backbone T, splits T into two trees T_1 and T_2. Let V_1 and V_2 be the union of all bags in T_1 and T_2 minus S. The nodes in V_1 and V_2 form the two sides of a bisection with separator S. It is possible that V_1 or V_2 are empty if the tree decomposition is degenerate.

For example, intersecting the bags $\{f,g,p,c\}$ and $\{e,i,p,c\}$ yields the separator $\{p,c\}$. The two sides of this separator are $V_1 = \{a,b,f,g,m,n,o\}$ and $V_2 = \{d,e,h,i,j,k,l,q,r\}$.

From Multilevel Partition to Rooted Tree Decomposition. For every cell c in the multilevel partition, we construct a bag b in a tree decomposition. b is the union of the boundary and interior nodes of c minus the interior nodes of all children. The parent-child relation between cells induces a tree on the bags. This is the tree backbone. The top level cell is the root of the rooted tree backbone. The so obtained tree decomposition can be degenerate, i.e., it is possible that bags exist that are subsets of other bags. That this construction yields a valid tree decomposition is the subject of the next theorem.

Theorem 7. The constructed tree decomposition is valid.

Proof. We need to show that the three conditions laid out in the tree decomposition definition are fulfilled.

We need to show that every node is in a bag. To prove this, we observe that every node is interior to the top level cell. A node v interior to a cell c is either in v's bag or interior to a child of c. As every cell has a finite number of descendants, we cannot build infinite chains of nested cells. We have thus proven that every node is in a bag.

Further, we need to show that for every edge $\{x,y\}$ there exists a bag b such that x and y are part of b. As touching cells are ordered by inclusion and as there are only finitely many cells, we know that there exists a smallest cell c_x that has x in its interior. x is in the bag of c_x because x is in c_x but not in a child of c_x. Let c_y be the analogous smallest cell for y. If $c_x = c_y$, then c_x is the required b. Otherwise, we observe that the existence of $\{x,y\}$ implies that c_x and c_y touch each other. They are therefore ordered by inclusion. Assume without loose of generality that $c_x \subseteq c_y$. The existence of $\{x,y\}$ implies that y is on the boundary of c_x and therefore in the bag of c_x. c_x is therefore the required bag b.

Finally, we need to show that for every node x the set of bags that include x forms a subtree of the back bone. Consider again the smallest cell c_x that contains x. All cells that contain x are ancestors of c_x. As x is in c_x, x cannot be in the bag of any ancestor of c_x. We therefore know that c_x is the only cell that has x in its interior and bag. Pick another cell d whose bag contains x. As $d \neq c_x$, we know that x is on d's

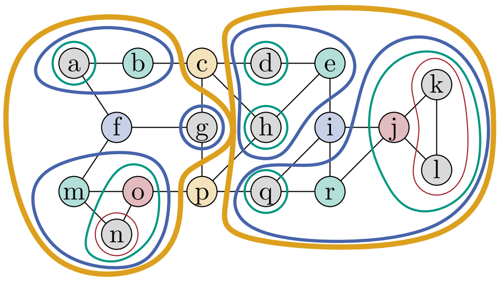


Figure 4.5: Example multilevel partition.

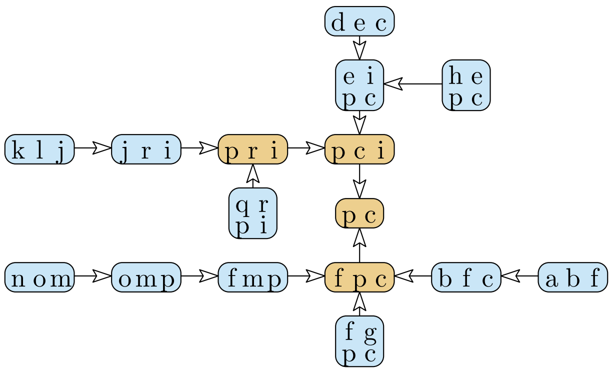


Figure 4.6: Rooted Tree Decomposition corresponding to Multilevel partition.

boundary. Denote by p the parent cell of d. x is in p or on the boundary of p. We can thus conclude that x is in the bag of p. From d we can iteratively follow the parent relation. As the parent-child relation is acyclic, we eventually arrive at c_x. As for every d the corresponding path ends at c_x, we have proven that the set of bags that include x forms a subtree of the backbone.

As we have proven all three properties, we have proven that the constructed tree decomposition is valid. □

Figure 4.5 depicts a multilevel partition with 16 cells. Every cell (except the top level cell) is depicted as closed curve. The color of a curve indicates the recursion depth. Orange indicates depth 1, blue depth 2, green has depth 3, and red indicates depth 4. Grey nodes are not part of any separator. The color of the remaining nodes indicates the depth of the separator that they are part of. Table 4.7 enumerates all cells in the multilevel partition and the derived bags.

Together with the parent-child relation of the cells, we obtain the rooted tree decomposition depicted in Figure 4.6. This tree decomposition is degenerate because

Cell Interior	Cell Boundary	Corresponding Bag
a	b, f	a, b, f
a, b	f, c	b, f, c
n	m, o	n, o, m
n, o	m, p	o, m, p
m, o, n	f, p	f, m, p
g	f, c, p	f, g, p, c
a, b, f, g, m, n, o	p, c	f, p, c
j, k, l	r, i	j, r, i
k, l	j	k, l, j
q	p, i, r	q, r, p, i
d	c, e	d, e, c
h	c, e, p	h, e, p, c
d, e, h	c, p, i	e, i, p, c
q, r, j, k, l	p, i	p, r, i
d, e, h, i, q, r, j, k, l	p, c	p, c, i
a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r	$\emptyset$	p, c

Table 4.7: Interior, boundary, and corresponding bags of the cells of the multilevel partition of Figure 4.5.

there exist bags that are subsets of other bags. After removing the direction of the tree edges and contracting the edges where one endpoint is a subset of another endpoint, we obtain the same tree decomposition as depicted in Figure 4.4(b).

There are cells, such as the cell with interior $\{a, b\}$, which is "divided" along the separator $\{b\}$ into one part, namely the cell with interior $\{a\}$. Sufficiently large cells are usually divided into more than one part. However, for tiny cells, it often occurs that such awkward separators are used.

From Tree Decomposition to Multilevel Partition. A tree decomposition T does not uniquely define a multilevel partition P because the tree backbone does not have a root. The transformation from T must therefore start by picking a root bag r. With respect to r, we can construct for every bag b except the root a cell as follows: Denote by p the parent of b, i.e., the first node on the unique path from b to r. We set the boundary of cell c to $b \cap p$. The interior of c is set to the union of all direct or indirect children bags of c minus c's boundary. We additionally construct a cell for the root bag. This cell's boundary is empty and its interior is the whole graph. It remains to show that the multilevel partition constructed this way is valid.

Theorem 8. The constructed multilevel partition is valid.

Proof. We need to show that touching cells are ordered by inclusion. Cells touch if one of two conditions is fulfilled:

- They share a node.
- There exists an edge with endpoints in both cells.

We show the required ordering property independently for both cases.

Consider some arbitrary node x and denote by T_x the subtree of the backbone induced by x. T_x contains a unique bag b_x that is closest to the root. The bags in the tree decomposition fall into three categories:

1. They are in T_x but are different from b_x.
2. They lie on the unique path from b_x to the root.
3. They are neither in T_x nor on the path.

We show that only the cells corresponding to the bags of category 2 contain x. The cells along a path are trivially ordered by inclusion.

Let q be a bag from category 1. As it is different from b_x it cannot be the root. Therefore, there exists a parent bag p of q. By construction x is also contained in p. We have that $x \in q \cap p$. x is therefore in the boundary and not in the interior of the cell corresponding to q.

Now let q be a bag from category 3. The corresponding cell is constructed by forming the union of bags that do not contain x. The union thus also does not contain x.

Finally, let q denote a bag on the path. The constructed union contains all bags of T_x and thus also contains x. It remains to show that x is not on the boundary of the corresponding cell. This follows from the fact that b_x is the only bag that contains x. As b_x is the bag farthest away from the root, no parent bag of a bag on the path is b_x. x is thus not part of any boundary.

This completes the first part of the proof. Next consider the case where the cells corresponding to two bags b_x and b_y touch because there exists an edge $\{x, y\}$ between them. By convention, we set $x \in b_x$ and $y \in b_y$. We know from the second property of the tree decomposition definition that there exists a bag b_{xy} that contains x and y.

We know that the tree backbone must contain the following four paths:

- There is a path D_x from b_{xy} to b_x along T_x because trees are connected.
- Using an analogous argument, we know that there exists a path D_y from b_{xy} to b_y along T_y.
- We can follow the parent relation from b_x to the root and obtain a path U_x.

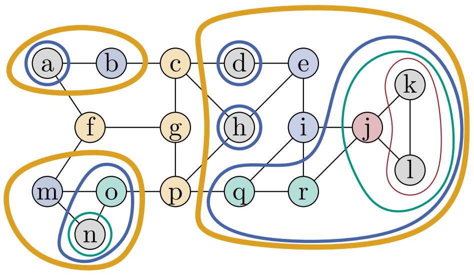


Figure 4.8: Example multilevel partition derived from tree decomposition in Figure 4.4(b) by choosing $\{f,g,p,c\}$ as root.

- Analogously, there exists a path U_y from b_y to the root.

By concatenating all four paths U_x, U_y, D_y, and D_x, we obtain a cycle in the tree backbone. As the tree backbone is a tree, we conclude that the cycle must be degenerate. The set of the root r, b_x, b_y, and b_{xy} can therefore only contain at most two elements. We conclude that one of the following conditions must hold as otherwise the set would contain three or more elements:

- $b_x = b_y$,
- $b_x = r$, or
- $b_y = r$.

In the first case, the cells are equal and thus ordered. In the second and third cases, one of the cells is the root cell and thus by definition a superset of the other cell.

This completes the second and last part of the proof. We have proven that the constructed multilevel partition is valid. □

If we pick $\{f,g,p,c\}$ as root in the tree decomposition of Figure 4.4(b), we obtain the multilevel partition of Figure 4.8. This multilevel partition is a strict subset of the multilevel partition of Figure 4.5. One can argue that "obviously" the obtained multilevel partition can be improved, because the separator used to divide the top level cell differs from $\{p,c\}$, which is a small, highly balanced separator. It is clearly "better" than $\{f,g,p,c\}$ as $\{p,c\}$ is a subset. Fortunately, we can create the desired cell, by placing a bag $\{p,c\}$ onto the tree backbone edge between the bags $\{e,i,p,c\}$ and $\{f,g,p,c\}$ and using $\{p,c\}$ as root. However, one can also argue that inserting this bag is "obviously" suboptimal, because the tree decomposition now contains adjacent bags where one bag is a subset of the other bag. It is interesting that the intuition for what is "obviously" better depends on whether a tree decomposition or a multilevel partition is used and the intuitions contradict themselves. This suggests that it is not obvious at all, what the better structure is.

Rooted Tree Decompositions are Multilevel Partitions. We described how to convert a multilevel partition P to a rooted tree decomposition T. Further, we described how to convert the rooted tree decomposition T back to a multilevel partition P'. The remaining open question is whether information is lost during the round trip, i.e., whether $P = P'$. This is the subject of the following theorem.

Theorem 9. There is a one-to-one correspondence between rooted tree decompositions and multilevel partitions.

Proof. We start with a multilevel partition P and transform it into a rooted tree decomposition T and transform it back into a multilevel partition P'. Both transformations have a one-to-one correspondence between bags and cells. We can thus pick a cell $c \in P$, the corresponding bag b of T, and the corresponding cell c' in P'. If we show that for every c the equality $c = c'$ holds, then we have shown the theorem.

Every node in c is by construction either in b or in a descendant bag of b. No interior node is thus lost in the conversion. Further, the descendant bags of b only consist of nodes in c or boundary nodes of c. No extra interior node is thus created in the conversion. We have thus $c = c'$, which completes the proof. □

4.1.3 Road Graphs Examples

In this section, we illustrate the insights described in the previous sections using road graphs. We extract a rectangular area around Germany from the DIMACS Europe graph and depict it in Figure 4.9. For every node v, we draw a dot into the image according to v's geographical position. Edges are not depicted.

Using FlowCutter, we compute a nested dissection order. From it we derive the chordal supergraph, i.e., build the CCH, and derive tree decomposition bags from the chordal supergraph. Using the maximum spanning tree algorithm, we compute a tree backbone. We select a subpath with six bags in the tree backbone. These are the bags illustrated in Figure 4.9.

A bag b corresponds to a cell c of a multilevel partition. b is the union of c's boundary and c's interior nodes minus the interior nodes of c's direct children. This is often equal to the union of the boundary of c and the boundaries of the direct children of c. In each image, we observe an approximately closed curve that is separated once along a separator. The curve is c's boundary. The remaining orange nodes is the separator along which c was separated. Every cell is divided into two children cells as FlowCutter uses nested dissection.

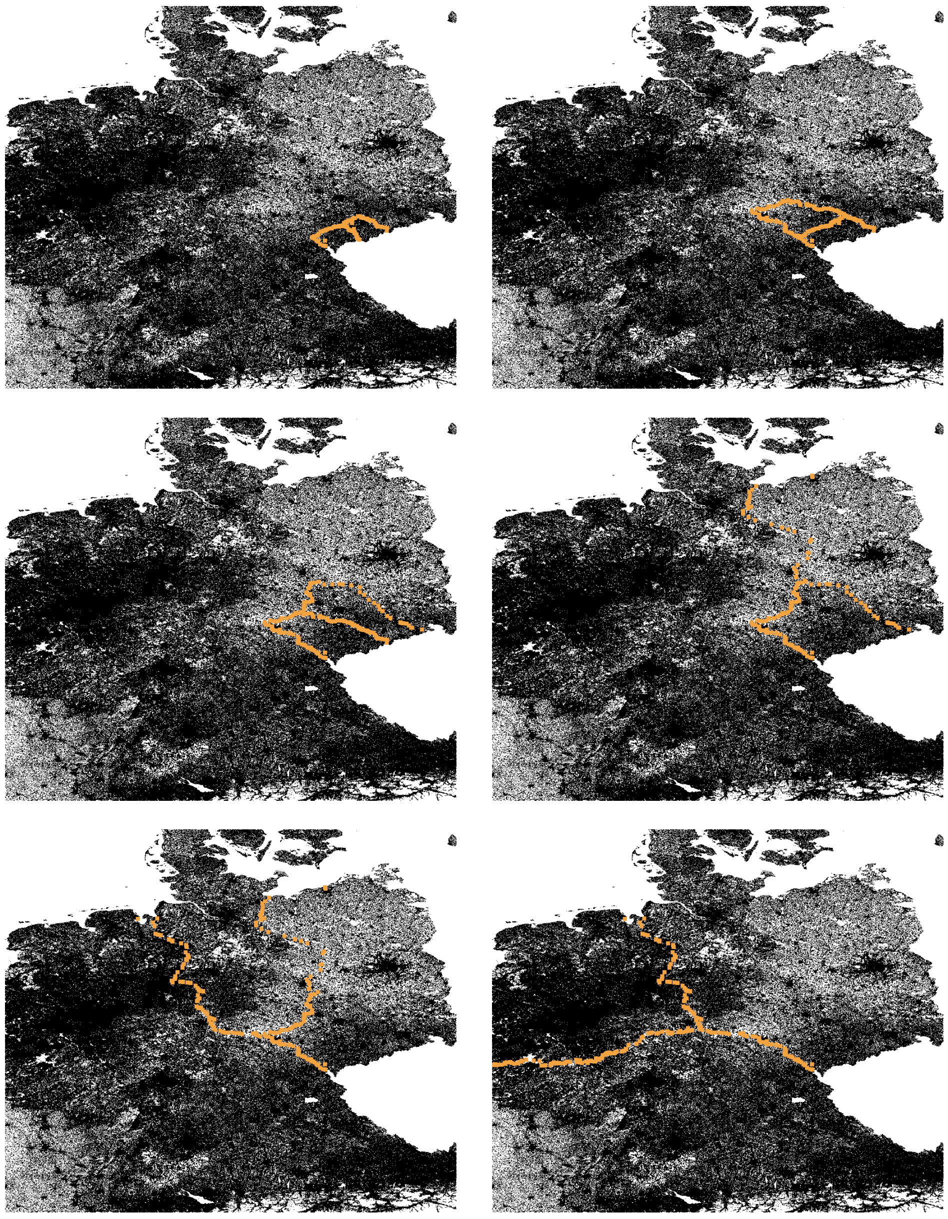


Figure 4.9: Subgraph of DIMACS Europe graph around Germany. Each image highlights the nodes in a bag of a tree decomposition. The six bags form a subpath of the tree backbone.

4.2 Worst Case Bounds for Customizable Contraction Hierarchies

We can bound the query, customization, and preprocessing running times of CCH in terms of structural graph parameters, such as the tree depth, which in turn can be bounded in terms of tree width. The *tree depth* td is defined as the minimum elimination tree height[1] over all contraction orders. The *tree width* tw is defined as the minimum width over all tree decompositions. It is known that $\text{td} \in O(\text{tw} \log n)$ [27].

A tree decomposition of minimum width does not necessarily have an elimination order that results in a minimum elimination tree height. For example, a path has a tree decomposition of width 1 as it is a tree. However, no elimination order exists that yields an elimination tree with a height smaller than $n/2$. Fortunately, a tree decomposition exists with a width in $O(\log n)$ that admits an elimination order that results in an elimination tree with a height in $O(\log n)$. It can be obtained using nested dissection with balanced separators. Besides small bags, a tree decomposition must also have a logarithmic diameter to allow for a low elimination tree depth. This logarithmic diameter corresponds to the logarithmic recursion depth obtained by recursively bisecting a graph along a balanced separator. [27] has shown that there always exists an elimination order of G that yields an elimination tree with height in $O(\text{tw} \log n)$ but the corresponding tree decomposition does not necessarily have a minimum width. Fortunately, the height of every elimination tree is an upper bound to the width of the corresponding tree decomposition. There is therefore always a tree decomposition with width $O(\text{tw} \log n)$ that admits an elimination tree of depth $O(\text{tw} \log n)$. Assuming that we could construct an elimination order Π of minimum elimination tree depth, which is NP-hard [110], we could bound the performance of the corresponding CCH in terms of td.

A CCH-query from s to t without path unpacking only explores the search spaces of s and t. The number of nodes in each of them is bounded by td and thus no more than 2td nodes are visited. The running time of this exploration is however only bounded by $O(\text{td}^2)$ as the subgraphs can be dense. We are further interested in the number of edges and triangles in the chordal supergraph G' as these correspond to the memory consumption and the customization running times, respectively. Denote by $d_o(v)$ the number of neighbors of v in the chordal supergraph G' that come after v in the contraction order, i.e., are higher than v. The number of edges in $G'$is equal to $\sum_v d_o(v)$. The number of triangles in which a node v appears as lowest node is $(d_o(v) \cdot (d_o(v) - 1))/2$ as the upper neighbors of v form a clique. The total number of triangles is therefore $\sum_v (d_o(v) \cdot (d_o(v) - 1))/2$.

We can bound $d_o(v)$ using the width of the tree decomposition corresponding to Π. Recall that this decomposition does not necessarily have a minimum width, but, as

[1]The literature uses "depth" and "height" as synonyms in this context.

shown by [27], its width can be bounded by td. We thus obtain the bounds of $O(n\text{td})$ for the number of edges and $O(n\text{td}^2)$ for the number of triangles.

It was shown by [81] that for every planar graph there exists a contraction order such that the number of edges in G' is bounded by $O(n \log n)$. This is usually a smaller bound than $O(n\text{td})$. Unfortunately, it is not known whether a contraction order exists for every planar graph that has this $O(n \log n)$-edge property and has an elimination tree height of at most $O(\text{td})$. However, we believe that it is likely true. If this order existed, then we could bound the number of triangles by $O(n\text{td} \log n)$, which is equal to $O(n\text{tw} \log^2 n)$, as follows: We can bound $\sum_v d_o^2(v)$ by $(\max_v d_o(v)) \cdot \sum_v d_o(v)$. $\max_v d_o(v)$ is at most td and $\sum_v d_o(v)$ is the number of edges, i.e., at most $O(n \log n)$. Road graphs are not strictly planar, but often planar enough to make this result relevant.

On general graphs, we thus obtain the following worst case running times parametrized in the tree width: Query running times without path unpacking are bounded by $O(\text{tw}^2 \log^2 n)$. Customization running times are bounded by $O(n\text{tw}^2 \log^2 n)$. The space requirement is bounded by $O(n\text{tw} \log n)$.

4.2.1 Comparison with Highway Dimension

The concept of highway dimension was introduced in [3] to explain why certain speedup techniques such as Contraction Hierarchies work well on road graphs. Since its introduction, the paper has been cited numerous times and has been very influential. The highway dimension h is defined with respect to the graph and the edge weights. This contrasts with the tree width tw that only depends on the graph. In the paper, the authors show that the CH query running time without path unpacking can be bounded by $O(h^2 \log^2 D)$, where D is the weighted diameter of the graph. They show that on sparse graphs the CH contains at most $O(nh \log D)$ arcs. The highway dimension theory assumes that the graph is undirected and thus the distance from s to t equals the distance from t to s.

The authors changed the exact highway dimension definition over the course of several papers. For example, in [3] and [1] they give slightly different definitions. For exposition purposes, we present the original definition of [3] as it is slightly simpler. However, we will not actively work with it. It is thus not necessary to understand it in depth to understand the rest of this section.

The idea is that for every center node c and every radius r the shortest path tree rooted at c only intersects the circle with center c and radius r at most roughly h times. This concept is formalized as follows: The highway dimension h is the smallest number, such that for every radius $r \geq 0$ and every center $c \in V$, there exists a node set $H \subseteq V$ such that

- H has at most h elements, and
- the distance from c to every node of H is at most $4r$, and

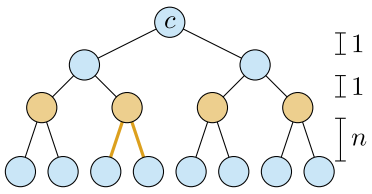


Figure 4.10: Example tree to demonstrate that the highway dimension is in $\Theta(n)$.

- for every pair of nodes s and t that have a distance of at most $4r$ from c, if a shortest path P from s to t has at least length r and contains no node that is further than $4r$ away from c, then P must pass through at least one node of H.

The obtained results are interestingly similar to ours. We are able to bound the query running time by $O(\text{tw}^2 \log^2 n)$ and using highway dimension one obtains a bound of $O(h^2 \log^2 D)$. Our bound on the number of CH arcs on general graphs is $O(n \text{tw} \log n)$. Using the highway dimension one obtains a bound of $O(nh \log D)$ for sparse graphs. The bounds are the same when exchanging tw with h and n with D. n is the maximum number of nodes in a simple path. It is very similar in spirit to the weighted diameter D. As the highway dimension theory imposes requirements on the edge weights, which our theory does not, it is a reasonable expectation that $h \in O(\text{tw})$.

Unfortunately, we can show that this expectation is false. The tree width of tree graphs is 1 whereas the worst case highway dimension is in $\Theta(n)$ [143]. To prove this consider a full binary tree where all edge weights are 1 except the weights of the edges towards the leafs which have weight n. The construction is illustrated in Figure 4.10. The highway dimension definition requires that there exists a node set H for every center node and radius. We use the tree root as center and n as radius. The graph diameter is $2(n + \log_2(n + 1) - 2)$. All nodes are thus closer than $4r$ to c as required by the definition. The paths from one leaf to the neighboring leaf has length $2n$, which is longer than r as required by the definition. The orange path in the figure illustrates such a path. There are $(n + 1)/4$ such paths. All are disjoint and all must go through H. There are thus at least $(n + 1)/4$ elements in H, which is in $\Theta(n)$. This completes the construction. The orange nodes in the figure are a possible choices for H.

This observation is a major weakness of the highway dimension theory and applies to all versions of the highway dimension definition. For nearly all speedup techniques, one can show good worst case query running times on tree graphs. Using highway dimension theory one only obtains $O(n^2 \log^2 D)$, which is significantly worse than even Dijkstra's algorithm with a running time of $O(n \log n)$.

A second major weakness of highway dimension theory is that, to the best of our knowledge, nobody has been able to numerically compute a non-trivial upper bound on any of the established benchmark graphs. It is therefore unknown whether road graphs actually have a low highway dimension. Fortunately, we were able to show that the tree width is small for many benchmark instances, i.e., our analysis does not share this weakness.

We conclude that the highway dimension does not fully explain why CHs work well on road graphs and definitely does not characterize the graphs on which a CH works well. Fortunately, our tree width-based analysis does not suffer from these weaknesses. However, our analysis assumes that the CH is constructed without witness search. It is thus possible that better bounds are achievable for CH with witness search. Such a theory could also explain why the use of metric-dependent orders in Chapter 2 results in smaller search spaces.

Recently, an alternative to the highway dimension called skeleton dimension was proposed [96]. Unfortunately, the authors only provide an analysis for the Hub Labeling speedup technique. An analysis for the Contraction Hierarchy algorithm is missing. Investigating whether the Contraction Hierarchy query running time can be bounded within the skeleton dimension seems worthwhile. Further, it would be interesting to investigate whether the skeleton dimension is a lower bound to the tree width.

4.3 Chapter Conclusion

Tree decompositions, Customizable Contraction Hierarchies, and multilevel partitions are tightly coupled. We bound the running time of our algorithm using notions from tree decomposition theory such as the tree width and depth. Finally, we compare our bounds with those obtained using highway dimension theory.

5 Dynamic Time-Dependent Routing in Road Networks through Sampling

In this Chapter, we study the road-based routing problem with predicted congestions. We propose a very simple, light-weight heuristic that works well on real world instances. A preliminary version of this work appeared on ArXiv [127]. A weakness of this preliminary study is that the employed test data is very old. It is unclear, whether the good results are due to the proposed method or are outdated test instances. After publishing [127], PTV made their current production-grade congestion predictions accessible to us for experiments. We rerun the experiments on the new test data and observe similar results as with the old real world data. This chapter is based on this newer submitted but not yet published version. Source code is available at [130].

5.1 Introduction

We study the earliest arrival and profile problems [37] with predicted and optionally realtime congestions. The input of the earliest arrival problem consists of nodes s, t, and a departure time τ. The task consists of computing a path with minimum arrival time. The input of profile problem consists only of s and t. The output consists of a function that maps a departure time onto the corresponding minimum arrival time. Formulated differently, the profile problem solves the earliest arrival time problem for every departure time. We refer to the scenario without congestions as *time-independent*. Dijkstra's algorithm [65] can be augmented to solve the earliest arrival problem [67] but it remains too slow.

Predicted congestions are usually modeled using functions as edge weights. Every edge e is associated with a function $f_e(x)$ that maps the entry time x of a car into e onto the travel time $f_e(x)$. Following [108], we require that waiting must not be beneficial. This property is called the FIFO-property and formally states that $x + f_e(x) \leq y + f_e(y)$ for all $x \leq y$. We further assume that the functions are periodic with a period length of one day. We refer to the lower bound of f_e as e's *freeflow weight*. The set of functions is the *congestion prediction*. The prediction is often done months in advance and all functions are available during preprocessing. Predictions are usually derived from past GPS traces or a traffic simulation. Routing with predicted congestions is well-researched topic [53, 56, 40, 107, 50, 13, 72, 18, 95].

Predicted congestions give a very rough traffic estimation. Often the actual traffic situation differs significantly from the prediction. For this reason, we also consider *realtime congestions*. We assume that there is a system, usually based on GPS navigation devices, that measures the current realtime travel time for every edge. A routing system that only accounts for realtime congestions computes shortest paths according to the realtime travel times. Such a routing system usually works in three phases. A preprocessing phase in which only the road network but not the travel times are known. This phase can be slow. A second customization phase that introduces the travel times and a third query phase. The customization should run within a few seconds and is rerun at regular intervals, such as for example every 10s. Solutions to routing with only realtime congestions problem are MLD/CRP [122, 88, 45] and Customizable Contraction Hierarchies (CCH) as described in Chapter 2.

Ideally, we want a routing system that accounts for predicted and realtime congestions. This scenario is also known as *dynamic time-dependent routing*. There are also works on this topic [50, 18]. In this chapter, we propose an algorithm TD-S that solves the earliest arrival problem with predicted congestions. The acronym stands for Time-Dependent Sampling. We extend it to TD-S+P, which solves the profile problem, and to TD-S+D which additionally takes realtime congestions into account.

The routing problem with no congestion and with only realtime congestions can be solved efficiently and exactly, i.e., the computed paths are shortest paths. However, when taking predicted congestions into account, many proposed algorithms compute paths with an error. Let d denote the length of the computed path and d_{opt} the length of a shortest path. We define the absolute error as $|d - d_{\text{opt}}|$ and the relative error as $\frac{|d-d_{\text{opt}}|}{d_{\text{opt}}}$. Ideally, we would like to compute paths with no error but this is not an easy task.

Fortunately, for most applications a small error is acceptable as a traveler will not notice a slightly suboptimal path. Further, the employed predictions have associated uncertainties. A shortest path with respect to the prediction is not necessarily shortest with respect to reality. Paths with a small error are therefore indiscernible from "optimal" ones in terms of quality in practice. Unfortunately, the predictions that we have access to do not quantify these uncertainties. However, it is easy to see that the uncertainty is huge: The time a typical German traffic light needs to cycle though its program is between 30s and 120s [71]. Traffic lights often adapt to the actual realtime traffic. Predictions can therefore not take red light phases into account as they cannot be predicted months in advance. Every traffic light on a shortest path thus induces an uncertainty on the same scale as its program cycle length. Consider a 2h path with a relative error of 1%. The corresponding absolute error is 72s, i.e., one to three typical traffic lights. An error of 1% is thus small even when ignoring the additional, larger uncertainties introduced by averaging travel times over many past days.

The routing problem with predicted congestions can be solved optimally using TCH [13]. Unfortunately, TCH requires a lot of memory. We compare against an open-source implementation by the primary author [12]. We cannot answer queries on a current Europe instance even with a 128GB machine — a show stopper if you need to get your software to run on your client's desktop machine which usually has far less memory. A further downside is the significant complexity of the algorithm which limits its attractivity for realworld applications and also makes extensions to, for example realtime congestions, difficult. Further, TCHs are only optimally if one assumes arbitrary precise numbers with $O(1)$-operations. Making sure that numeric instabilities do not get out of hand is possible but not easy. The simplicity of an algorithm is a huge asset in applications. Unfortunately, it is difficult to formalize when an algorithm is "simple" and when not. For the context of this chapter, we use the following crude approximation: An algorithm is complex if it combines functions using linking and merging operations[1]. Numeric stability issues are typically caused by these operations. Avoiding these operations thus also avoids these issues.

Even though a lot of research into time-dependent routing exists, in-depth experimental studies of the very simple approaches are, to the best of our knowledge, lacking. How good is an optimal solution to the time-independent routing problem with respect to the routing problem with predicted congestions? We refer to this simple approach as *Freeflow* heuristic. A further interesting question is, how good is an optimal solution to the earliest arrival problem with predicted congestions with respect to the earliest arrival problem with both congestion types? We refer to this approach as *Predicted-Path* heuristic. Both heuristics obviously compute paths with an error. However, how bad are these errors in practice? To the best of our knowledge, this fundamental question has not been investigated in existing papers. One of the objectives of our work is to fill this gap.

Our proposed algorithms can be seen as extensions of the Freeflow heuristic that trade an increased query running time for a significant decrease in error. TD-S and TD-S+P can be implemented with minimum effort assuming that a blackbox solution to the routing problem with no congestions is available. TD-S+D additionally needs a blackbox that can handle realtime congestions. Our program is build on top of the open-source CH and CCH implementations of RoutingKit [129]. Fortunately, one can swap them out for any other algorithms that fulfills the requirements. An open-source TD-S implementation is available [130].

An experimental error evaluation crucially depends on high quality realworld data. Fortunately, PTV [111] has given us access to their current production-grade congestion data for 2017 specifically to evaluate TD-S. As this data has a significant commercial value, we are not allowed to freely share it[2].

5.1.1 Related Work.

There exist a lot of papers beside the already mentioned ones. As we do not build upon them nor rerun their experiments, we limit our exposition to a very brief overview. For a detailed survey, we refer to [7].

[1]Definitions: Merge respectively link of f and g is h such that $h(x) = \min\{f(x), g(x)\}$ respectively $h(x) = f(x) + g(f(x) + x)$

[2]We are not aware of high-quality open congestion prediction data. OSM does **not** include predictions.

The authors of [107] observed that ALT [84], a time-independent speedup technique, can be applied to the graph weighted with the freeflow weights. This yields the simple algorithm TD-ALT. Unfortunately, the query running times are not convincing. In [50] the technique was therefore extended to TD-CALT by first coarsening the graph and then applying TD-ALT to the core. TD-CALT was also evaluated with respect to simulated realtime congestion. Unfortunately, coarsening, and thus TD-CALT, requires linking and merging operations. In [40] SHARC [17] (a combination of Arc-Flags [97] with shortcuts and coarsening) was extended to the time-dependent setting. SHARC was combined with ALT yielding L-SHARC [40]. Another technique is FLAT [95]. Here the idea is to precompute the solutions from a set of landmarks to every node and during the query phase to route all sufficiently long paths through a landmark. FLAT also works without linking- and merging operations. In [18] MLD/CRP was combined with travel time functions yielding TD-CRP, which can also be used in the dynamic scenario.

A deep structural insight was shown in [72]. Graphs and source and target nodes s and t exist, such that the st-profile contains a superpolynomial number of paths. Fortunately, realworld graphs do not have this worst-case structure.

5.1.2 Outline

We start by describing our implementation of the freeflow heuristic. Afterwards, we describe TD-S and the profile extension TD-S+P. In the next step, we introduce the dynamic extension TD-S+D. Finally, we experimentally evaluate all three algorithms on production-grade instances and compare our results with TCH.

5.2 The Freeflow Heuristic

The freeflow travel time along an edge assumes that there is no congestion. Formally, the freeflow travel time of an edge e is the minimum value of e's travel time function f_e. The freeflow heuristic works in two steps:

1. Find shortest time-independent path H with respect to the freeflow travel time.

2. Compute the time-dependent travel time along H for the given departure time.

The first step is independent of the edge function weights. The functions are only used in the second step. The running time is dominated by the first step. Fortunately, this step can be accelerated using any time-independent speedup technique. In our implementation, we use a CH. In a preprocessing step, we compute a CH for the road graph weighted by freeflow travel times. The first step of the freeflow heuristic computes H using a CH-query.

5.3 Time-Dependent-Sampling: TD-S

The freeflow heuristic never reroutes based on the current traffic situation. TD-S tries to alleviate this problem. Similarly to the freeflow heuristic, TD+S's query algorithm works in two steps:

1. Compute a subgraph H.
2. Run the time-dependent extension of Dijkstra's algorithm on the subgraph.

If a shortest time-dependent path P is part of H, then P is found by TD-S and the computed arrival time is exact. Otherwise, TD-S's solution has an error.

We compute the subgraph using a sampling approach. We define a constant number k of time-intervals. Within each time-interval, we average the time-dependent travel times. For every interval, we thus obtain a time-independent graph. For every graph, we compute a shortest time-independent path. The union of these paths is the subgraph H.

Similarly to the freeflow heuristic, the shortest time-independent path computations can be accelerated using existing speedup techniques. In our implementation, we compute for each time-interval a time-independent CH. The first step of TD+S executes k CH-queries. The second step uses the time-dependent extension of Dijkstra's algorithm restricted to the subgraph H.

The freeflow heuristic can be seen as a special case of TD+S, where subgraph consists of a shortest freeflow path. The number of time-intervals k is a trade-off between query and preprocessing running times, and space consumption on the one hand, and solution error on the other hand. We recommend using small numbers below 10 for k. The chosen interval boundaries should reflect rush hours in the input data.

5.4 Computing Profiles: TD-S+P

The query algorithm of TD-S+P also works in two steps:

1. Compute a subgraph H.
2. Sample at regular intervals the travel time by running the time-dependent extension of Dijkstra's algorithm restricted to the subgraph H.

The subgraph computation step is the same as for TD-S.

We interpolate linearly between the sampled travel times. For a sampling rate of 10min, our algorithm first computes the subgraph H, then runs Dijkstra's algorithm with the departure times 0:00, 0:10, 0:20...23:50 restricted to H. Denote by $a_1, a_2, a_3 \ldots a_{144}$ the computed arrival times. For example, the computed arrival time for departure time 0:07 is $0.3a_1 + 0.7a_2$.

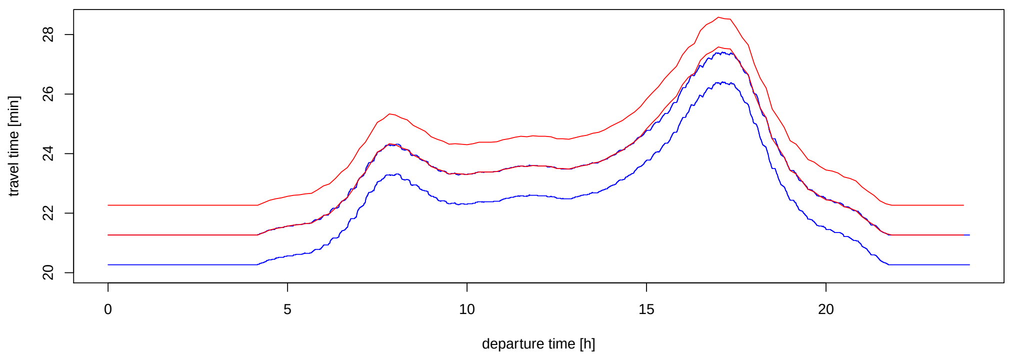


Figure 5.1: Example profile over 24h. The red curve (top) was computed with TD-S+P, while the blue one (bottom) is the exact solution. The middle overlapping curves are the actual profiles. To improve readability, we plot both curves a second time shifted by one unit on the y-axis.

We can bound the error that the profile algorithm induces on top of the error of TD-S using a theoretical argument: Denote by $\Lambda_{\max}$ the maximum and by $-\Lambda_{\min}$ the minimum slope of every linear piece in every profile function and by r the sample rate. As shown in Section 5.5, the maximum absolute error is bounded by $r(\Lambda_{\max} + \Lambda_{\min})/4$. Following [95], we assume that $\Lambda_{\max}$ and $-\Lambda_{\min}$ are bounded by a small constant. In the same paper, the values $\Lambda_{\max} = 0.19$ and $\Lambda_{\min} = 0.15$ were experimentally estimated for the Berlin instance. With $r = 10\text{min}$ this gives a maximum error of 51 seconds. Our analysis is very similar to the TRAP oracle from [95]. Unfortunately, this bounds on the error induced by the profile algorithm. The base algorithm TD-S induces additional error, which were not able to bound using theoretical arguments.

TD-S+P is faster than iteratively running TD-S as the subgraph is only computed once. Further, Dijkstra's algorithm iteratively runs on the same small subgraph H. The first run loads H into the cache and thus all subsequent runs incur nearly no cache misses.

Figure 5.1 illustrates a typical profile computed with TD-S+P. The source node of the example is near the inner city of Stuttgart, a city notoriously known for its large daily traffic congestions, in front of the central train station. The target node lies in Denkendorf, a village about 20min south-east outside of Stuttgart. The computed profile is smoother than the "exact" profile, which has a lot of small fluctuations. However, this does not mean that the "exact" profile is more accurate. Most of the small fluctuations are within only a few seconds, i.e., well below the accuracy of the input data. Formulated differently, these fluctuations are imprecisions in the input that are propagated to the output. Fortunately, the general form of both curves is very similar, which is the important information. The largest absolute error is 19s,

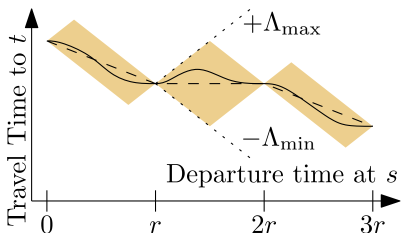


Figure 5.2: The *st*-profile is the solid line. It must be in the orange area. The dashed function is our approximation.

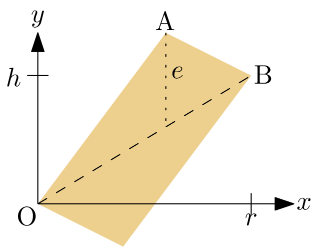


Figure 5.3: Proof Illustration. The optimal function must be in the orange area. e is the maximum error of our estimation.

respectively 1.17% relative error, at the peak of the evening spike. Recall, that crossing a single red traffic light can induce a delay of more than 19s.

5.5 Profile Error Guarantee

The proof is sketched in Figure 5.2. The solid line is the unknown optimal function. The dashed function is the computed approximative function. As the slopes are bounded, we know that the unknown optimal function is inside of the orange area. The maximum difference between the approximative function and the boundary of the orange area therefore bounds the error induced by the profile algorithm.

Consider the situation depicted in Figure 5.3. Our objective is to compute the maximum vertical distance from the dashed line inside of the orange region. As the triangle below the dashed line is the same as the triangle above it rotated by 180°, we can focus solely on the upper triangle OAB. As the distance grows from O to A and decreases from A to B we know that the distance is maximum at $x = A_x$, i.e., the maximum vertical distance is the length e of the dotted segment. From the application we know that $B_x = r$, and that the slope of the line OA is $\Lambda_{\max}$, and that the slope of AB is $-\Lambda_{\min}$. We denote by h the y-position of B. Our objective is to compute the maximum value of e over all values of h. We start by computing e and then maximize the resulting expression over h.

The line OA is described by $y = \Lambda_{\max} x$, and the line OB is described by $y = \frac{h}{r}x$, and the line AB is described by $y = -\Lambda_{\min} x + (r\Lambda_{\min} + h)$. By intersecting OA and AB we get

$$\Lambda_{\max} A_x = -\Lambda_{\min} A_x + (r\Lambda_{\min} + h)$$

which can be solved for A_x yielding

$$A_x = \frac{r\Lambda_{\min} + h}{\Lambda_{\max} + \Lambda_{\min}}$$

which leads to

$$\begin{aligned} e &= \Lambda_{\max} A_x - \frac{h}{r} A_x \\ &= (\Lambda_{\max} - \frac{h}{r}) A_x \\ &= \frac{(\Lambda_{\max} - \frac{h}{r})(r\Lambda_{\min} + h)}{\Lambda_{\max} + \Lambda_{\min}} \end{aligned}$$

which is an expression for the desired vertical height. As $0 < \Lambda_{\max}$ and $0 < \Lambda_{\min}$ the value of e is maximum if and only if

$$\begin{aligned} &(\Lambda_{\max} - \frac{h}{r})(r\Lambda_{\min} + h) \\ = \; & -\frac{h^2}{r} + (\Lambda_{\max} - \Lambda_{\min})h + r\Lambda_{\min}\Lambda_{\max} \end{aligned}$$

is maximum. As $-h^2 < 0$ this parabola is maximum when its derivative is zero. We therefore compute the derivative

$$-\frac{2h}{r} + (\Lambda_{\max} - \Lambda_{\min})$$

which is zero for

$$h = \frac{r(\Lambda_{\max} - \Lambda_{\min})}{2}$$

which we can insert into the expression of e to obtain

$$\begin{aligned} \max_h e &= \frac{(\Lambda_{\max} - \frac{r(\Lambda_{\max} - \Lambda_{\min})}{2r})(r\Lambda_{\min} + \frac{r(\Lambda_{\max} - \Lambda_{\min})}{2})}{\Lambda_{\max} + \Lambda_{\min}} \\ &= \frac{(\frac{\Lambda_{\max} + \Lambda_{\min}}{2})(\frac{r(\Lambda_{\max} + \Lambda_{\min})}{2})}{\Lambda_{\max} + \Lambda_{\min}} \\ &= \frac{r(\Lambda_{\max} + \Lambda_{\min})}{4} \end{aligned}$$

which was the absolute error bound we needed to compute.

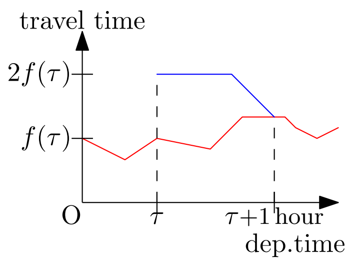


Figure 5.4: Travel-time weight function with simulated congestion. The red line is the original travel time function f. The blue line is the congested function.

5.6 Dynamic Traffic: TD-S+D

TD-S and TD-S+P work with predicted congestions. However, in many applications we must also take realtime congestions into account. We adapt TD-S by modifying the computation of the subgraph H yielding TD-S+D. The second step is left unchanged.

The subgraph H is the union of k paths. TD-S+D adds a shortest st-path according to the current realtime traffic as $k+1$-th path to the union. To efficiently determine this path, a efficient solution to the routing problem with realtime congestions is needed. We use the CCH algorithm from Chapter 2 with a FlowCutter contraction order as described in Chapter 3.

In the preprocessing step, TD-S+D computes a CCH in addition to the k CHs of TD-S. At regular time intervals, such as every 10s, TD-S+D updates the CCH edge weights to reflect the realtime traffic situation. This update involves a CCH customization, which runs within at most a few seconds. A TD-S+D query consists of running k CH queries and one CCH query. The subgraph H is the union of the $k+1$ shortest paths. Finally, the time-dependent extension of Dijkstra's algorithm is run restricted to the subgraph H.

5.6.1 Simulating Traffic

We have access to production-grade time-dependent edge data. Unfortunately, we do not have access to good measured realtime traffic. We therefore simulate realtime congestions to study the performance of TD-S+D.

For an earliest arrival time query from s to t with departure time τ, we first compute the shortest time-dependent path P with respect to the historic travel times. On P we generate three traffic congestions by picking three random start edges. From each of these edges we follow P for 4min, yielding three subpaths. For every edge e in a subpath, we generate a congestion according to the construction illustrated in Figure 5.4.

Denote by f the travel time function of e. We modify f by doubling the travel time at $f(\tau)$ and assume that it remains constant for some time. The congestion should

	Nodes [K]	Directed Edges [K]	TD-Edges [%]	avg. Break Points per TD-Edge
Lux	54	116	34	30.9
Ger	7 248	15 752	29	29.6
OGer	4 688	10 796	7	17.6
CEur	25 758	55 504	27	27.5

Table 5.5: Node count, edge count, percentage of time dependent edges, and number of break points per time-dependent weight function for all instances.

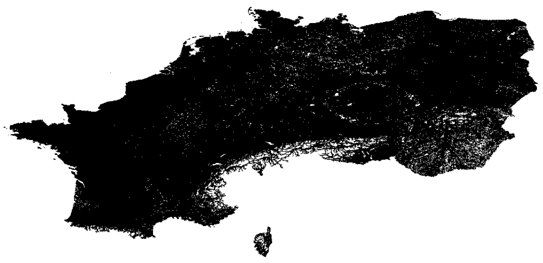


Figure 5.6: Central Europe.

be gone at $\tau + 1h$. To maintain the FIFO-property, the modified function must have slope of -1 before $\tau + 1$ before joining the predicted travel time. In the awkward and rare situation where $2f(\tau) < f(\tau + 1h)$, we do not generate a congestion.

5.7 Experimental Results

5.7.1 Setup

We use three production-grade instances (Lux, Ger, CEur) with traffic predictions for the first half of 2017. To compare with related work, we further include an a decade-old Germany instance (OGer) in our study. We thank PTV for giving us access to this data. All instances model a car on a typical Tuesday. The instance sizes are depicted in Table 5.5. The European graph contains several central European countries as illustrated in Figure 5.6. Our experiments were run on a machine with two E5-2670 processors with a total of 16 hardware non-HT threads and 64GB of DDR3-1600. For every instance, we generated 10^5 queries with source stop, target stop and source time picked uniformly at random. All experiments use the same set of test queries.

The large instances (Ger, CEur) are useful to investigate scaling behavior. The old

Graph	Algo	Exact [%]	Relative Error [%]				Absolute Error [s]			
			Avg	Q99	Q99.9	Max	Avg	Q99	Q99.9	Max
Lux	Freeflow									
Lux	TD-S+4	97.7	0.008	0.2	1.5	4.9	0.2	4	30	141
Lux	TD-S+9	99.6	<0.001	0.0	0.1	1.7	<0.1	0	3	27
Ger	Freeflow									
Ger	TD-S+4	94.6	0.005	0.1	1.0	3.0	0.8	17	159	474
Ger	TD-S+9	98.2	0.001	<0.1	0.4	3.0	0.3	1	76	374
OGer	Freeflow									
OGer	TD-S+4	96.4	0.002	0.1	0.4	2.0	0.3	6	47	333
OGer	TD-S+9	98.5	0.001	<0.1	0.2	2.0	0.1	1	24	276
CEur	Freeflow									
CEur	TD-S+4	91.1	0.006	0.2	0.7	3.8	1.8	47	226	547
CEur	TD-S+9	96.8	0.001	<0.1	0.3	1.2	0.5	6	109	397

Table 5.7: Number of exact time-dependent queries and absolute and relative errors for Freeflow, TD-S+4, and TD-S+9. "Q99" refers to the 99%-quantile and "Q99.9" the 99.9%-quantile.

instance (OGer) is useful to compare with related work. The city with periphery instance (Lux) is useful to investigate errors in urban contexts.

We evaluate TD-S with two selections of time windows. TD-S+4 uses the windows 0:00-5:00, 6:00-9:00, 11:00-14:00, and 16:00-19:00. TD-S+9 uses the windows 0:00-4:00, 5:50-6:10, 6:50-7:10, 7:50-8:10, 10:00-12:00, 12:00-14:00, 16:00-17:00, 17:00-18:00, and 19:00-21:00. These time windows reflect the rush hours in the dataset and were created using manual trial-and-error. Developing algorithms to automatically determine time windows bounds is an interesting avenue for future research.

To provide an in-depth comparison with related work, we run TCH on our test instances and test machine. The implementation we used is based on KaTCH [12], an open-source reimplementation of the algorithm of [13] by the primary author. Unfortunately, it only supports earliest arrival queries and no profile queries. We do not have access to implementations of other competing algorithms.

In Table 5.7, we report the errors for various algorithms and all instances. We report the percentage of queries that are solved exactly, the average, maximum, and quantiles[3] of the relative and absolute errors. The number of exactly solved queries decreases with instance size as the paths lengths grow with size. A longer path has

[3]Definition x-quantile of n values: Sort n values increasingly, pick the $(n \cdot x)$-th value. 0-quantile is min. 0.5-quantile is median. 1-quantile is max.

Graph	TD-Dijkstra	Freeflow	TD-S+4	TD-S+9	TCH
Average Query Running Time [ms]					
Lux	4	0.04	0.11	0.26	0.18
Ger	1 116	0.40	0.99	3.28	1.81
OGer	813	0.26	0.97	2.09	1.12
CEur	4 440	1.50	3.83	6.85	OOM
Max. Query Memory [MiB]					
Lux	13	17	29	47	328
Ger	1 550	2 132	3 630	6 127	42 857
OGer	461	855	1 880	3 589	8 153
CEur	4 980	7 058	12 411	21 336	>131 072
Total Preprocessing Running Time [min]					
Lux	—	<0.1	<0.1	0.1	0.6
Ger	—	1.9	7.6	14.7	86.2
OGer	—	1.5	5.9	16.4	26.8
CEur	—	11.0	33.9	70.7	381.4

Table 5.8: Average preprocessing and running times and memory consumption of various algorithms.

more opportunities for errors. Similarly, the absolute errors grow with instance size as the paths get longer. The relative errors quantiles shrink with growing instance size, as individual errors have a smaller impact. The average errors are very small, as most queries are answered exactly. We report maximum error values as most related papers report them. However, these values are very sensitive to the random seed used during the query generation. The Freeflow heuristic achieves significantly lower error values than we expected at first. For some applications, these are low enough. However ideally, we want to have even lower error values. Fortunately, using TD-S significantly lower values are achievable. TD-S+9 answers 99.6% of the queries exactly in an urban scenario and 96.8% on a continental-sized instance. Even the 99.9%-quantiles are well below a relative error of 0.5%. TD-S+4 has larger errors as it uses fewer time windows. Fortunately, the query running times and the memory footprint of TD-S+4 are lower.

In Table 5.8, we compare query and preprocessing running times and memory consumption. We observe that the lower the solution error of an algorithm is, the more memory it requires. TCH is guaranteed to be exact. Unfortunately, its memory requirements are prohibitive on CEur. We tried to run it on a 128GB machine but got out-of-memory crashes while executing queries. The TCH preprocessing algorithm

	Freeflow		TD-S+P4		TD-S+P9	
Graph	SubG.	Total	SubG.	Total	SubG.	Total
Lux	<0.1	2.5	0.1	3.0	0.3	3.4
Ger	0.2	18.0	0.7	19.5	1.7	22.2
OGer	0.2	9.8	0.8	11.2	1.8	12.4
CEur	0.4	36.9	2.1	49.9	5.3	53.4

Table 5.9: Average 24h-profile running times in milliseconds. “SubG” is the subgraph computation time.

writes no longer needed data to the disk and evicts it from main memory. We were therefore able to run the preprocessing step. However, the whole data structure must be loaded into main memory to answer queries. TD-S+9 only needs 21GB on the same instance and TD-S+4 only 12GB. The 5GB memory consumption of TD-Dijkstra consists essentially of the input data. TD-S+4 only needs about 2.4 times the memory required by the input. TD-S+9 needs 4.1 times the memory. TD-S+4 has an about a factor 10 lower preprocessing time than TCH. The freeflow heuristic has, for obvious reasons, the fastest query running time. It is followed by TD-S+4 which is about 33% to 50% faster than TCH. TD-S+9 is slightly slower than TCH.

In Table 5.9, we report profile query running times. In addition to the total running time, we report the amount of time needed to compute the subgraph. Even with TD-S+P9 on the large CEur graph the average running times are only slightly above 50ms for a 24h profile. The query running time of TD-S+P4 is only slightly lower than the running time of TD-S+P9. However, the later has a significantly lower error. TD-S+P9 is therefore superior to TD-S+P4 with respect to profile queries, if the larger memory footprint is not prohibitive.

A path can be optimal for several departure times throughout a day. For an st-query, we can count the number of paths that are optimal for at least one departure time. Two paths are the same if they have the same sequence of edges. It is not necessary that the travel times are the same. In Figure 5.10, we depict the number of optimal paths as function of the percentage of queries with at most that many paths. For most st-query there are only very few paths. However, there are outliers for which there can be a significant number of paths. The maximum number of observed paths on CEur is 34.

5.7.2 Dijkstra Rank Plots

In theory, it is possible that the low errors we observe in our experiments are an artifact of our test query generation method. The employed generation method is the current state-of-the-art and is used in all competitor papers. However, it has a bias

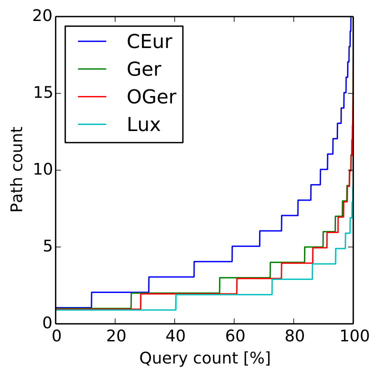


Figure 5.10: The number of optimal paths (y-axis) in function of number of queries (x-axis) for a 24h-profile of TD-S+P9.

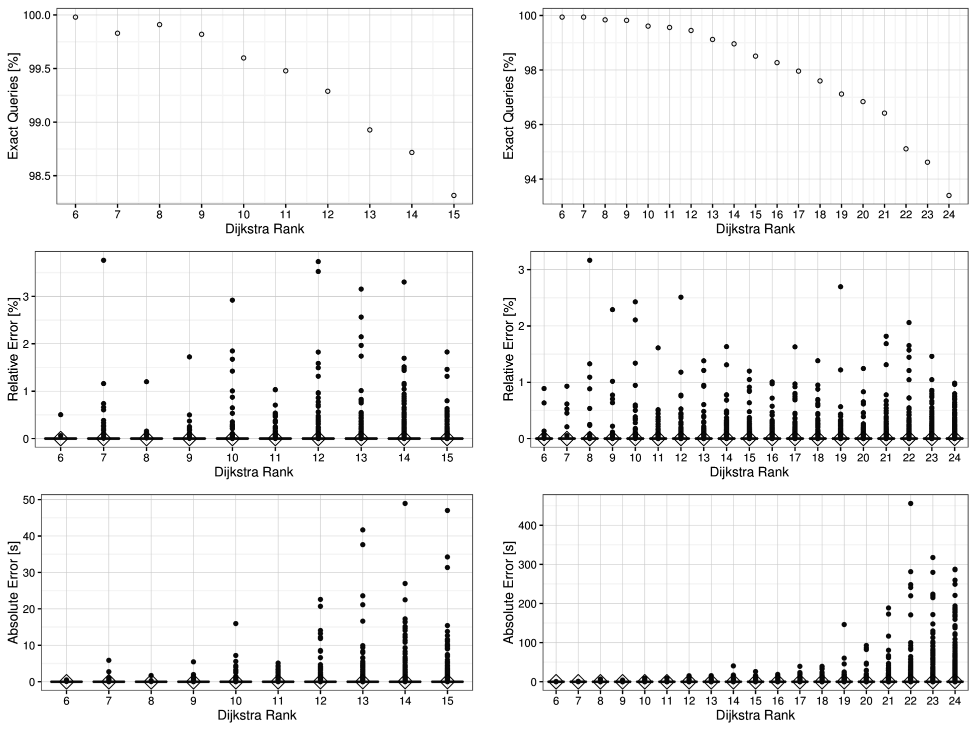


Figure 5.11: Percentage of correct queries (top), relative error (mid), and absolute error (bottom) for Lux (left) and CEur (right) in function of Dijkstra-rank.

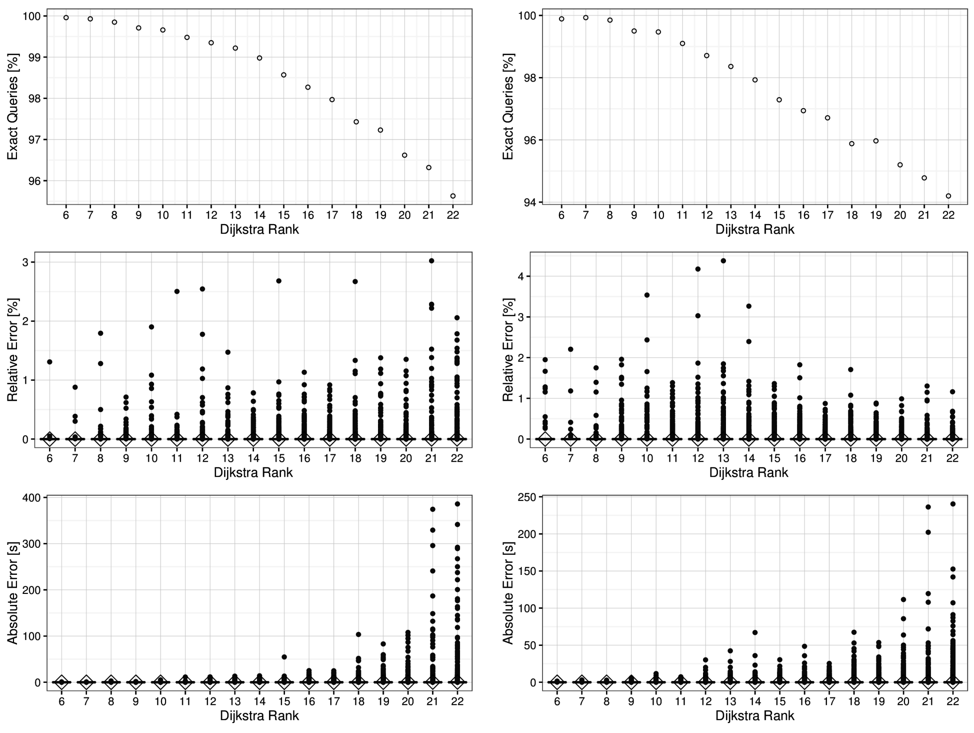


Figure 5.12: Percentage of correct queries (top), relative error (mid), and absolute error (bottom) for Ger (left) and OGer (right) in function of Dijkstra-rank.

and maybe this bias is exploited by TD-S. Picking the source and the target nodes uniformly at random nearly always yields a long-distance query. This means that on the Europe instance it is very unlikely that a test query is generated, such that the resulting path has a running time of for example 30min, which is short compared to the diameter of the instance. In theory, it is possible that only short-distance queries have a significant error but we do not generate such test queries.

To investigate this bias, we compute Dijkstra-rank plots in Figures 5.11 and 5.12. Dijkstra-rank queries are generated as follows: We pick a random source node s uniformly at random. In the next step, we order all other nodes by freeflow distance from s using Dijkstra's algorithm. The query from s to the 2^i-th node in this order has Dijkstra-rank i. We generated 1 000 random source nodes s for our experiments. The intuition is that a query with a low Dijkstra-rank is more local than a query with a high Dijkstra-rank.

We plot the number of queries that were answered optimally by TD-S+4. We further plot the relative and absolute errors of TD-S+4. It is usual to use a box plot to visualize Dijkstra-rank results. However, because of the low errors of TD-S the boxes of our plots are degenerate and squashed onto the x-axis. As most errors are zero, zooming into the plots does not help. All observable dots are outliers. To emphasize the low achieved errors and because the squashed boxes do not cover the outliers, we decided to stick with a degenerate box plot representation.

One can observe that on every instance the number of correct queries shrinks with growing Dijkstra-rank. Similarly, the absolute error grows with a growing a Dijkstra-rank. This is non-surprising as the path length also grows with Dijkstra-ranks. The relative error seems to be independent of the Dijkstra-rank.

As the number of optimally solved queries is minimum with a large Dijkstra-rank, we conclude that TD-S is not exploiting the test query generation and our experimental results from the previous section are representative.

5.7.3 Dynamic Time-Dependent Routing

		Exact [%]	Relative Error [%]				Absolute Error [s]			
Graph	Algo		Avg	Q99	Q99.9	Max	Avg	Q99	Q99.9	Max
Lux	Predict.P	1.6	17.228	56.1	75.0	93.8	323.0	739	826	997
Lux	TD-S+D4	94.7	0.017	0.5	2.3	6.2	0.6	15	93	231
Lux	TD-S+D9	95.0	0.016	0.5	2.2	6.2	0.5	14	89	231
Ger	Predict.P	55.1	1.2	17.9	36.9	79.3	78.5	552	741	1 001
Ger	TD-S+D4	90.9	0.032	1.0	2.6	7.0	3.5	116	233	474
Ger	TD-S+D9	93.4	0.026	0.9	2.5	6.2	2.8	99	216	469
OGer	Predict.P	52.3	1.352	18.7	38.3	65.8	84.9	563	738	934
OGer	TD-S+D4	91.5	0.031	1.0	2.6	5.4	3.2	108	224	462
OGer	TD-S+D9	92.9	0.028	0.9	2.5	5.4	2.9	102	219	462
CEur	Predict.P	72.6	0.392	7.0	25.9	81.9	41.0	443	653	1 870
CEur	TD-S+D4	89.5	0.015	0.5	1.6	5.2	3.3	106	244	547
CEur	TD-S+D9	94.0	0.011	0.3	1.4	5.2	1.9	69	205	397

Table 5.13: Number of exact dynamic, time-dependent queries and absolute and relative errors for the predicted path, TD-S+D4, and TD-S+D9. “Q99” refers to the 99%-quantile and “Q99.9” the 99.9%-quantile.

In the dynamic scenario, we consider two types of congestions: (a) the predicted congestion, and (b) the realtime congestion. The predicted congestions are formalized as edge weight functions. The predicted congestions used in our setup come from realworld production-grade data. The realtime congestions are randomly generated according to the scheme described in Section 5.6.1. In Table 5.13, we compare the errors

	Lux	Ger	OGer	CEur
TD-S+D4	0.3	2.3	1.7	4.3
TD-S+D9	0.5	3.6	2.9	7.8

Table 5.14: Average query running times for TD-S+D.

induced by three approaches: The Predicted Path heuristic (Predict.P) as baseline, TD-S+D4, and TD-S+D9. The Predicted Path heuristic computes a shortest path P with respect to only the predicted congestion. P is then evaluated with respect to both congestion types. In Table 5.14, we report the query running times of TD-S+D4 and TD-S+D9.

Freeflow and Predicted Path are similar in spirit. Freeflow solves the time-dependent routing problem with predicted congestions by ignoring predicted congestions. Similarly, Predicted Path solves the dynamic time-dependent routing problem by ignoring realtime congestion. The freeflow heuristic produces surprisingly small errors. This contrasts with the predicted path heuristic, whose measured errors in Table 5.13 are very large. On the Luxembourg instance only 1.6% of the queries are solved optimally. Fortunately, TD-S+D significantly reduces these errors. Over all instances, the minimum number of optimally solved queries is 92.9%. This is a huge improvement compared to 1.6%. The errors induced by TD-S+D in the dynamic scenario are larger than those of TD-S in the static scenario. Fortunately, even the 99%-quantile of TD-S+D9 is well below 1% on all test instances, which is good enough for many applications.

5.7.4 Comparison with Related Work

In Tables 5.15 and 5.16, we compare TD-S with related work on the OGer instance. Comparing the reported error values is very difficult. The state-of-the-art consists in reporting the maximum relative error over 10^x uniform random queries where x varies among papers. Most use 10^5 queries but some use 10^4 queries. We marked papers with only 10^4 queries using a †. We use 10^5. Unfortunately, maximum error is a very bad quality score. The maximum error heavily depends on x: The more queries are performed, the larger the maximum error usually gets. To illustrate this effect, we ran TD-S with only 10^4 queries and report the maximum error values in parentheses. The measured “maximum” error decreased from 2.0 to 0.7 for TD-S+9. Further, the maximum error heavily depends on the random seed used to generate the test queries. Comparing maximum error values across papers is therefore unfortunately not meaningful unless they differ by orders of magnitude. To mitigate this problem, we also report quantiles.

Besides TD-S and Freeflow, only FLAT does not need link and merge operations. As the reported maximum error values are very similar, a detailed error comparison is not meaningful. A limitation of FLAT is its large memory consumption: For OGer

	Numbers from		Link & Merge?	Relative Error [%]			Run T. [ms]	
				avg.	Q99.9	max.	ori	scaled
TDCALT-K1.00	[50]	OGer	●	0	0	0	5.36	3.77
TDCALT-K1.15	[50]	OGer	●	0.051	n/r	13.84†	1.87	1.31
eco SHARC	[40]	OGer	●	0	0	0	25.06	19.7
eco L-SHARC	[40]	OGer	●	0	0	0	6.31	5.0
heu SHARC	[40]	OGer	●	n/r	n/r	0.61	0.69	0.54
heu L-SHARC	[40]	OGer	●	n/r	n/r	0.61	0.38	0.30
TCH	Tab. 5.8	OGer	●	0	0	0	1.12	1.12
TDCRP (0.1)	[18]	OGer	●	0.05	n/r	0.25	1.92	1.38
TDCRP (1.0)	[18]	OGer	●	0.68	n/r	2.85	1.66	1.19
Freeflow	Tab. 5.7 & 5.8	OGer	○	0.05	2.2	6.5	0.26	0.26
FLAT-SR_{2000}	[95]	OGer	○	n/r	n/r	1.444†	1.28	1.18
TD-S+4	Tab. 5.7 & 5.8	OGer	○	0.002	0.4	2.0 (1.8†)	0.97	0.97
TD-S+9	Tab. 5.7 & 5.8	OGer	○	0.001	0.2	2.0 (0.7†)	2.09	2.09

Table 5.15: Comparison of earliest arrival query algorithms. "n/r" stands for not reported. We report the running times as published in the corresponding papers (ori) and scaled by processor clock speed (scaled).

	Num-bers from	Run T. [ms]	
		ori	scaled
eco SHARC	[40]	60 147	47 388
heu SHARC	[40]	1 075	847
ATCH ϵ=2.5%	[13]	38.57	30
TD-S+P4	Tab. 5.9	19.5	19.5
TD-S+P9	Tab. 5.9	22.2	22.2

Table 5.16: Comparison of profile query algorithms. We report the running times as originally reported in the corresponding publications. Further, we report running times scaled by processor clock speed.

51GB are reported [95]. TD-S4 only needs 1.8GB. We doubt that FLAT scales in terms of memory consumption to CEur. Compared with link- and merge-based techniques TD-S is highly competitive. TD-S4's average error is smaller than the average error of every non-exact competitor that reports average errors. TD-S4's query running time is only beat by Freeflow and heu L-SHARC. A major downside of L-SHARC is that it is complex. Not only is linking and merging needed. L-SHARC combines A*/ALT, Arc-Flags, and contraction. None of these components is easy to implement. TCH has, in addition to being complex, a large memory consumption.

Profile queries have not been described and evaluated for all competitors. Only [40] and [13] report experiments. We present an overview in Table 5.16. ATCH is a TCH variant. We do not expect ATCH to scale to CEur because of memory restrictions. We believe that SHARC would run on CEur but the query running times could significantly increase. On OGer, TD-S+P clearly wins in terms of query running time.

Eco SHARC and ATCH, but not heu SHARC, are exact and therefore the comparison with TD-S+P is not completely fair. However, to compute profiles on CEur, ATCH is not an option because of memory constraints. Further, eco SHARC likely has query running times above a minute and is therefore too slow for many applications. There are thus no alternatives to TD-S+P on this instance.

5.8 Chapter Conclusion

We introduce TD-S, a simple and efficient solution to the earliest arrival problem with predicted congestions on road graphs. We extend it to TD-S+P which is the only algorithm to solve the profile variant in at most 50ms on all test instances. Further, we demonstrate with TD-S+D that additional realtime congestions can easily be incorporated into TD-S.

Part II

Routing in Timetable Networks

6 Connection Scan

In the previous part of this thesis, we exclusively focus on accelerating queries in road-based transportation networks. In this part, we shift our focus to timetable-based networks. We present the Connection Scan family of algorithms, which represents a very flexible and light-weight solution to a large variety of routing problems in timetable-based networks.

Our work was published in several papers [58, 133, 62]. This part is based on a submitted but not yet accepted extended journal version. A preprint of the extended version is available [57]. For the extended version, we refined and several algorithms. Further, we reran all experiments and improved the compairison with realted work. This part is joint work with Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. It is a continuation of the topic of my diploma thesis [126]. Significant improvements to the algorithm and its design have been made with respect to every part of the algorithm compared to the variant presented in my diploma thesis. For example CSAccel, Pareto optimization, trip bits, efficient journey extraction, and the discussion about decision graph representation are not described in [126].

6.1 Introduction

We study the problem of efficiently answering queries to timetable information systems. Efficient algorithms are needed as the foundation of complex web services such as the Google Transit or bahn.de - the German national railroad company's website. To use these websites, the user enters his desired departure stop, arrival stop, and a vague moment in time. The system then computes a journey telling the user when to take which train. In practice, trains do not adhere perfectly to the timetable. It is therefore necessary to quickly adjust the scheduled timetable to the actual situation or account in advance for possible delays.

At its core, the studied problem setting consists of the classical shortest path problem. This problem is usually solved using Dijkstra's algorithm [65], which is build around a priority queue. Algorithmic solutions that reduce timetable information systems to variation of the shortest path problem, that are solved with extensions of Dijkstra's algorithm, are therefore common. The time-dependent and time-expanded graph [112] approaches are prominent examples.

In this work, we present an alternative approach to the problem, namely the *Connection Scan Algorithm* (CSA). The core idea consists of removing the priority queue and replacing it with a list of trains sorted by departure time. Contrary to most competitors, CSA is therefore not build upon Dijkstra's algorithm. The resulting algorithm is comparatively simple because the complexity inherent to the queue is missing. Further, Dijkstra's algorithm spends most of its execution time within queue operations. Our approach replaces these with faster more elementary operations on arrays. The resulting algorithm therefore achieves low query running times. A further advantage of our approach is that the data structure consists primarily of an array of trains sorted by departure time. Maintaining a sorted array is easy, even when train schedules change.

Modern timetable information systems do not only optimize the arrival time. A common approach consists of optimizing several criteria in the Pareto sense [105, 66, 22]. The practicality of this approach was shown by [106]. The most common second criterion is the number of transfers. Another often requested criterion is the price [104] but we omit this criterion from our study because of very complex realworld pricing schemes. A further commonly considered problem variant consists of profile queries. In this variant, the input does not contain a departure time. Instead, the output should contain all optimal journeys between two stops for all possible departure times. As further problem variant, we propose and study the minimum expected arrival time (MEAT) problem setting to compute delay-robust journeys.

CSA does not possess a heavyweight preprocessing step. This makes the algorithm comparatively simple but it also makes the running time inherently dependent on the timetable's size. For very large instances this can be a problem. We therefore study an algorithmic extension called Connection Scan Accelerated (CSAccel) which combines a multilevel overlay approach [121, 88, 45] with CSA.

Related Work. Finding routes in transportation networks is the focus of many research projects and thus many publications on this subject exist. The published papers can be roughly divided into two categories depending on whether the studied network is timetable-based or not. As our research focuses on timetable routing, we restrict our exposition to it and refer to a recent survey [7] for other routing problems.

Some techniques are preprocessing-based and have an expensive and slow startup phase. The advantage of preprocessing is, that it decreases query running times. A major problem with preprocessing-based techniques is that the preprocessing needs to be rerun each time that the timetable changes. We start by proving an overview over techniques without preprocessing and afterwards describe the preprocessing-based techniques.

The traditional approach consists of extending Dijkstra's algorithm. Two common methods exist and are called the time-dependent and time-expanded graph models [112]. In [49] the time-dependent model is refined by coloring graph elements. The authors further introduce SPCS, an efficient algorithm to answer earliest arrival profile queries. A parallel version called PSPCS is also introduced. We experimentally compare CSA to SPCS, to the colored time-dependent model, and the basic time-expanded model.

Another interesting preprocessing-less technique is called RAPTOR and was introduced in [52]. Just as CSA, it does not employ a priority queue and therefore is not based on Dijkstra's algorithm. It inherently supports optimizing the number of transfers in the Pareto-sense in addition to the arrival time. A profile extension called rRAPTOR also exists. We experimentally compare CSA with RAPTOR and rRAPTOR.

Adjusting the time-dependent and time-expanded graphs to account for realtime delays is conceptually straightforward but the details are non-trivial and difficult as the studies of [102] and [36] show.

In [21], SUBITO was introduced. This is an acceleration of Dijkstra's algorithm applied to the time-dependent graph model. It works using lower bounds on the travel time between stops to prune the search. As slowing down trains does not invalidate the lower bounds, most realworld train delays can be incorporated. However, CSA supports more flexible timetable updates. For example contrary to SUBITO, CSA supports the efficient insertion of connections between stops that were previously not directly connected.

In [144], trip-based routing (TB) was introduced. It works by computing all possible transfers between trains in a preprocessing step. The preprocessing running times are still well below those of other preprocessing-based techniques but non-negligible. Unfortunately, the achieved query speedup lacks behind techniques with more extensive preprocessing. In [145], the technique was extended with a significantly more heavy-weight preprocessing algorithm that stores a large amount of trees to achieve higher speedups.

Many more preprocessing-based techniques exist. For example, in [75] the Contraction Hierarchy algorithm, a very successful technique for road-based routing, was adapted for timetable-based routing. In [42], Hub-labeling, another successful technique for roads, was also adapted for timetable-based routing. A further labeling-based approach was proposed in [140]. In addition to SUBITO, [21] introduces k-flags. k-flags is an adaptation of Arc-Flags [97], a further successful technique for roads, to timetables. Another well-known preprocessing-based algorithm is called Transfer Patterns (TP). It was introduced in [6] and was refined since then over the course of several papers. In [11], the authors combined frequency-based compression with routing and used it to decrease the TP preprocessing running times. In [8], TP was combined with a bilevel overlay approach to further decrease preprocessing running times. CSAccel is not the first technique to combine multilevel routing with timetables. This was already done in [123].

We postpone giving an overview over the existing papers related to the MEAT problem until Section 9.1, as the details of the MEAT problem are described in Chapter 9.

6.2 Preliminaries

We describe the Connection Scan algorithm in terms of train networks. Fortunately, many other transportation networks exist with the same timetable-based structure. Flight, ship, and bus networks are examples thereof. We could therefore formulate our work in more abstract terms such as vehicles. However, to avoid an unnecessary clumsy language, we refrain from it, and just refer to every vehicle as train.

6.2.1 Timetable Formalization

In this section, we formalize the notion of timetable, which is part of the input of nearly every algorithm presented in this paper. We are not the first to present a formalization. However, even though many previous works exist, they use different notations. Further, they differ with respect to the exact problem formalization. We therefore explain our terminology and the model used in our work in detail to avoid confusion.

A timetable encodes what trains exist, when they drive, where they drive, and how travelers can transfer between trains. Especially, the details of the last part — changing trains — vary significantly across related work. Unfortunately, unlike one intuitively might expect, these details impact the algorithm design and can have a huge impact on the running time behavior. Further, these details can make a timetable description verbose. Therefore, we first describe the entities not related to transfers, give examples for these, and only afterwards describe the transfer details.

A *timetable* is a quadruple $(\mathcal{S},\mathcal{C},\mathcal{T},\mathcal{F})$ of stops $\mathcal{S}$, connections $\mathcal{C}$, trips $\mathcal{T}$, and footpaths $\mathcal{F}$. The footpaths are used to model transfers. We therefore postpone their description until we describe transfers. A *stop* is a position outside of a train where a traveler can stand. At a stop, trains can halt and passengers can enter or leave trains. A *trip* is a scheduled train. A *connection* is a train that drives from one stop to another stop without intermediate halt. Formally, a connection c is a five tuple $(c_{\text{dep\_stop}}, c_{\text{arr\_stop}}, c_{\text{dep\_time}}, c_{\text{arr\_time}}, c_{\text{trip}})$. We refer to these attributes as c's *departure stop*, *arrival stop*, *departure time*, *arrival time*, and *trip*, respectively. We require from every connection c that $c_{\text{dep\_stop}} \neq c_{\text{arr\_stop}}$ and $c_{\text{dep\_time}} < c_{\text{arr\_time}}$.

All connections with the same trip form a set. We require that this set can be ordered into a sequence $c^1, c^2 \dots c^k$ such that $c^i_{\text{arr\_stop}} = c^{i+1}_{\text{dep\_stop}}$ and $c^i_{\text{arr\_time}} < c^{i+1}_{\text{dep\_time}}$ for every i. In a slight abuse of notation, we sometimes identify a trip with its corresponding sequence of connections.

Examples. Examples for stops are the train main stations, such as “Karlsruhe Hbf”. Other examples include subway or tram stations.

Trips include high speed trains, subway trains, trams, buses, ferries, and more. An example for a trip is the “ICE 104” from Basel to Amsterdam that departs at 15:13 on the 2-nd of August 2016. Note, that the description “ICE 104” without the departure time does not uniquely identify a trip as such a train exists on every day of August 2016. In our model, there is a trip for every day, even though these trips share the same stop sequence and the operator refers to all trains by the same name.

Pick one of the “ICE 104” trips and name it x. The first three stops at which x halts are Basel, Freiburg, and Offenburg. There is a connection with departure stop Basel, arrival stop Freiburg, and trip x. There further is a connection with departure stop Freiburg, arrival stop Offenburg, and trip x. However, there is no connection with departure stop Basel, arrival stop Offenburg, and trip x, as we require that the train of a connection does not halt at an intermediate stop.

Transfers. A traveler standing at stop s at the time point τ can be described using a pair (s, τ). To lighten our notation, we denote these pairs as $s@\tau$. Denote by P the set of these pairs. A *transfer model* is a relation on P, which we denote using the $\rightarrow$ symbol. A traveler sitting in an incoming connection c, wishing to transfer to an outgoing connection c' of another trip, can do so by definition if and only if $c_{\text{arr\_stop}}@c_{\text{arr\_time}} \rightarrow c'_{\text{dep\_stop}}@c'_{\text{dep\_time}}$ holds.

Many transfer models exist and the details vary significantly across the literature. Unfortunately, there is no consent on what the best model is. In the following, we focus our description on the model used in our work, which is based upon footpaths. We also briefly discuss the differences to other models.

A *footpath* f is a triple $(f_{\text{dep\_stop}}, f_{\text{arr\_stop}}, f_{\text{dur}})$, which we refer to as f’s *departure stop*, f’s *arrival stop*, and f’s *duration*. We require all footpath durations to be positive, i.e., . The set of footpaths $\mathcal{F}$ is the last element of the quadruple that characterizes timetables. These footpaths can be viewed as weighted, directed *footpath graph* $G_{\mathcal{F}} = (\mathcal{S}, \mathcal{F})$, where the stops are the nodes, the footpaths the arcs, and the duration the weights. We define the transfer relation as follows: $a@\tau_a \rightarrow b@\tau_b$ holds, if and only if there is a path from a to b whose length is at most $\tau_b - \tau_a$.

Having a large connected footpath graph makes the considered problems significantly harder than having only loosely connected components. Following [52], we therefore introduce two restrictions on the footpath graph. It must be transitively closed and fulfill the triangle inequality. Transitively closed means that if there is an edge ab and an edge bc, then there is an edge ac. The triangle inequality further requires that $ab_{\text{dur}} + bc_{\text{dur}} \geq ac_{\text{dur}}$. From these two properties one can show that if there is a path from a to b, then there is a shortest ab-path with a single edge. The transfer relation in this special case therefore boils down to

$$(a@\tau_a \rightarrow b@\tau_b) \iff \exists f \in F : \tau_b - \tau_a \geq f_{\text{dur}} \text{ and } a = f_{\text{dep\_stop}} \text{ and } b = f_{\text{arr\_stop}}$$

which allows us to limit our searches to single-edge paths. These restrictions come at a price. In each connected component there is a quadratic number of edges because of the transitive closure. As a quadratic memory consumption is prohibitive in practice, we can therefore have no large components.

Our footpath-based transfer model is transitive, i.e., if $a@\tau_a \rightarrow b@\tau_b$ and $b@\tau_b \rightarrow c@\tau_c$ then $a@\tau_a \rightarrow c@\tau_c$. We exploit this property in our algorithms. While transfer model transitivity sounds like a very reasonable and desirable property, there is a common class of competitor transfer models that do not have it. They are similar to our model, except that instead of requiring transitive closure and triangle inequality, they limit the maximum path length by some constant m. It is possible that one can walk within time m from a to b and within time m from b to c but require longer than time m to get from a to c, which demonstrates that transitivity breaks. The missing transitivity is the main reason why we chose a different model.

An interesting special case are *loops* in the footpath graph. Without a loop at a stop s, a traveler cannot exit at s and enter another train at s. In practice, all stops have therefore loops. The duration of the loop footpath at stop s is called the change time[1] s^{change}. Some competitor works even assume that there are no footpaths beside these loops, which is a significant restriction compared to our model. Footpaths that are not loops are *interstop footpaths*.

Our transfer model is in general not reflexive, i.e., it is possible that there are stops s and time points τ such that $s@\tau \nrightarrow s@\tau$. However, one can study the special case of reflexive transfer models. This requirement translates to every stop having a change time of 0. The London benchmark instance of [52], which we also use, has this additional property.

Examples. In our Germany instance, the Karlsruhe main station is modeled as two stops. There is a stop that represents the main tracks used by the long distance trains. Further, there is a stop that represents the tracks where the local trams halt. Both are connected using a footpath per direction. Further, both stops have loop footpaths. The loop of the main track stop has a duration of 5min and the loop of the local tram stop has a duration of 4min. The footpaths between the two stops have a duration of 6min.

Transferring between local trams is therefore possible within 4min. To transfer between long distance trains, the traveler needs 5min. Finally, to transfer from tram to long distance train 6min are needed.

Other main stations are modeled using more stops. For example many stations have an additional stop per subway line.

Within cities, it can make sense to insert footpaths between neighboring tram stops. However, one has to be careful not to create large connected components in the footpath graph by doing so.

It is also possible to model stations in greater detail using a stop per platform. The London instance uses this approach. This approach gives more precise transfer times at the expense of more stops. Fortunately, the so obtained timetables usually have a reflexive transfer model.

[1]Several other works refer to s^{change} as minimum change time.

6.2.2 Journeys

A journey describes how a passenger can travel through a timetable network. They are composed of legs, which are pairs of connections $(l^i_{\text{enter}}, l^i_{\text{exit}})$ within the same trip. l^i_{enter} must appear before l^i_{exit} in the trip. Formally, a journey consists of alternating sequence of legs and footpaths $f^0, l^0, f^1, l^1 \ldots f^{k-1}, l^{k-1}, f^k$. A journey must start and end with a footpath. All intermediate transfers must be feasible according to the transfer model, i.e., for all i, $(l^{i-1}_{\text{exit}})_{\text{arr\_stop}}@(l^{i-1}_{\text{exit}})_{\text{arr\_time}} \rightarrow (l^i_{\text{enter}})_{\text{dep\_stop}}@(l^i_{\text{enter}})_{\text{dep\_time}}$ must hold. We refer to f^0 as *initial footpath* and to f^k as *final footpath*. The remaining footpaths are called *transfer footpaths*. Further, for a journey j we refer to $f^0_{\text{dep\_stop}}$ as *j's departure stop*, to $f^k_{\text{arr\_stop}}$ as the *j's arrival stop*, to $(l^0_{\text{enter}})\text{dep\_time} - f^0_{\text{dur}}$ as *j's departure time*, to $(l^{k-1}_{\text{exit}})\text{arr\_time} + f^k_{\text{dur}}$ *as j's arrival time*, and to k as *j's number of legs*. We also use j_{leg} to refer to the number of legs, i.e., k. Finally, we refer to $j_{\text{arr\_time}} - j_{\text{dep\_time}}$ as *j's travel time*. Formally, journeys are allowed to consist of a single footpath and no leg. However, we forbid this special case in certain problem settings to avoid unnecessary, simple but cumbersome special cases in our algorithms.

A journey j that is missing its initial footpath, i.e., a sequence $l^0, f^1, l^1 \ldots f^{k-1}, l^{k-1}, f^k$ is called a *partial journey*. We say that j departs in the connection l^0_{enter}.

The number of legs and the number of transfers differ slightly. For every journey with at least one leg, the number of transfers is $j_{\text{leg}} - 1$. The numbers are therefore essentially the same, except for a subtle difference. A journey without leg has 0 legs but also has 0 transfers and not -1 transfers. Counting legs eliminates some special cases in our algorithms and avoids some -1/+1-operations. Hence, for simplicity, we count legs.

6.2.3 Considered Problem Settings

In this section, we describe most problem settings studied in this paper. Several of these problems are defined in terms of Pareto-optimization. We therefore first recapitulate the definition of and domination and then state the problems considered in our paper. Section 9 introduces another problem setting called Minimum Expected Arrival Time problem. As its details are more involved, we introduce the problem setting in its own section.

Definition. A tuple x *dominates* a tuple y if there is no component in which y is strictly smaller than x and there is a component in which x is strictly smaller than y.

Pareto-optimal is defined in terms of domination.

Definition. Denote by P a multi-set of n-dimensional with scalar components. A tuple x is *Pareto-optimal* with respect to P, if no other tuple $y \in P$ exists, such that y dominates x.

In our setting, the tuples are journey attributes such as a journey's travel time. P is the set of attribute-tuples of all journeys.

The easiest problem, that we consider, asks when a traveler will arrive the earliest possible. Formally, it can be stated as follows:

Earliest Arrival Problem

Input: Timetable, source stop s, target stop t, source time τ

Output: The minimum arrival time over all journeys that depart after τ at s and arrive at t.

While simple, the earliest arrival problem has several downsides. For one, a traveler often does not have a fixed departure time, but is flexible and has a range of possible departure times. One can resolve this issue by iteratively solving the earliest arrival problem with varying source times. Fortunately, we can do better and therefore formalize the aggregated problem as follows:

Earliest Arrival Profile Problem

Input: Timetable, source stop s, target stop t, minimum departure time τ_s, maximum arrival time τ_t

Output: The set of all $(j_{\text{dep\_time}}, j_{\text{arr\_time}})$ over journeys j such that

- j departs not before τ_s at s,
- j arrives not after τ_t at t,
- the pair $(-j_{\text{dep\_time}}, j_{\text{arr\_time}})$ is Pareto-optimal among all journeys, and
- j contains at least one leg.

The result of the profile problem can be represented using a plot such as the one in Figure 6.1. The result is a compact representation of the functions that maps a departure time at s onto the earliest arrival time at t. We refer to this function as *profile function*. Formulated differently, the profile problem asks to simultaneously solve the earliest arrival problem for all source times.

We require j to have at least one leg, to be able to guarantee that the profile function is a step function. Dropping this restriction can break this property if s and t are connected via a footpath f. At least in our setting, handling such a situation is trivial but requires special case handling in our algorithm. To simplify our descriptions and to focus on the algorithmically interesting aspects, we decided to forbid journeys without leg.

An issue common with the earliest arrival problem and with its profile counterpart is that solely optimizing arrival time can lead to very absurd but "optimal" journeys. For example, Figure 6.2 depicts a journey that is "optimal" with respect to its arrival

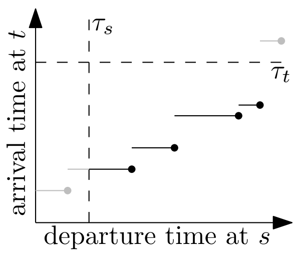


Figure 6.1: Profile function that maps the departure times at a stop s onto the arrival times at stop t. The black dots represent the solution to the earliest arrival profile problem. Only the black part needs to be computed. The grey part is excluded by the minimum departure time or maximum arrival time.

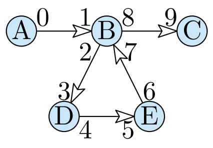


Figure 6.2: Example for an "optimal" journey that visits a stop twice. Circles depict stops, arrows depict connections and are annotated with their departure and arrival times. The journey $A \rightsquigarrow B \rightsquigarrow D \rightsquigarrow E \rightsquigarrow B \rightsquigarrow C$ visits stop B twice and has a minimum arrival time. The journey $A \rightsquigarrow B \rightsquigarrow C$ has the same arrival time but uses fewer legs.

time but visits a stop twice. Similarly, "optimal" journeys exist that enter a trip multiple times. When computing earliest arrival journeys and not just their arrival time, one therefore usually also requires that the journeys visit no stop or trip twice.

A simple solution to this problem consists of picking among all journeys with a minimum arrival time one that minimizes the number legs. This implies that no stop or trip is used twice. We say that the first optimization criterion is arrival time and the second criterion is the number of legs. This slight change is enough to guarantee that no stop is visited twice.

While this small change solves many transfer-related problems, some remain. Suppose, for example that there are two journeys whose arrival times differ by one second but the earlier one needs significantly more legs. In this case, one would like to pick the journey that arrives slightly later. This problem can be mitigated by rounding the arrival times at the target stop. However, in many application one wants to find both journeys. We therefore also consider the following problem setting.

Pareto Profile Problem

Input: Timetable, source stop s, target stop t, minimum departure time τ_s, maximum arrival time τ_t, maximum number of legs $\max_{\text{leg}}$

Output: The set of all $(j_{\text{dep\_time}}, j_{\text{arr\_time}}, j_{\text{leg}})$ over journeys j such that

- j departs not before τ_s at s,
- j arrives not after τ_t at t,
- j has at most $\max_{\text{leg}}$ legs,
- the pair $(-j_{\text{dep\_time}}, j_{\text{arr\_time}}, j_{\text{leg}})$ is Pareto-optimal among all journeys, and
- j contains at least one leg.

Besides the profile problem setting, we also consider *range problem* variants. In these, we set τ_t to $\tau_s + 2 \cdot (x - \tau_s)$, where x is the earliest arrival time. Formulated differently, we are only interested in journeys that are at most two times as long as possible. The solution to the range problems is a subset of the solution to the profile problems. The range problems can therefore often be solved faster. Fortunately, travelers usually do not want to arrive significantly later than the earliest arrival time. The solution to the range problem thus often consists of the journeys that actually interest a traveler. The range problem special cases are therefore of high practical relevance.

Beside determining the attributes of optimal journeys, i.e., departure time, arrival time, and number of legs, we also consider the problem of computing corresponding journeys in Sections 6.3.2 and 7.6. Optimal journeys are usually not unique. There usually are multiple journeys for a specific combination of departure time, arrival time, and number of legs. We regard all of them as being equal and only extract one of them. Extracting all journeys for a specific combination is a different problem setting.

6.3 Earliest Arrival Connection Scan

In this section, we describe the earliest arrival Connection Scan variant. It assumes that the connections are stored as array of quintuples that are sorted by departure time. Further, the footpaths must be stored in a data structure that allows an efficient iteration over the incoming and outgoing footpaths of a stop, such as for example an adjacency array. Similar to Dijkstra's algorithm, CSA maintains a tentative arrival time array, that stores for each stop the earliest known arrival time. A connection is called *reachable* if there is a way for the traveler to sit in the connection. Contrary to Dijkstra's algorithm, ours does not employ a priority queue. Instead, it iterates over all connections by departure time. The algorithm tests for every connection

```
1 for all stops x do $S[x] \leftarrow \infty$;
2 for all trips x do reset $T[x]$;
3 for all footpaths f from s do $S[f_{\text{arr\_stop}}] \leftarrow \tau + f_{\text{dur}}$;
4 for all connections c increasing by $c_{\text{dep\_time}}$ do
5     if $T[c_{\text{trip}}]$ is set or $S[c_{\text{dep\_stop}}] \le c_{\text{dep\_time}}$ then
6         raise $T[c_{\text{trip}}]$;
7         for all footpaths f from $c_{\text{arr\_stop}}$ do
8             $S[f_{\text{arr\_stop}}] \leftarrow \min\{S[f_{\text{arr\_stop}}], c_{\text{arr\_time}} + f_{\text{dur}}\}$;
```

Algorithm 6.3: Unoptimized earliest arrival Connection Scan algorithm. s is the source stop and τ the source time.

whether it is reachable. For each reachable connection, the algorithm adjusts the tentative arrival times of the stops reachable by foot from the connection's arrival stop. After the execution of our algorithm, the output is t's tentative arrival time. Contrary to most adaptations of Dijkstra's algorithm, our algorithm touches more connections. But the work required per connection does not involve a priority queue operation and is therefore significantly faster.

Our algorithm maintains two arrays S and T. The array S stores for every stop the tentative arrival time. The array T stores for every trip a bit indicating whether the traveler was able to reach any of the connections in the trip. Testing whether a connection c is reachable boils down to testing, whether $S[c_{\text{dep\_stop}}] \le c_{\text{dep\_time}}$ or $T[c_{\text{trip}}]$ is set. To adjust the tentative arrival times, our algorithm relaxes all footpaths outgoing from $c_{\text{arr\_stop}}$. The algorithm is described in pseudo-code form in Algorithm 6.3.

6.3.1 Optimizations

In this subsection, we describe three optimizations to the earliest arrival Connection Scan algorithm. Algorithm 6.4 presents pseudo-code that incorporates all three optimizations. In the following, c always denotes the connection currently being processed.

Stopping criterion. We can abort the execution of the algorithm as soon as $S[t] \le c_{\text{dep\_time}}$. This is correct because processing a connection c never assigns a value below $c_{\text{dep\_time}}$ to any tentative arrival time. Further, as we process the connections increasing by $c_{\text{dep\_time}}$, it follows that $S[t]$ will not be changed by our algorithm after the inequality holds.

Starting criterion. No connection departing before the source time τ is reachable, as for every journey j, $\tau \le j_{\text{dep\_time}} < j_{\text{arr\_time}}$ must hold. The proposed optimization ex-

```
1  for all stops $x$ do $S[x] \leftarrow \infty$;
2  for all trips $x$ do reset $T[x]$;
3  for all footpaths $f$ from $s$ do $S[f_{\text{arr\_stop}}] \leftarrow \tau + f_{\text{dur}}$;
4  Find first connection $c^0$ departing not before $\tau$ using a binary search;
5  for all connections $c$ increasing by $c_{\text{dep\_time}}$ starting at $c^0$ do
6      if $S[t] \leq c_{\text{dep\_time}}$ then
7          Algorithm is finished;
8      if $T[c_{\text{trip}}]$ is set or $S[c_{\text{dep\_stop}}] \leq c_{\text{dep\_time}}$ then
9          raise $T[c_{\text{trip}}]$;
10         if $c_{\text{arr\_time}} < S[c_{\text{arr\_stop}}]$ then
11             for all footpaths $f$ from $c_{\text{arr\_stop}}$ do
12                 $S[f_{\text{arr\_stop}}] \leftarrow \min\{S[f_{\text{arr\_stop}}], c_{\text{arr\_time}} + f_{\text{dur}}\}$;
```

Algorithm 6.4: Optimized earliest arrival Connection Scan algorithm. s is the source stop, τ the source time, and t that target stop.

ploits this. It runs a binary search to determine the first connection c^0 departing no later than τ. The iteration is started from c^0 instead of the first connection in the timetable.

Limited Walking. If $S[c_{\text{arr\_stop}}]$ cannot be improved even with an instant transfer, i.e., $S[c_{\text{arr\_stop}}] \leq c_{\text{arr\_time}}$ holds, then no tentative arrival time can be improved. The optimization consist of not iterating over the outgoing footpaths of $c_{\text{arr\_stop}}$ in this case.

The correctness of this optimization relies on the transitivity of the transfer model. Denote by $y = c_{\text{arr\_stop}}$. As $S[y] \neq \infty$ a journey j ending at y has already been found. Denote by f^{xy} the last footpath of j departing at x. It is possible that $x = y$ and that f^{xy} is a loop. For every outgoing footpath f^{yz} of y to some stop z, there exists a footpath f^{xz} from x to z such that $f^{xz}_{\text{dur}} \leq f^{xy}_{\text{dur}} + f^{yz}_{\text{dur}}$. We can replace the last footpath of j by f^{xz} and have obtained a journey arriving at z no later than the journey involving c. As this argumentation works for all outgoing footpaths, no tentative arrival time can be improved. Iterating over the outgoing footpaths is therefore superfluous. The optimization is thus correct.

The limited walking optimization crucially depends on the transitivity of the transfer model. For example, it does not hold for a transfer model with a maximum path length. Consider the example depicted in Figure 6.5. Assume that both c^A and c^B are reachable connections. When processing the connection c^A arriving at A the tentative arrival time at B is set to 8:07. However, the tentative arrival time at C remains ∞ as the path is too long. Because the tentative arrival time at B is smaller than 9:00 the limited

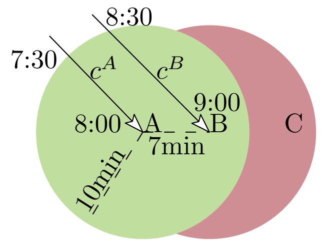


Figure 6.5: Counterexample for the correctness of limited walking optimization in combination with a maximum path length transfer model. The example includes three stops A, B, and C, two connections c^A, and c^B with annotated departure and arrival times, and walking radii of 10min. The green area is reachable by foot from A. The red area is reachable by foot from B but not from A.

walking optimization activates when processing c^B. The tentative arrival time at C therefore remains at ∞, which is clearly incorrect.

6.3.2 Journey Extraction

The algorithm described in the previous section only computes the earliest arrival time. In this section, we describe how to compute an earliest arrival journey in a post processing step. Our algorithm guarantees that the extracted journey visits no stop nor trip twice.

The algorithm comes in two variants. The first variant augments the data structures used during the Connection Scan with additional *journey pointers* that can be used to reconstruct a journey. The second variant leaves the earliest arrival scan untouched but needs to perform more complex tasks to reconstruct an earliest arrival journey. The trade-off between the two variants is that the former is conceptually slightly more straight-forward and therefore easier to implement. Further, the former has a lower extraction running time, which comes at the cost of a higher scan running time. Finally, the later requires additional data structures, which must be computed in a fast preprocessing step. If only a journey towards one target stop should be extracted, then the later variant is faster. If journeys from one source stop to many target stops should be extracted, the former can be faster.

6.3.2.1 With Journey Pointers

Algorithm 6.6 stores for every stop x a triple $J[x]$ of final enter connection, final exit connection, and final footpath of an earliest arrival journey towards x. We refer to this triple as *journey pointer*. If no optimal journey exists, then the journey pointer is set to an invalid value.

```
for all stops x do $S[x] \leftarrow \infty$;
for all trips x do $T[x] \leftarrow \perp$;
for all stops x do $J[x] \leftarrow (\perp, \perp, \perp)$;
for all footpaths f from s do $S[f_{\text{arr\_stop}}] \leftarrow \tau + f_{\text{dur}}$;
for all connections c increasing by $c_{\text{dep\_time}}$ do
    if $T[c_{\text{trip}}] \neq \perp$ is set or $S[c_{\text{dep\_stop}}] \leq c_{\text{dep\_time}}$ then
        if $T[c_{\text{trip}}] = \perp$ then
            $T[c_{\text{trip}}] \leftarrow c$;
        for all footpaths f from $c_{\text{arr\_stop}}$ do
            if $c_{\text{arr\_time}} + f_{\text{dur}} < S[f_{\text{arr\_stop}}]$ then
                $S[f_{\text{arr\_stop}}] \leftarrow c_{\text{arr\_time}} + f_{\text{dur}}$;
                $J[f_{\text{arr\_stop}}] \leftarrow (T[c_{\text{trip}}], c, f)$;
$j \leftarrow \{\}$;
while $J[t] \neq (\perp, \perp, \perp)$ do
    Prepend $j$ with $J[t]$;
    $t \leftarrow J[t]^{\text{enter}}_{\text{dep\_stop}}$;
Prepend $j$ with the footpath from $s$ to $t$;
Output $j$;
```

Algorithm 6.6: Earliest arrival Connection Scan algorithm with journey extraction. s is the source stop. τ the source time, and t the target stop.

A journey j from a source stop s to a target stop t can be constructed backwards. Initially j is empty. If t has a valid journey pointer, we the prepend t's journey pointer to j. Further, we set t to the departure stop of the journey pointer's enter connection and iterate. If t does not have a valid journey pointer, we prepend j with a footpath from s to t and the journey extraction terminates.

Journey Pointer Construction. When a tentative arrival time is modified, our algorithm stores a corresponding journey pointer. To this end, our algorithm must determine the three elements of the triple. Determining the exit connection and the final footpath is easy. These are the values denoted by the variables c and f in the code depicted in Algorithm 6.6. Computing the enter connection is more difficult.

We replace the bit array T in the base algorithm by an array that contains for every trip a connection ID. This connection ID indicates the earliest connection reachable in a trip. It may be invalid, if no connection was reached. The ID being valid corresponds to the bit being set in the base algorithm. We set an ID when a trip is first reached.

It remains to show that the extracted journey does not visit a stop or trip twice. A trip cannot be visited twice by the extracted journey because T is set to the first connection reachable in a trip. A stop cannot be visited twice as our algorithms stores at each stop the first journey pointer found towards it. Fortunately, a journey pointer leading to a journey with a loop cannot be the first.

6.3.2.2 Without Journey Pointers

A journey can be extracted without storing journey pointers. However, additional data structures are necessary.

Additional . Our algorithm needs to enumerate the connections in a trip that precede a given connection. We therefore construct an adjacency array that maps a trip ID onto the IDs of the connections in the trip. The connections are sorted by position in the trip. Our algorithm can thus enumerate all connections in a trip rapidly and stop once the given connection is found.

Further, our algorithm needs to enumerate the connections arriving at a given stop at a given timepoint. We therefore construct a second adjacency array that maps a stop ID onto the IDs of the connections arriving at the stop. The connections are sorted by arrival time. We can use a binary search to efficiently enumerate all requested connections.

Extraction. Our algorithm works similar to the one using journey pointers. However, the journey pointer is generated on the fly. We therefore need a subroutine to determine a triple of enter connection c^{enter}, exit connection c^{exit}, and final footpath f. We start by constructing a set of candidates for c^{exit}. This set is then pruned. Finally, our algorithm iterates over the candidates and tries to find a corresponding c^{enter}.

To generate the candidate set our algorithm enumerates all incoming footpaths f of the stop t. For every f, all connections arriving at $f_{\text{dep\_stop}}$ at $S[t] - f_{\text{dur}}$ are added to the candidate set.

A candidate can only be a valid c^{exit} if it is reachable. If it is reachable then the trip bit must be set. We can therefore prune all candidate connections c for which $T[c_{\text{trip}}]$ is false. The bit being set does not imply that the candidate is reachable. It is also possible that only a later connection in the same trip is reachable.

Finally, our algorithm iterates over the remaining candidates c. For each candidate, it enumerates all connections x in c_{trip} not after c. It then checks whether $S[x_{\text{dep\_stop}}] \leq S[x_{\text{dep\_time}}]$. If it holds, then x is a valid enter connection, c a valid exit connection and f a valid final footpath. Further, as x is the first connection in the trip, the extracted journey cannot visit a trip twice. As our approach constructs a journey for the earliest timepoint where t is reachable, we can guarantee that the extracted journey does not visit a stop twice.

Instance	Stops	Connections	Trips	Routes	Interstop Footpaths
Germany	252 374	46 218 148	2 395 656	248 261	103 535
London	20 843	4 850 431	125 537	2 135	45 652

Table 6.7: Instance sizes.

If no journey pointer can be generated, then t was reached by foot from s. This corresponds to $J[t]$ being invalid in Algorithm 6.3.

6.4 Experiments

We experimentally evaluate the earliest arrival Connection Scan Algorithm and compare it with competing algorithms. Besides only measuring the query running times, we also report how much time is needed to setup the data structures. The setup time is an upper bound to the time needed to update a timetable.

The section is structured as follows: We first describe the machines on which we run our experiments. We then describe the test instances and how we generate our test queries. Afterwards, we report the running times needed by the Connection Scan Algorithm. Finally, we compare the achieved running times with related work.

6.4.1 Experimental Setup

Machine. Unless specified otherwise, we ran all experiments on a single pinned thread of an Intel Xeon E5-1630v3, with 10 MiB of L3 cache and 128 GiB of DDR4-2133MHz. This is a CPU with Haswell architecture. Some experiments were executed on an older dual 8-core Intel Xeon E5-2670, with 20 MiB of L3 cache and 64 GiB of DDR3-1600 RAM, a CPU with Sandy Bridge architecture. Hyperthreading was deactivated in all experiments. Our implementation is written in C++ and is compiled using g++ 4.8.4 with the optimization flags `-O3 -march=native`.

Instances. We performed our experiments on two main benchmark instances. Table 6.7 reports the sizes. The first instance is based on the data of `bahn.de` during winter 2011/2012. The data was provided to use by Deutsche Bahn (DB), the German national railway company. We thank DB for making this data accessible to us for research purposes. The data contains European long distance trains, German local trains, and many buses inside of Germany. The data includes vehicles of local operators DB. The raw data contains for every vehicle a day of operation. Unfortunately, no day exists at which every local operator operates. The planning horizon of some operators ends before the reported data of other operators begins. To avoid holes in our timetable, we therefore extract all trips regardless of their day of operation and assume that they depart within the first day. Our extracted instance contains therefore more connections per day than the instance in productive use. Further, to support night trains, we consider two successive identical days. The raw data contains footpaths. We did not generate additional ones based upon geographic positions but did add footpaths to make the graph transitively closed. We removed data errors such as exactly duplicated trips, vehicles driving at more than 300 km/h or footpaths at more than 50 km/h.

The second instance is based on open data made available by Transport for London (TfL). The raw input data is available in the London data store [100]. We thank TfL for making this data openly available. The data includes tube (subway), bus, tram, Dockland Light Rail (DLR). The data corresponds to a Tuesday of the periodic summer schedule of 2011. In contrast to the Germany instance, the London instance thus only contains data for a single day. Stops correspond to platforms in this data set. As a consequence all change times are zero, i.e., the transfer model is reflexive. This data set is the main instance used in [51], one of our main competitor algorithms. We removed some obvious data errors from the data. The instance sizes we report are therefore slightly smaller than in [51].

Test Query Generation. To evaluate our algorithms, we generate random test queries. The source and target stops are chosen uniformly at random. The source time is chosen uniformly at random within the first 24 hours. Unless noted otherwise, all reported running times are averaged over 10^4 queries.

6.4.2 Earliest Arrival Connection Scan

We experimentally evaluated the earliest arrival Connection Scan algorithm and report the average running time in Table 6.8. We successively activate the proposed optimizations. Further, we evaluate the running time of the journey extraction without journey pointers.

The start and stop criteria drastically reduce the running times. The explanation is that significantly fewer connections have to be scanned. On the London instance, the speedup is 15 times whereas the speedup on the Germany instance is “only” 5. This is due to the differences in journey lengths. In London a traveler needs on average less time to traverse the whole network than in Germany. The stop criterion therefore activates sooner reducing the number of scanned connections. The limited walking optimization further reduces running times by 1.5 to 2.0 times.

Finally, we report the running time needed to perform a journey extraction in addition to the earliest arrival Connection Scan. As we only extract a single journey per scan, we use the extraction process that does not store journey pointers. The extraction process is faster than the scan. On the Germany instance, it only needs about and on the London instance 0.1ms.

Instance	Start Crit.	Stop Crit.	Limited Walk.	Journey Extraction	Running Time [ms]
Germany	○	○	○	○	329.0
Germany	●	○	○	○	298.9
Germany	●	●	○	○	67.9
Germany	●	●	●	○	44.9
Germany	●	●	●	●	47.1
London	○	○	○	○	41.2
London	●	○	○	○	37.9
London	●	●	○	○	2.7
London	●	●	●	○	1.2
London	●	●	●	●	1.3

Table 6.8: Earliest arrival Connection Scan running times.

Instance	Sort [s]	Journey [s]
Germany	3.56	6.15
London	0.35	0.39

Table 6.9: Datastructure construction running time averaged over 100 runs. “Sort” is the time needed to sort the connection array by departure time. “Journey” is the additional time needed to construct the journey extraction data structures.

6.4.3 Datastructure Construction

In Table 6.9, we report the running time needed to sort the connection array and the running time needed to construct the journey extraction data structures. To avoid accelerating the sort algorithm by providing it with nearly sorted data, we randomly permute the array before sorting it. We use GCC’s std::sort implementation. If the timetable significantly changes, then these two steps need to be rerun. If the changes are only small, then it is probably faster to patch the existing data structures.

In practice, when delays , the operator needs to simulate how the delay propagates through the network. This propagation is in practice probably slower than the few seconds needed to construct the data structures needed by our algorithms.

Instance	Algorithm	Pareto	Running Time [ms]
Germany	TED	○	1 996.6
Germany	TD	○	448.5
Germany	TD-col	○	163.3
Germany	RAPTOR	●	325.8
Germany	CSA	○	44.9
London	TED	○	29.3
London	TD	○	9.5
London	TD-col	○	3.7
London	RAPTOR	●	6.4
London	CSA	○	1.2

Table 6.10: Comparison with related work with respect to the earliest arrival time problem.

6.4.4 Comparison with Related Work

In Table 6.10, we compare our algorithms with related work. The employed implementations are based upon the code of [52]. All competitors are run with stopping criterion active.

We compare the Connection Scan algorithm's running times against three extensions of Dijkstra's algorithm and RAPTOR. The first extension is based on a time-expanded graph model. The second uses a time-dependent graph model. We refer to [112] for a detailed exposition of these models. The third uses an optimized time-dependent graph model, proposed in [49], that merges nodes based on colored timetable elements. Finally, we compare against RAPTOR [52], an algorithm that does not employ a graph based model. Instead, it operates directly on the timetable, similarly to the Connection Scan algorithm.

We experimentally compare the performance of the algorithms with respect to the earliest arrival time problem. However, RAPTOR does not fit precisely into this category. It is designed in a way that inherently optimizes the number of transfers in the Pareto-sense. It can and must thus solve a more general problem. It does not benefit from restricting the problem setting. We therefore report its running times alongside the other earliest arrival time algorithms.

Table 6.10 shows that the non-graph based algorithms clearly dominate the base versions of the time-dependent and time-expanded extensions of Dijkstra's algorithm. The time-dependent extension can be engineered to be about a factor of 2 faster than RAPTOR. The Connection Scan algorithm is faster than all of the competitors.

6.5 Chapter Conclusion

CSA enables answering earliest arrival time queries in mere milliseconds. A corresponding earliest arrival journey can be extracted afterwards in a nearly negligible amount of addition query running time. Even on the large Germany network with integrated local transit, average query running times are below 50ms. The data structures can be constructed in less than 10 seconds even for the large Germany instance. This enables an easy straightforward and fast integration of realtime train delays.

7 Profile Connection Scan

The Connection Scan algorithm can be extended to solve the profile problem variants. The algorithm is very flexible and, compared with many other algorithms to solve the profile problem, comparatively easy.

We first present the algorithm on a very high level in the form of an abstract framework. Afterwards, we illustrate how this framework can be used to solve the various profile problem variants. We start with a very restricted problem setting to simplify the exposition. We then extend the algorithm, iteratively dropping these restrictions. The initial simplifications are:

- The time horizon is unbounded, i.e., there is no minimum departure nor maximum arrival time in the input.
- We solve the all-to-one problem, i.e., there the input contains only a target stop and the profile functions from every stop to this target should be computed.
- We assume that there are no interstop footpaths, i.e., there are only change times, i.e., there are only loops in the footpath graph.
- We solve the earliest arrival profile problem, i.e., we do not optimize the number of transfers.

7.1 Framework

Algorithm 7.1 depicts the high level framework of all Connection Scan based profile algorithms. Understanding this structure is crucial to understand any of the algorithms. At its core, the algorithm uses dynamic programming. It constructs journeys from late to early and exploits that an early journey can only have later journeys as subjourneys. Further, it exploits the observation that a traveler sitting in a connection only has three options to continue his journey. The three options to continue his journey are:

- The traveler can exit the train and, if there is a footpath to the target, walk there, or
- he can remain seated reaching the next connection in the trip, if there is a next connection, or

```
1 for all stops $x$ do Initialize stop data structure $S[x]$;
2 for all trips $x$ do Initialize trip data structure $T[x]$;
3 for connections $c$ decreasing by $c_{\text{dep\_time}}$ do
      /* 1. Determine arrival time when starting in $c$ */
4     $\tau_1 \leftarrow$ arrival time when walking to the target;
5     $\tau_2 \leftarrow$ arrival time when remaining seated, uses $T[c_{\text{trip}}]$;
6     $\tau_3 \leftarrow$ arrival time when transferring, uses $S[c_{\text{arr\_stop}}]$;
      /* $\tau_c$ = arrival time when starting in $c$ */
7     $\tau_c \leftarrow \min\{\tau_1, \tau_2, \tau_3\}$;
      /* 2. Incorporate $\tau_c$ into the data structures */
8     Incorporate $\tau_c$ into $S[x]$ for all stops $x$ with footpath $(x, c_{\text{dep\_stop}})$;
9     Incorporate $\tau_c$ into $T[c_{\text{trip}}]$;
```

Algorithm 7.1: Pseudo-Code of the Connection Scan profile framework.

- he can exit the train and use a footpath towards some other stop and enter another train.

The two ways how a traveler can have reached a connection c are:

- He can have been sitting in the train, i.e, he reached a connection before c in the same trip, or
- or he entered the train at $c_{\text{dep\_stop}}$ proceeded by a footpath.

The algorithm scans the connections by departure time. In the following, we always use the letter c to indicate the connection currently being scanned. The algorithm stores at each stop x a profile from x to the target t and at each trip the earliest arrival time over all partial journeys departing in a connection of the trip. The algorithm's structure is depicted in the pseudo-code of Algorithm 7.1 which mirrors this high level description very closely.

7.2 Earliest Arrival Profile Algorithm without Interstop Footpaths

Algorithm 7.1 contains the pseudo-code of the basic Connection Scan profile framework. In this section we describe, how to instantiate this framework to obtain an algorithm to solve the earliest arrival profile algorithm. The pseudo-code of the instantiated algorithm is depicted in Algorithm 7.2.

```
1  for all stops x do $S[x] \leftarrow \{(\infty,\infty)\}$;
2  for all trips x do $T[x] \leftarrow \infty$;
3  for connections c decreasing by $c_{\text{dep\_time}}$ do
4      if $c_{\text{arr\_stop}}$ = target then
5          $\tau_1 \leftarrow c_{\text{arr\_time}} + c^{\text{change}}_{\text{arr\_stop}}$
6      else
7          $\tau_1 \leftarrow \infty$
8      $\tau_2 \leftarrow T[c_{\text{trip}}]$;
9      $p \leftarrow$ earliest pair of $S[c_{\text{arr\_stop}}]$;
10     while $p_{\text{dep\_time}} < c_{\text{arr\_time}}$ do
11         $p \leftarrow$ next earlier pair of $S[c_{\text{arr\_stop}}]$;
12     $\tau_3 \leftarrow p_{\text{arr\_time}}$;
13     $\tau_c \leftarrow \min\{\tau_1, \tau_2, \tau_3\}$;
14     $p \leftarrow (c_{\text{dep\_time}} - c^{\text{change}}_{\text{dep\_stop}}, \tau_c)$;
15     $q \leftarrow$ earliest pair of $S[c_{\text{dep\_stop}}]$;
16     if q does not dominate p then
17         if $q_{\text{dep\_time}} \neq p_{\text{dep\_time}}$ then
18             Insert p at the front of $S[c_{\text{dep\_stop}}]$;
19         else
20             Replace q as the earliest pair of $S[c_{\text{dep\_stop}}]$ with p;
21     $T[c_{\text{trip}}] \leftarrow \tau_c$;
```

Algorithm 7.2: Earliest arrival Connection Scan profile algorithm without interstop footpaths.

We start by describing how the stop data structure S and the trip data structure T are implemented. Afterwards, we describe the operations that modify S and T.

For every trip, our algorithm stores one integer, i.e., T is an array of integers whose size is the number of trips. This number represents the earliest arrival time for the partial journey departing in the earliest scanned connection of the corresponding trip.

For every stop, we store a profile function. A function is stored as sorted array of pairs of departure and arrival times. This means that S is an array whose size is the number of stops. The elements of S are arrays with a dynamic size. The elements of these inner arrays are pairs of departure and arrival times. After the execution of the algorithm, $S[x]$ contains the xt-profile.

We initialize all elements of T with ∞ and all elements of S with a singleton array containing a (∞, ∞)-pair. This algorithm state encodes that all travel times are ∞, i.e., the traveler cannot get anywhere. This would also be the correct solution, if the timetable contained no connections. When scanning the connection c, we modify S and T to account for all journeys that use c. One can thus view our approach as maintaining profiles corresponding to the timetable consisting of only the latest connections. We start with no connection and iteratively add connections. The scanned connection c is the connection currently being added.

Scanning c consists of two parts. First, τ_c must be computed and then τ_c must be integrated into T and S. Computing τ_c consists of the already mentioned three subcases and the integration has two subcases. Luckily, most of these cases are trivial in the simplified problem variant considered here.

Computing the arrival time at the target τ_1 is trivial: Either c arrives at the target stop, in which case the arrival time is $c_{\text{arr\_time}} + c^{\text{change}}_{\text{arr\_stop}}$, or the target is unreachable, as there are no interstop footpaths. If the traveler remains seated, then his arrival time will be the same, as the arrival time if he was sitting in the next connection of the trip. This arrival time is stored in $T[c_{\text{trip}}]$. Incorporating τ_c into the trip data structure is also trivial, it consists of a single assignment: $T[c_{\text{trip}}] \leftarrow \tau_c$.

Slightly more complex are the incorporation of τ_c into the profile and the efficient computation of τ_3. Incorporating τ_c consists of adding the pair $p = (c_{\text{dep\_stop}} - c^{\text{change}}_{\text{dep\_stop}}, \tau_c)$ into the array if it is non-dominated. Because the connections are scanned decreasing by departure time, there cannot be a pair with an earlier departure time. However, there can be a pair with the same departure time. It is therefore sufficient for the domination test to look at the earliest pair q already in the array. If p is not dominated by q, we either add p or replace q, depending on whether the departure times are equal.

Evaluating the function is done by finding the pair p in the array $S[c_{\text{arr\_stop}}]$ with the earliest departure time no earlier than $c_{\text{arr\_time}}$. The arrival time of p is τ_3. As the array is sorted, the evaluation can be done in logarithmic running time using a binary search. However, as $c_{\text{arr\_time}} - c_{\text{dep\_time}}$ is usually small in practice, the requested pair is usually near the beginning of the array. A sequential search is therefore faster in practice.

7.3 Optimizations

Several optimizations exist for the Connection Scan profile algorithm. The first optimization, that we describe exploits a hardware feature called prefetching. The next three optimizations exploit that in most cases we do not want to compute journeys from every stop to the target. They exploit additional information in the input such as the source stop to accelerate the computation.

Memory Prefetching. The Connection Scan profile algorithm can be slightly accelerated by using processor memory prefetch instructions. Modern processors are capable of detecting simple memory access patterns and to fetch data sufficiently early to hide memory access latency. The sequential scan over the connection array is an example of such a simple memory access pattern. However, detecting the stop profile access is more complex. When scanning the c-th connection, we therefore execute prefetch instructions for the stop profiles $S[(c-4)_{\text{dep\_stop}}]$, $S[(c-4)_{\text{arr\_stop}}]$, and the trip arrival time $T[(c-4)_{\text{trip}}]$. These instructions help hide memory latency by overlapping the processing of connection c with the memory fetching of the four connections $c-4$, $c-3$, $c-2$, and $c-1$.

Bounded Time-Horizon. The minimum departure time τ_s and maximum arrival time τ_t can exploited by only scanning connections c with $\tau_s \leq c_{\text{dep\_time}} \leq \tau_t$. The earliest connection can be determined using a binary search. To determine the latest connection, another binary search can be used. However, it is also a byproduct of the next optimization.

Scanning only Reachable Trips. The source stop and source times can be exploited by running a non-profile earliest arrival scan before the profile scan. The objective of this initial scan is to determine, which trips are reachable. If a trip is not reachable, then no connection in it can be reachable. We do not have to scan non-reachable connections as they cannot influence the profile at the source stop. We can thus skip connections for which the trip bit is not set. An efficient implementation starts by finding the first connection departing not before τ_s using a binary search. It then performs the earliest arrival scan increasing by departure time until a connection departing after τ_t is encountered. The same connections are then scanned in the reverse order in the profile scan.

Source Domination. The source stop can be exploited in another way. In the profile framework depicted in Algorithm 7.1, scanning a connection consists of two parts. The first part determines the arrival time when sitting in the connection τ_c. The second part incorporates τ_c into the data structures. Consider the pair $p = (c_{\text{dep\_time}}, \tau_c)$. If p is dominated by the pairs in the profile of the source stop, then the second part can be skipped. This optimization is correct because every journey starting at the source stop and using c would be dominated.

It remains to describe, how to efficiently implement the domination test. For the test, we need to know the arrival time of the earliest pair q in the profile of the source stop such that $q_{\text{dep\_time}} \geq c_{\text{dep\_time}}$. This information can be obtained by evaluating the source stop's profile. However, as the connections are scanned decreasing by departure time, we can do better by maintaining a pointer to the relevant pair in the source stop's profile. When scanning a connection, our algorithm first decreases the pointer if necessary and then looks up the arrival time. As the pointer can only be decreased as often as there are pairs in the source stop's profile, we can bound the running time needed to perform these evaluations by the size of the source stop's profile.

7.4 Interstop Footpaths

In this section, we expand the profile algorithm to handle interstop footpaths. Initial and transfer footpaths are handled in the same way, but a different strategy is needed for final footpaths. We start our description with final footpaths, as the idea is simpler. This algorithm variant is presented in Algorithm 7.3.

Final Footpaths. Handling final footpaths consists of modifying the computation of τ_1 in the framework of Algorithm 7.1. In the base algorithm, the traveler can only arrive at the target by train. In the extended version, he can also walk at the end. For this extension, we add a new array of integers . It stores for every stop the walking distance to the target or ∞, if walking is not possible. Computing τ_1 for a connection c can be done in constant time by evaluation $c_{\text{arr\_time}} + D[c_{\text{arr\_stop}}]$.

For , we do not reset all elements of D for each query. Instead, we initialize all elements of D to ∞ during the algorithm setup. We do this initialization only once. Each query begins by iterating over the incoming footpaths of the target stop. It sets D to the appropriate values for all stops from which the traveler can transfer to the target. After the profile computation, our algorithms iterates a second time over the same footpaths to reset all values of D to ∞.

Transfer and Initial Footpaths. Our algorithm handles transfer and initial footpaths by iterating over the incoming footpaths f of $c_{\text{dep\_stop}}$ when incorporating τ_c into the profiles. It inserts a pair $p = (c_{\text{dep\_time}} - f_{\text{dur}}, \tau_c)$ into the profile of the stop $f_{\text{dep\_stop}}$, if p is not dominated in $f_{\text{dep\_stop}}$'s profile.

Unfortunately, we can no longer guarantee that the departure time of p will be the earliest in each profile. A slightly more complex insertion algorithm is therefore needed: Our algorithm temporarily removes pairs departing before the new pair. It then inserts p, if non-dominated, and then reinserts all previously removed pairs that are not dominated by p.

Limited Walking. If the number of interstop footpaths is large, handling transfer and initial footpaths can be computationally expensive. Especially the iteration over the incoming footpaths of $c_{\text{dep\_stop}}$ can be costly. Fortunately, the limited walking optimization can be adapted and can drastically reduce running time on some instances. The idea is as follows: If the pair $(c_{\text{dep\_time}}, \tau_c)$ is dominated in the profile of

```
   /* $D[x] \leftarrow \infty$ for every stop $x$ in a preprocessing step */
1  for all footpaths $f$ with $f_{\text{arr\_stop}}$ = target do ;
2  for all stops $x$ do $S[x] \leftarrow \{(\infty, \infty)\}$;
3  for all trips $x$ do $T[x] \leftarrow \infty$;
4  for connections $c$ decreasing by $c_{\text{dep\_time}}$ do
5      $\tau_1 \leftarrow c_{\text{arr\_time}} + D[c_{\text{arr\_stop}}]$;
6      $\tau_2 \leftarrow T[c_{\text{trip}}]$;
7      $\tau_3 \leftarrow$ evaluate $S[c_{\text{arr\_stop}}]$ at $c_{\text{arr\_time}}$;
8      $\tau_c \leftarrow \min\{\tau_1, \tau_2, \tau_3\}$;
9      if $(c_{\text{dep\_time}}, \tau_c)$ is non-dominated in profile of $S[c_{\text{arr\_stop}}]$ then
10         for all footpaths $f$ with $f_{\text{arr\_stop}} = c_{\text{dep\_stop}}$ do
11             Incorporate $(c_{\text{dep\_time}} - f_{\text{dur}}, \tau_c)$ into profile of $S[f_{\text{dep\_stop}}]$;
12     $T[c_{\text{trip}}] \leftarrow \tau_c$;
13 for all footpaths $f$ with $f_{\text{arr\_stop}}$ = target do $D[x] \leftarrow \infty$;
```

Algorithm 7.3: Earliest arrival Connection Scan profile algorithm with interstop footpaths and the limited walking optimization.

$c_{\text{dep\_stop}}$, then all pairs computed when scanning c are dominated. The correctness argument is essentially the same as for the non-profile algorithm. One can prefix the journey of the dominating pair with each footpath and obtain at each stop a pair that would dominate each of the pairs created during the scanning of c. We thus do not need to generate them as they would be dominated anyway, i.e., we do not need to iterate over the incoming footpaths.

Different Set of Footpaths for Initial and Final Footpaths. In our proposed transfer model, we only have one type of footpaths. However, many applications have an extended set of footpaths for the initial and final footpaths. In some applications the traveler can walk for a longer amount of time at the beginning or at the end of his journey than when changing trains. Further, some applications have source and target locations that are not stops but might, for example, be city districts. Luckily, our algorithm can easily be extended to handle these cases.

Final footpaths can be handled by iterating over the extended footpath set during the initialization of D. Handling initial footpaths is slightly more complex. Denote by s the source location, for which the profile should be computed. In a first step, we create a set X of pairs that may contain dominated entries. After removing the dominated entries, the profile of s is obtained.

Our algorithm starts by iterating over all outgoing extended footpaths f of s. For every pair (d, a) in the profile of $f_{\text{arr\_stop}}$, there is a $(d - f_{\text{dur}}, a)$ pair in X. After removing dominated pairs from X, the profile of s is obtained.

It is possible to generate the set of extended footpaths using Dijkstra's algorithm on the fly. We can therefore drop the requirement that the set of extended footpaths must be transitively closed. This allows us to have very long initial and final footpaths. Unfortunately, the restrictions still apply for transfer footpaths.

7.5 Optimizing the Number of Legs

In the previous section, we presented the basic Connection Scan profile algorithm and extended it to a footpath-based transfer model. Now, we further extend it to optimize the number of legs the arrival time. We present three ways to perform this optimization. The first and easiest approach optimizes the number of legs as a secondary criterion. The second approach is a refinement of the first that heuristically mitigates some of its problems. Finally, we present as third approach an extension that optimizes the number of legs and the arrival time in the Pareto-sense.

The overhead of the first two approaches over the basic algorithm is negligible. Unfortunately, the optimization in the Pareto-sense adds a significant overhead. We therefore recommend to the reader to first try the first two approaches and only use the third if it is really necessary for the particular application at hand.

Our algorithm optimizes the number of legs by counting the number of times a traveler exits a train. As there is an exit per leg, the number of exits and the number of legs coincide. The exit counter is increased each time that a profile is evaluated, i.e., during the computation of τ_3 in the framework.

7.5.1 Number of Legs as Secondary Criterion

Optimizing the number of legs as secondary criterion, i.e., computing a journey with a minimum number of legs among all journeys with a minimum arrival time, is surprisingly easy. Denote by ϵ a negligibly small time value, i.e., think of ϵ as one millisecond. The modification of our algorithm consists of increasing τ_3 by ϵ after each profile evaluation, i.e., the modification consists of inserting a single addition compared to the base algorithm. If two journeys have different arrival times, then the earlier journey is chosen. If the arrival times are equal, the number of ϵs added determines which journey is chosen. As an ϵ is added each time that the travelers exits a train, the number of ϵs corresponds to the number of legs. The number of legs is thus optimized as secondary criterion.

In a real implementation, we multiply all departure and arrival times in the timetable with a small constant, such as for example 2^5. Timestamps, even with a , usually require significantly fewer than 32 bits. For example, to encode all seconds within a year, 25 bits are enough. We can therefore encode the modified timestamps using

32 14	13 9	8 1
rounded arr. time	number of legs	lower arr. time bits

Figure 7.4: Encoding used to represent timestamps. The numbers represent the bit-offsets within a 32 bit integer of the three data items.

32 bit integers. The value of ϵ is set to 1. The modifications to the algorithm depicted in Algorithm 7.3 adding a "+1" in line 7 and perform the scaling using two bit shift operations between the lines 4 and 5.

Stated differently, we encode the number of legs in the lower 5 bits of a timestamp. The higher 27 bits encode the arrival time. As an integer comparison only compares the lower bits if the higher bits are equal, we obtain the desired effect, that the journeys are tie-broken using the number of legs.

7.5.2 Rounding the Arrival Times

Optimizing the number of legs as secondary criterion, eliminates the most problematic earliest arrival journeys, such as those visiting a stop several times or those entering a trip multiple times. However, a journey that arrives at 8:02 with 10 legs is still preferred over a journey with 2 legs arriving at 8:03. While the former arrives earlier, most travelers prefer the later. This problem can be avoided by optimizing the number of legs in the Pareto-sense. Fortunately, a simpler partial solution to the problem exists that might be good enough for some applications.

The idea consists of rounding the value of τ_1 in the framework of Algorithm 7.1. If τ_1 is rounded down the lowest multiple of say 5 minutes, then both journeys are equal with respect to arrival time and therefore the journey with 2 legs is chosen. Rounding down to multiple of 5 minutes divides a day into 288 time buckets. Journeys arriving within one bucket are regarded as arriving at the same time and thus one with a minimum number of legs is picked. This avoids many problematic journeys, but it is only a partial solution as the problem remains at the time bucket borders. Further, the trick has no effect, if the difference in journey arrival times is larger than the bucket size.

We are only rounding the arrival times at the target stop. We do not round the departure or arrival times of intermediate connections. This trick therefore does not modify the transfer model.

A problem with this trick is that the profiles contain rounded arrival times. However, we want to display the non-rounded arrival times to the user. Further, there will only be one journey per bucket. Fortunately, these problems can be solved by permuting some bits in the timestamps.

Suppose that, we want to use 5 bits to encode the number of legs. Further, assume that we round the arrival times down to $2^8 = 256$. With seconds resolution that corresponds to rounding down to multiples of ≈4.2 minutes. The idea consists of not encoding the number of legs in the lowest bits of a timestamp. Instead, we use bits in the middle. The lowest 8 bits are the lower bits of the arrival time. The next higher 5 bits are the number of legs. The remaining bits encode the higher bits of the arrival time. Figure 7.4 illustrates the layout. The effect of this modification is that our algorithm now optimizes three criteria. These are:

1. The rounded arrival time,
2. the number of legs, and
3. the exact arrival time.

Criteria 2 and 3 are used as second and third criteria, i.e., they are tie-breakers. The exact arrival times can easily be reconstructed from this encoding. Further, assume that there are two journeys that arrive within the same bucket and have the same number of legs but have different arrival times. In the base version only one would be found. Using the refined algorithm both are found as they are not identical with respect to the third criterion.

Unfortunately, as already mentioned this trick mitigates but does not resolve the problem of trading many transfers for a tiny improvement in arrival time. However, for certain applications this trick reduces the number of problematic cases to a sufficiently small amount. The main advantage is that it is significantly easier to implement than the more complex solution described in the next paragraph. Further, the incurred overhead is comparatively low.

7.5.3 Pareto Optimization

The number of legs and the arrival times can be optimized in the Pareto-sense. For a fixed target t, we want to compute for every source stop s, every source time τ_s, and every number of legs ℓ, the earliest arrival time τ_t over all journeys from s to t not departing before τ_s with at most ℓ legs. To simplify this problem slightly, we bound ℓ by leg_{max} which is a constant in the algorithm. We usually set leg_{max} to 8 or a similarly large value, exploiting that travelers in practice do not care about journeys with too many legs.

We modify our algorithm by replacing all arrival times by constant-sized vectors. leg_{max} is the dimension of the vectors. We denote the elements of a vector A as $A[1], A[2] \ldots A[\text{leg}_{\text{max}}]$. The element $A[\ell]$ is the arrival time at the target, if the journey has at most ℓ legs. We define two operations that modify these vectors. The first is the *component wise minimum*, i.e., the result of the minimum operation of two vector A and B is a vector C such that $C[i] = \min\{A[i], B[i]\}$ for all indices i. The second operation is the *shift* operation, which is defined as follows: Shifting A yields a vector B such that $B[1] = \infty$ and $B[i] = A[i-1]$ for all other indices i.

```
1 for all stops x do $S[x] \leftarrow \{(\infty, (\infty, \infty \dots \infty))\}$;
2 for all trips x do $T[x] \leftarrow (\infty, \infty \dots \infty)$;
3 for connections c decreasing by $c_{\text{dep\_time}}$ do
4     if $c_{\text{arr\_stop}} = \text{target}$ then
5         $x \leftarrow c_{\text{arr\_time}} + \text{target}_{\text{change}}$;
6     else
7         $x \leftarrow \infty$;
8     $\tau_1 \leftarrow (x, x \dots x)$;
9     $\tau_2 \leftarrow T[c_{\text{trip}}]$;
10    $\tau_3 \leftarrow$ shift(evaluate $S[c_{\text{arr\_stop}}]$ at $c_{\text{arr\_time}}$);
11    $\tau_c \leftarrow \min(\tau_1, \tau_2, \tau_3)$;
12    $y \leftarrow$ arrival time of earliest pair of $S[c_{\text{dep\_stop}}]$;
13    if $y \neq \min(y, \tau_c)$ then
14        Add $(c_{\text{dep\_time}} - (c_{\text{dep\_stop}})_{\text{change}}, \min(y, \tau_c))$ at the front of $S[c_{\text{dep\_stop}}]$;
15    $T[c_{\text{trip}}] \leftarrow \tau_c$;
```

Algorithm 7.5: Pareto Connection Scan profile algorithm without interstop footpaths.

The interpretation of the minimum operation consists of taking the best of two options. Further, the shift operation can be interpreted as increasing the number of legs.

All τ-variables in the framework from Algorithm 7.1 become vectors. The trip data structure T becomes an array of vectors. The profile data structure S becomes an array of arrays of pairs of an integer and a vector. The walking distance to the target D remains an array of integers.

It is possible that a vector A dominates another vector B in one component, for example $A[1] < B[1]$, but B dominates A in another component, for example $A[2] > B[2]$. For this reason, the vector insertion must be modified. If all components of the new vector are dominated, then the profile is not modified. Otherwise, we insert the minimum of the new vector and the minimum of the earliest vector already in the profile. Two successive pairs can have the same arrival time with respect to certain but not all values of ℓ but different departure times.

In the base algorithm the profiles are initialized with a sentinel (∞, ∞) pair. The arrival time of this pair is a vector in the extended algorithm, i.e., the new sentinel is $(\infty, (\infty, \infty \dots \infty))$.

The computation of τ_1 starts analogous to the non-Pareto case. Our algorithm starts by computing the walking time x to the target. Afterwards x is converted to a vector A by setting $A[i] = x$ for all indices i. The operation of setting all components of a vector to one value is called *broadcast*.

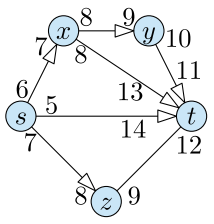


Figure 7.6: Example timetable. The circles are stops and the arrows are connections annotated by their departure and arrival times. All connections are part of different trips. There are four journeys from s to t with a varying number of legs: $s@5{\to}t@14$, $s@7{\to}z{\to}t@12$, $s@6{\to}x{\to}t@13$, and $s@6{\to}x{\to}y{\to}t@11$

In Algorithm 7.5, we present the profile Pareto algorithm in pseudo-code form. To simplify its exposition, we omit interstop footpaths. Fortunately, they can be incorporated in the same way as already described in Section 7.4 and depicted in Algorithm 7.3.

Example. Consider the example timetable depicted in Figure 7.6. We describe how the profile of s evolves during the execution of our algorithm. We set the target stop to t and leg_{max} to 3. The profile is a dynamic array of pairs of departure time and arrival time vectors. Initially it only contains an infinity sentinel, i.e., initially we have $S[s] = \{(\infty,(\infty,\infty,\infty))\}$.

The profile $S[s]$ is changed for the first time when the connection from s to z is scanned. The value of τ_c is $(\infty,12,12)$. As there is no way to reach t with at most 1 leg, the first component $\tau_c[1]$ is ∞. $\tau_c[2]$ is 12 as the target can be reached at 12 with 2 legs. Further, $\tau_c[3]$ is 12 also as the target can be reached at 12 with at most 3 legs. $\tau_c[3]$ is 12 even corresponding journey only contains 2 legs. τ_c is better in two components than the earliest vector in the profile, which is the (∞,∞,∞) sentinel. The algorithm therefore inserts a new pair, namely $(7,(\infty,12,12))$ into the profile $S[s]$. The profile $S[s]$ after the scan is $\{(7,(\infty,12,12)),(\infty,(\infty,\infty,\infty))\}$.

The profile $S[s]$ is changed for the second time, when the connection from s to x is scanned. The value of τ_c is $(\infty,13,11)$. $\tau_c[1]$ is ∞ as t cannot be reached without transfer. $\tau_c[2]$ is 13 because the journey $s@6{\to}x{\to}t@13$ contains two journeys. Further, $\tau_c[3]$ is 10 because the journey $s@6{\to}x{\to}y{\to}t@11$ with 3 legs exists. As the later has more than 2 legs, we have that $\tau_c[2] \neq 11$. τ_c is better in at least one component than the earliest vector in the profile, i.e., $(\infty,12,12)$. However, it is not better in every component. The algorithm therefore computes the minimum $\min\{(\infty,13,11),(\infty,12,12)\} = (\infty,12,11)$. The pair $(6,(\infty,12,11))$ is added to the profile $S[s]$. The resulting profile has the value $\{(6,(\infty,12,10)),(7,(\infty,12,12)),(\infty,(\infty,\infty,\infty))\}$.

The last time that the profile $S[s]$ might be changed is when the connection from s to

t is scanned. The value of τ_c is $(14, 14, 14)$. However, τ_c is not better in any component than the earliest vector in the profile, i.e., $(\infty, 12, 10)$. No pair is thus added.

After the execution of the algorithm the profile $S[s]$ is $\{(6, (\infty, 12, 10)), (7, (\infty, 12, 12)), (\infty, (\infty, \infty, \infty))\}$. To determine the arrival time for a source time τ_s and maximum number of legs ℓ, find the earliest pair with a departure time no earlier than τ_s. The ℓ-th component of the corresponding arrival time vector contains the answer.

For $\tau_s = 6.5$ and $\ell = 3$, we therefore first look up the first pair with a departure time after 6. This is $(7, (\infty, 12, 12))$. The ℓ-th, i.e., third, component is 12. The traveler can thus arrive at 12.

Earliest Arrival Time. In some cases, one is more interested in the minimum arrival time over all journeys than in the minimum arrival time over all journeys with at most leg_{max} legs. This can be implemented using a small change in the definition of the shift operation. The result of the modified shift of a vector A is a vector B such that $B[1] = \infty$, $B[\text{leg}_{\text{max}}] = \min\{A[\text{leg}_{\text{max}} - 1], A[\text{leg}_{\text{max}}]\}$, and $B[i] = A[i-1]$ for all other indices i. With this modification, the leg_{max}-th vector component contains the earliest arrival times over all journeys.

SIMD. All vectors operations, i.e., component-wise minimum, component shifting, and broadcasting a value to all components, can be implemented using SIMD operations on all common processor architectures. This includes x86 processors with the SSE and AVX2 instruction sets. One SSE vector has four components with 32 bit integers. Concatenating two vectors, yields an efficient implementation for $\text{leg}_{\text{max}} = 8$. Alternatively, AVX2 vectors have eight components with 32 bit integers. One AVX2 vector is therefore large enough.

7.6 Journey Extraction

In the previous section, we introduced an algorithm to compute profiles. In this section, we describe how to extract corresponding journeys in a post processing step.

Similar to the extraction process for the earliest arrival time Connection Scan algorithm, the extraction comes in two variants. The first conceptually simpler approach consists of storing journey pointers. The second approach computes the journey pointers on the fly during the extraction.

The input consists of a source stop s and source time τ_s, the output of an earliest arrival journey towards the target stop for which the profile was computed. If transfers are optimized in the Pareto-sense, then the input contains additionally a maximum number of legs ℓ.

Several journeys can exist that are identical with respect to all considered criteria, i.e., they depart at the same source stop at the same source time and arrive at the same target stop at the same target time and have the same number of transfers. We only consider the problem setting of extracting one of these journeys. Our algorithms guarantee that the extracted journey visits no stop or trip twice even when the number of legs is not optimized.

7.6.1 Journey Pointers

In the base profile algorithm, the pairs (d, a) contain two pieces of information namely a departure time d and an arrival time a. We extend the pairs with two connection IDs l^{enter}, l^{exit}, turning the pairs into quadruples $(d, a, l^{\text{enter}}, l^{\text{exit}})$. The meaning of such a quadruple is that there is an optimal journey j that arrives at the target stop at time a and departs at time d. The extracted journey j starts with a footpath towards $l^{\text{enter}}_{\text{dep\_stop}}$. j leaves the stop using the connection l^{enter}. The traveler exits the train at the end of the connection l^{exit}. These quadruples can be used to iteratively extract an optimal journey.

The extraction starts by computing the time needed to directly transfer to the target. Doing this is trivial without interstop footpaths. With footpaths, we use the D array of the base profile algorithm. In the next step, our algorithm determines the first quadruple p after τ_s in the profile $P[s]$ of the source stop s. If directly transferring to the target is faster, then the journey consists of a single footpath and there is nothing left to do. Otherwise, p contains the first leg of an optimal journey. The algorithm then sets s to $l^{\text{exit}}_{\text{arr\_stop}}$ and τ_s to $l^{\text{exit}}_{\text{arr\_time}}$ and iteratively continues to find the remaining legs of the output journey.

It remains to describe how l^{enter} and l^{exit} are determined when inserting the quadruple into the profile during the scan. l^{enter} is the connection being scanned and is therefore already known. To determine l^{exit} efficiently, we extend the trip information T with a connection ID for each trip, i.e., T becomes an array of pairs of arrival times and connection IDs. Each time that the arrival time stored in T is decreased, the algorithm sets the trip's connection ID to the currently scanned connection. When inserting the quadruple, c^{exit} is the connection ID stored with currently scanned connection's trip.

This approach can be combined with Pareto-optimization by replacing l^{enter}, l^{exit}, and the trip connection IDs with constant-sized vectors. The input of the algorithm must be extended with the maximum number of desired legs.

7.6.2 Without Journey Pointers

Similarly to the earliest arrival Connection Scan, it is possible to implement a journey extraction without modifying the scan.

Our algorithms require enumerating the outgoing connections of a stop ordered by departure time. To efficiently support this operation, we create an auxiliary data structure that consists of an adjacency array that maps a stop s onto the departure time and the ID of all connections c departing at s, i.e., onto the connections c for which $c_{\text{dep\_stop}} = s$ holds. The outgoing connections are ordered by departure time. Further, our algorithm needs to be able to enumerate all connections in a trip after a given connection. To efficiently support the second operation, we create another auxiliary adjacency array that maps a trip t onto the IDs of the connections c in the trip, i.e., onto the connections c for which $c_{\text{trip}} = t$ holds. The connections are ordered by their position in the trip. To enumerate the connections in a trip after a given connection c, we enumerate the connections in c_{trip} from late to early and abort the enumeration once c is encountered. All auxiliary data structures are independent of the target stop. Further, both data structures can be computed by essentially sorting the connections by various criteria. We can therefore compute the auxiliary data in a fast preprocessing step.

Similarly, to the journey pointer approach, our second approach starts by checking, whether directly walking from the source stop s and the source time τ_s to the target t is optimal. It terminates, if this is the case. Otherwise, our algorithm must compute a pair of valid l^{enter} and l^{exit}. In the first approach, these were stored in the pairs which is no longer the case in the second approach. Our algorithm therefore needs to infer the values. It does so by searching for the earliest pair (d, a) after τ_s in s's profile $P[s]$ using a binary search. We know that there must be a footpath f outgoing from s towards $l^{\text{enter}}_{\text{dep\_stop}}$ such that $l^{\text{enter}}_{\text{dep\_time}} = d + f_{\text{dur}}$. By iterating over the outgoing footpaths of s and checking this condition, we obtain a set $\{c^1, c^2 \dots c^k\}$ of candidates for l^{enter}. We know that there must be an optimal first leg l, such that l^{enter} is among the candidates.

We can optionally prune the candidate set using the trip arrival times $T[x]$ computed during the profile scan. $T[x]$ is the minimum arrival time over all optimal journeys departing in a connection of trip x. We therefore know that if for a candidate $T[c^i_{\text{trip}}] > a$ holds, that c^i cannot be l^{enter} and we can therefore remove c^i from the set.

For the remaining candidates, we need to look at the connections in their trips. For each potential candidate c^i, our algorithm enumerates all connections c in its trip that come after c^i, including c^i itself. For each c, our algorithm searches for the earliest pair (d', a') in $c_{\text{arr\_stop}}$'s profile after $c_{\text{arr\_time}}$ using a binary search. If $a = a'$, then we found an optimal first leg and c is the corresponding l^{exit}. If we only wish to extract one journey, then our algorithm can discard the remaining candidates. Our algorithm iterates by setting s to $l^{\text{exit}}_{\text{arr\_stop}}$ and τ_s to $l^{\text{exit}}_{\text{arr\_time}}$. To assure that no trip is used twice in a journey, we pick the latest valid l^{exit} in the trip. As we enumerate connections from late to early, the first valid l^{exit} we encounter is automatically the latest.

7.6.3 Pareto Optimization

The candidate set is computed by finding the first pair (d, a) departing after τ_s. This is correct for the base profile scan algorithm. However, the Pareto-extension can insert several pairs with the same departure time with respect to ℓ. A modification to the extraction is therefore necessary.

Consider for example the example illustrated in Figure 7.6. Suppose that the traveler departs at s at 5 and wants to use at most 2 legs. Already the first pair $(6, (\infty, 12, 10))$ in the profile departs later than 5. However, there is no earliest arrival journey towards t departing at 6 towards t with at most 2 legs. The corresponding journey departs at 7. Indeed, the second pair $(7, (\infty, 12, 12))$ in the profile has the correct departure time and arrives at the same time.

To fix this problem, we slightly modify the algorithm. First we find the earliest pair p departing no earlier than τ_s. In a second step, we iterated over the pairs in the profile from early to late starting at p until we find the last pair q with the same time than p for the requested number of legs. The departure time of q is used to determine the candidate set.

7.7 Experiments

We use the experimental setup described in Section 6.4.1. In Table 7.7, we report the running times of the earliest arrival Connection Scan profile algorithm. We report the running times for both main instances on both of our test machines. We iteratively activate optimizations to show their impact. Activating range queries also includes not processing unreachable trips. We also report the running time needed to perform the scan and extract for every pair in the source stop's profile a corresponding earliest arrival journey.

The comparison between the two machines is interesting. We expect the newer machine to be faster, as it has a faster processor, a newer architecture, and faster RAM. This expected behavior is nearly always the observed behavior, except on the Germany instance for non-range queries. The differences in L3 cache sizes explains the effect. The newer machine is better with respect to every criterion except L3 cache. The old machine has 20 MiB while the newer one only has 10 MiB. The London instance is smaller and therefore a larger part of the stop profiles fit into the 10 MiB. If we compute range queries, only parts of the stop profiles are computed. This part is smaller and therefore a greater percentage fits into the L3 cache. The newer machine is therefore faster on range queries and slower on non-range queries. The conclusion is that a sufficiently large cache is necessary for a good Connection Scan profile performance.

Activating prefetching decreases the running times. On the newer machine and the London instance the speedup is only about 1.02. However, on the Germany instance the gain is already 1.05. This observation again illustrates that caching effects matter for good performance. On the London instance large parts of the frequently used data structures are never evicted from L3 cache. The gain from prefetching comes therefore mostly from moving data to the lower cache levels. On the Germany

Instance	Pre-fetch	Limited Walk.	Source Dom.	Range Query	Journeys Extraction	Running Time [ms]
Older Machine with 20 MiB of L3 cache						
Germany	○	○	○	○	○	2 132.1
Germany	●	○	○	○	○	1 995.7
Germany	●	●	○	○	○	1 567.2
Germany	●	●	●	○	○	1 119.3
Germany	●	●	●	●	○	253.1
Germany	●	●	●	○	●	1 118.4
Germany	●	●	●	●	●	253.1
London	○	○	○	○	○	287.8
London	●	○	○	○	○	279.7
London	●	●	○	○	○	162.3
London	●	●	●	○	○	119.9
London	●	●	●	●	○	11.1
London	●	●	●	○	●	121.2
London	●	●	●	●	●	11.2
Newer Machine with 10 MiB of L3 cache, used in most experiments						
Germany	○	○	○	○	○	2 517.2
Germany	●	○	○	○	○	2 391.0
Germany	●	●	○	○	○	1 684.4
Germany	●	●	●	○	○	1 246.2
Germany	●	●	●	●	○	217.9
Germany	●	●	●	○	●	1 246.4
Germany	●	●	●	●	●	218.0
London	○	○	○	○	○	242.3
London	●	○	○	○	○	238.7
London	●	●	○	○	○	140.0
London	●	●	●	○	○	106.9
London	●	●	●	●	○	9.4
London	●	●	●	○	●	107.9
London	●	●	●	●	●	9.4

Table 7.7: Earliest arrival profile computation running times.

instance prefeching moves data from the RAM into L3 cache more often. As the absolute differences in access speeds between L3 cache and RAM are greater than between L2 and L3 cache, the speedup is lower for the London instance.

Activating the limited walking optimization further reduces the running times. The speedup is about 1.4 to 1.7, which is roughly comparable to the speedups achieved for the non-profile algorithm variants.

Activating source domination further reduces the running times. As source domination prunes pairs from profiles except the source stop, the algorithm solves a more restricted problem setting. Instead of computing the profiles from every stop towards the target stop, it now only computes a single profile from the source stop to the target.

Switching to range queries drastically reduces the running times. The specified maximum travel time of twice the minimum travel time allows the algorithm to limit the connections that need to be scanned. On the Germany instance the speedup is about a factor 6. On the London instance the speedup of 11 is higher. These speedups are roughly comparable to the speedups achieved by activating the start and stop criteria in the non-profile earliest arrival algorithm. The reason is that the decrease in scanned connections is roughly comparable. Further, as already observed, a traveler needs less time to traverse London than to traverse Germany. The relative decrease in scanned connections is thus higher on the London instance and as a consequence the achieved speedups are higher.

In Table 7.8, we report running times of the Connection Scan Pareto profile algorithm. It optimizes the number of legs, the arrival time, and the departure time in the Pareto-sense. The maximum number of legs is set to 8. We use the algorithm variant that computes the earliest arrival time in the 8-th vector component. We iteratively activate our proposed optimizations to demonstrate their effectiveness.

We present three SIMD variants. All three use the same memory layout. All use vectors with 256 bits that contain 8 components with a 32-bit timestamp. They differ in what processor instructions are used to operate on the vectors. The first variant uses no special instructions and works with loops with a fixed number of iterations. The second variant uses SSE instructions. SSE registers are 128 bits wide. To process one vector, two SSE instructions are thus required. The third variant uses AVX registers. Luckily, these are 256 bits wide and therefore a single instruction is sufficient. We use integer AVX arithmetic instructions. These were introduced with AVX2, a feature introduced in the Haswell processor architecture. Our AVX code can therefore not run on our older test machine, which does not yet support AVX2.

The first optimization that we consider consists of prefetching memory. On the Germany instance without SSE nor AVX, a speedup of 1.16 was achieved. This is significant, considering that no algorithmic changes were performed. Interestingly, the speedup is only 1.02, when comparing the AVX prefetch and AVX non-prefetch running times. It is also interesting that by using AVX, compared to the base version,

Instance	SIMD	Pre-fetch	Limited Walk.	Source Dom.	Range Query	Running Time [ms]
Germany	—	○	○	○	○	8 298.5
Germany	—	●	○	○	○	7 109.3
Germany	SSE	○	○	○	○	4 792.2
Germany	SSE	●	○	○	○	4 612.6
Germany	SSE	●	●	○	○	3 519.9
Germany	SSE	●	●	●	○	2 834.6
Germany	SSE	●	●	●	●	279.5
Germany	AVX	○	○	○	○	4 402.9
Germany	AVX	●	○	○	○	4 332.7
Germany	AVX	●	●	○	○	3 220.7
Germany	AVX	●	●	●	○	2 489.6
Germany	AVX	●	●	●	●	259.2
London	—	○	○	○	○	777.5
London	—	●	○	○	○	749.1
London	SSE	○	○	○	○	424.1
London	SSE	●	○	○	○	420.1
London	SSE	●	●	○	○	261.2
London	SSE	●	●	●	○	213.8
London	SSE	●	●	●	●	11.9
London	AVX	○	○	○	○	355.6
London	AVX	●	○	○	○	359.8
London	AVX	●	●	○	○	206.1
London	AVX	●	●	●	○	170.2
London	AVX	●	●	●	●	10.7

Table 7.8: Profile computation running times with optimization of the number of legs and the earliest arrival time in the Pareto-sense.

a speedup of 1.9 is achievable. Especially the later is interesting, as we expect SIMD to have the largest benefit in compute-bound algorithms and our previous experiments suggest that the Connection Scan algorithm heavily depends on memory access speeds. One explanation for these two effects is that the AVX code has fewer instructions, making it easier for processor to predict memory access patterns. This would explain why the benefit of prefetching nearly vanishes but running times drastically decrease. This explanation is also consistent with the observation that using AVX is a benefit over SSE as the AVX code requires fewer instructions.

The speedups of the limited walking and source domination optimizations are comparable to those observed for the earliest arrival profile algorithm. We refer to the discussion of these experiments for an interpretation of the observed effects. The speedup of the range query variant is about 10 on the Germany instance and 17-19 on the London instance. These speedups are larger than those observed for the earliest arrival profile algorithm. The difference is likely due to the Pareto algorithms having a larger overall memory consumption. As a consequence, caching effects have a larger impact and therefore a of the memory footprint yields a large relative advantage.

7.7.1 Comparison with Related Work

In Table 7.9, we compare the Connection Scan profile algorithm with two competitor algorithms. The first is the Self-Pruning Connection-Setting (SPCS) algorithm [49]. It computes profiles that optimize departure and arrival time in the Pareto-sense but does not optimize transfers. The algorithm can be combined with the colored timetable optimization, which was used in our experiments. We therefore refer to the algorithm as SPCS-col in Table 7.9. The second competitor is rRAPTOR [52]. Similar to the base RAPTOR algorithm, it inherently optimizes transfers in the Pareto-sense. The Connection Scan algorithm (CSA) was run with AVX and limited walking activated.

Both, rRATPOR and CSA, clearly dominate SPCS-col in terms of running time. The difference between CSA and rRATPOR is smaller. CSA is always faster, but on the Germany instance, the gap is only up to a factor of 2. On the London instance, there is a speedup of up to 4.7.

7.8 Chapter Conclusion

By using CSA and exploiting the full capabilities of modern processors, it is possible to answer Pareto range queries on the large Germany instance in a quarter of a second. It is feasible to construct interactive timetable information systems upon these running times. However, ideally lower running times are desirable. For example, spending a quarter of a second per query in a web server severely limits its throughput. Fortunately, we were able to achieve these running times without

Instance	Algorithm	Pareto	One-to-one	Running Time [s]
Germany	CSA	○	○	1.68
Germany	CSA	○	●	1.25
Germany	CSA	●	○	3.22
Germany	CSA	●	●	2.49
Germany	SPCS-col	○	○	10.95
Germany	SPCS-col	○	●	8.40
Germany	rRAPTOR	●	○	6.27
Germany	rRAPTOR	●	●	4.73
London	CSA	○	○	0.14
London	CSA	○	●	0.11
London	CSA	●	○	0.21
London	CSA	●	●	0.17
London	SPCS-col	○	○	1.19
London	SPCS-col	○	●	0.79
London	rRAPTOR	●	○	0.97
London	rRAPTOR	●	●	0.68

Table 7.9: Comparison of profile algorithms.

compromising the excellent data structure construction times of the base algorithm. Flexible realtime updates are possible.

8 Connection Scan Accelerated

In the previous sections, we presented the Connection Scan family of algorithms. We demonstrated that queries can be answered very quickly on modern hardware. Even Pareto range queries can be answered in well below a second even on the large Germany instance. A significant advantage of the Connection Scan algorithms is the lightweight preprocessing. It mainly consists of sorting the connections, which can be done in very few seconds. This allows us to quickly update the timetable to account for disturbances, such as delayed trains, blocked stops or tracks, or overbooked trains.

While all of these properties make the Connection Scan family of algorithms a good fit for many applications, it is also interesting to investigate whether further gains are achievable by using more heavy-weight preprocessing techniques. Further, even though the achieved running times on the Germany instance of the base algorithms are low enough for interactive applications, we expect them to consume a significant amount of resources. Lower running times are therefore very desirable in practice. Investigating the combination of Connection Scan with more heavy-weight preprocessing techniques is therefore the topic of this chapter.

We investigate a multilevel overlay extension to the Connection Scan algorithms, which we call Connection Scan Accelerated (CSAccel). The central ideas are similar to those used in [121, 88, 45]. In several studies, this approach has proven to enable very fast queries in road networks. Compared to Dijkstra's algorithms, speedups on the order of 1000 are possible. It is therefore reasonable to expect similar speedups on timetable networks. We are not the first to investigate this question. Unfortunately, previous research [19, 21] has shown that achieving similar speedups is harder than one would naively expect. Our work is no exception to this observation. Our multilevel extension manages to provide a significant speedup on the Germany instance. However, the speedup lacks far behind of what is achievable in road networks.

The core idea of our extension is best illustrated using an example: When planning a journey from Karlsruhe to Stuttgart, do not scan rural bus connections around Hamburg. We use overlays to formalize the concept of rural bus. Our algorithm partitions the stop set into cells. Karlsruhe and Stuttgart are put into the same cell. Hamburg is in a different cell. For every cell, our algorithms computes a subset of *transit connections*. For every pair of connections entering and leaving a cell z, there must be a journey with a minimum number of transfers, that only enters or exits trips at transit connections of z. For rural buses, usually no such journey exists and thus they are not in the transit connection set. When traveling from Karlsruhe to

Stuttgart, our algorithm only looks at the transit connections of Hamburg's cell and thus skips the rural buses around Hamburg.

Following the setup and terminology of [45], our algorithm works in three phases. In the first phase, called *preprocessing phase*, a multilevel partition of the stop set is computed. In the second phase, called *customization phase*, our algorithm computes overlays for every cell. Finally, in the third phase, called *query phase*, our algorithm computes arrival times and journeys. The second phase uses the results of the first phase. Similarly, the third phase uses the results of the first and the second phases. The preprocessing phase should only use data that rarely changes, such as for example what tracks exist and perhaps what tracks are highly frequented. The idea is that the preprocessing phase does not have to be rerun very often and may therefore be slow. To update the timetable, it should be sufficient to rerun the customization, which should be fast. Our customization phase works with every stop partitioning, as long footpaths do not cross cell boundaries and the stop sets are identical. However, if the timetables used during preprocessing and customization differ too much, then customization and query performance will significantly degrade.

Multilevel approaches inherently rely on the structure of the network. Small, balanced graph cuts are a necessity. Without these, the achievable speedups crumble. Fortunately, as shown in many studies, such as the one in Chapter 3, road graphs typically have this structure. However, for timetables, the situation is less clear. Indeed, country-wide timetables that consist of many urban centers differ in structure from timetables that consist of a single large urban region. There typically exist small, balanced cuts between cities, however, cutting through a city is significantly more difficult. Many cities contain natural cuts such as rivers or large main roads. This property is exploited to achieve fast shortest path queries in road networks. Unfortunately, in timetable networks, rivers are not necessarily advantageous. Often, several trains or buses lines pass over a single bridge. Cutting through tracks with a high public transit frequency is expensive, in the context of timetables, as we need to weight the cuts by the number vehicles that pass over it. We therefore expect the performance of all multilevel overlay extensions to perform poorer on pure urban instances. This differs from the basic Connection Scan algorithm, whose performance is nearly independent of the timetable structure.

Connection Scan algorithms find a journey j with legs $l^1, l^2 \dots l^k$, if the connections l^1_{exit}, $l^1_{\text{enter}} \dots l^k_{\text{exit}}$, l^k_{enter} are scanned in the correct order. These are the connections where the traveler transfers, i.e., enters or exits. A connection where the traveler does neither does not have to be scanned. Scanning all connections ordered by departure time fulfills this property for all journeys. This is the core observation exploited by the Connection Scan base algorithms. For a fixed source and target stop it can be sufficient to only scan a subset of the connections. Our algorithm exploits this observation. Our query phase thus works in two subphases. In the first subphase, a sorted connection

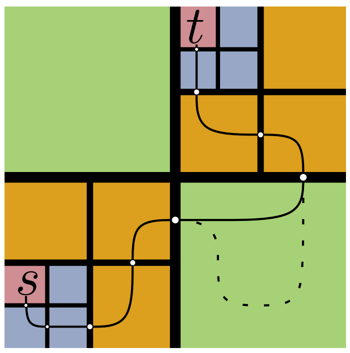


Figure 8.1: Multi level journey example from stop s to stop t.

subset $C_\mathcal{S}$ is assembled. For every pair in the st-profile, there must be a journey j, such that all transfer connections of j are included in $C_\mathcal{S}$. In the second subphase, the Connection Scan base algorithms are run restricted to the connections in $C_\mathcal{S}$.

Our algorithm computes $C_\mathcal{S}$ by merging arrays of sorted connections. In the base setting, every cell has an associated sorted array of transit connections. To compute $C_\mathcal{S}$, one would identify all potentially relevant cells and merge their transit connections. Unfortunately, the number of these cells can be large and merging sorted arrays is a task that requires some running time. We therefore want to reduce the number of arrays merged. We introduce the concept of *long distance connections*. A transit connection of a cell z is a long distance connection of its direct parent cell. On the lowest level, all connections within a cell z are long distance connections of z. For every cell, our algorithm stores a sorted array of long distance connections. To assemble $C_\mathcal{S}$, our algorithm merges the long distance connections of all cells that contain the source or target stop or both.

If the long distance connections of a cell z are merged into $C_\mathcal{S}$, then also the connections of z's parent are merged. We can exploit this observation to further thin out the long distance connection set. If c is a long distance connection of a cell z and of z's parent cell, then it is sufficient to store c in the parent cell's array. Further, we can construct the transit connections with the property that if c is a long distance connection of z's parent, then c is a long distance connection of z. A consequence of this is that every connection is contained in at most one thinned out long distance connection set. The memory consumption is therefore linear in the number of connections.

To prove that our algorithm is correct, we show that for every Pareto-optimal journey j, there exists a Pareto-optimal journey j' that only enters or exists trips in the merged connection subset $C_\mathcal{S}$, such that j and j' have the same departure and arrival time and have the same number of legs. Before formally proving the correctness, we illustrate the employed arguments using an example.

Figure 8.1 illustrates a stop set that was recursively partitioned along the straight solid lines. At every level, every cell was partitioned into four parts. The thickness of the lines indicates the level – the thicker the line the higher a level. The solid bent line represents the journey j. The colored areas represent transit connections merged into $C_\mathcal{S}$. Red means lowest level, blue is next higher, then orange and green is the highest level. The white dots represent connections, where j crosses cell boundaries. The dotted line represents an alternative subjourney of j within the green bottom-right cell.

The journey j consists of a prefix from s to the first boundary connection, several subjourneys that traverse cells, and a suffix from the last boundary connection to t. The subjourneys are enclosed by the white dots in Figure 8.1. The constructed journey j' has the same prefix and suffix and crosses the cell boundaries in the same connections as j, i.e., in the white dots. Because the prefixes and the suffixes are equal, the departure and arrival times of j and j' are equal. The subjourneys within a cell can differ. For example, it is possible that j uses the solid line, whereas j' uses the dotted line. By construction, we know that for every cell, entry connection, and exit connection, there exists a subjourney with a minimum number of transfers that only enters or exits trips at transit connections in $C_\mathcal{S}$. For every cell z that j traverses, replace the subjourney of j within z with the corresponding minimum transfer journey to obtain j'. j' cannot have more transfers than j because otherwise one of the employed subjourneys would not have had a minimum number of transfers. Further, as j was Pareto-optimal, j' cannot have fewer transfers than j. j and j' therefore have the same number of transfers.

In the following, we describe the details of our multi level extension. The text is organized along the three main phases. We first describe the preprocessing phase which mostly consists of a graph partitioning problem. Afterwards, we describe the customization phase, which primarily consists of computing the transit connections. Next, we explain how to perform the queries, which consists of computing $C_\mathcal{S}$. Finally, we present an experimental evaluation of the algorithm and a comparison with related work.

8.1 Phase 1: Partitioning the Stop Set

A k-partition of the stop set V divides V into k cells such that every stop is in exactly one cell. We require that stops connected by footpaths must be in the same cell, i.e., footpaths must not cross cell borders. A connection is interior (exterior) to a cell if it departs at a stop inside (outside) the cell. In an l-level partition with k children, the stop set is recursively split into k cells over l levels. At the bottom level there are k^l cells. The top level consists of a single cell that contains all stops. The parent p of a cell z is the cell which was split to create z. Similarly, z is a child of p. The bottom level cells do not have children and the top level cell does not have a parent.

The preprocessing step consists of computing an l-level partition with k children where l and k are tuning parameters of the algorithm. We perform the partitioning using a graph partitioner. From the timetable, we build an undirected, weighted graph as follows: The stops form the node set of the graph. There is an edge between two nodes if there is a connection or footpath between the corresponding stops. If there is a footpath, we weight the corresponding edge with ∞, to assure that it is not cut. Otherwise, the weight of an edge reflects the number of connections between the edge's endpoints. We partition the graph into k parts using KaHip v1.0c[1] with 20% imbalance. We recursively repeat this operation l times. We run KaHip using the "strong"-preconfiguration. Unfortunately, the results we get from KaHip vary significantly depending on the random seed given to it. We therefore run KaHip at each level in a loop with varying seeds until for 10 iterations no smaller cut is found. This setup is definitely not the fastest partitioning method. Fortunately, it is fast enough and the obtained cuts are reliably small.

8.2 Phase 2: Computing Transit Connections

In this section, we describe how to compute the transit and long distance connections. We start by describing how to compute journeys with a minimum number of transfers. In the next step, we describe how to use this algorithm to compute transit and long distance connections sequentially. Finally, we describe how the customization algorithm can efficiently be parallelized.

8.2.1 Minimum Number of Transfers

To compute overlays, our algorithm needs to quickly compute journeys with a minimum number of transfers between each pair of connections entering and leaving a cell z. We implement this using a variant of the earliest arrival Connection Scan profile algorithm with secondary transfer optimization. We run this algorithm on a part of the network restricted to the connection inside of z. Contrary to the algorithms described in Section 6.3, the traveler does not start and end at a stop but starts in an entry connection c_s and ends in an exit connection c_t. A key observation is that, for a fixed target exit connection c_t, the arrival times of all journeys are the same. The algorithm therefore only optimizes the number of transfers.

Our algorithm iterates in an outer loop over the exit connections c_t and computes a backward profile for each. In an inner loop, it iterates over all entry connections and evaluates the profile. It extracts a corresponding journey j from c_s to c_t. All connections where j exits or enters a trip are marked as transit connections including c_s and c_t.

In the following, we describe the inner loop of our algorithm in greater detail. In the inner loop, we have a fixed target exit connection c_t and a set of source enter

[1]We also tried Metis in a preliminary experiment and the resulting query and customization running times were dominated.

```
for all stops $x$ do $S[x] \leftarrow \{(\infty,\infty)\}$;
for all trips $x$ do $T[x] \leftarrow \infty$;
for all footpaths $f$ with $f_{\text{arr\_stop}} = (c_t)_{\text{dep\_stop}}$ do
    Incorporate $((c_t)_{\text{dep\_time}} - f_{\text{dur}}, 0)$ into profile of $S[f_{\text{dep\_stop}}]$;
$T[c_t] \leftarrow 0$;
for all connections $c$ strictly before $c_t$ decreasing by $c_{\text{dep\_time}}$ do
    $\tau_2 \leftarrow T[c_{\text{trip}}]$;
    $\tau_3 \leftarrow$ (evaluate $S[c_{\text{arr\_stop}}]$ at $c_{\text{arr\_time}}$) + 1;
    $\tau_c \leftarrow \min\{\tau_2, \tau_3\}$;
    if $(c_{\text{dep\_time}}, \tau_c)$ is non-dominated in profile of $S[c_{\text{arr\_stop}}]$ then
        for all footpaths $f$ with $f_{\text{arr\_stop}} = c_{\text{dep\_stop}}$ do
            Incorporate $(c_{\text{dep\_time}} - f_{\text{dur}}, \tau_c)$ into profile of $S[f_{\text{dep\_stop}}]$;
    $T[c_{\text{trip}}] \leftarrow \tau_c$;
    if $c \in C_s$ then
        Extract journey from $c$ to $c_t$ and mark transit connections;
```

Algorithm 8.2: Minimum transfer profile algorithm between connections.

connections C_s. For every connection c_s in C_s, a minimum transfer journey from c_s to c_t should be computed. The pseudo-code of the algorithm is given in Algorithm 8.2. We start our scan with the connection c_t, as all connections after it are obviously not reachable. The body of the loop is left mostly unchanged compared to the base algorithm. There is only one major modification: We no longer compute a walk-to-target time τ_1. It would be ∞ for every connection except c_t, which is not useful. Instead, we introduce a special case for c_t outside the loop. As all journey end in c_t, it does not matter what arrival time we give c_t. For simplicity, we use 0.

To quickly extract journeys, we use journey pointers. The extraction works analogous to the base algorithm with one modification. To extract the first leg of a journey starting in a connection c_s, we need to look at the exit connection stored with c_s's trip. However, this exit connection may be overwritten, if there are several entry connection inside of this trip. We therefore extract the journey directly after processing c_s. A trip can contain multiple entry connections, if it leaves and enters a cell multiple times.

8.2.2 Computing Transit and Long-Distance Connections

We compute transit connections bottom-up, i.e., the transit connections of the lowest level are computed first. To accelerate the computations on the higher levels, we use transit connections of lower levels. A central observation is that for every cell z there is a valid transit connection set T_z that is a subset of the long-distance connection set L_z of z. Our algorithm thus works as following: For all levels l from bottom to the top and all cells z in the level l, first compute the long-distance connections L_z of z, then compute the transit connections T_z of z by restricting the search to the long-distance connections L_z. For the lowest level cells, the long-distance connection set contains all interior connections. In a second, faster step, we iterate a second time over the levels and cells and thin out the long distance connection sets.

8.2.3 Parallelization

Significant speedups can be achieved by parallelizing the transit connection computation. There are two levels of granularity on which we can parallelize: (1) we can compute the transit connections of cells on the same level in parallel, and (2) we can compute the journeys for different exit connections within a cell in parallel. The former has the advantage that the data structures of different cells are completely disjoint, minimizing the necessary communication and synchronization. However, the boundary sizes of cells are very skewed because of urban centers. It is therefore difficult to keep all threads fully occupied. The later is more fine-grained and therefore allows us to fully occupy all threads. However, more communication and synchronization is needed.

We use a hybrid approach that combines the best properties of both. In a first step, we sort all cells first by level from bottom to top and as a secondary criterion by decreasing boundary size. The obtained list is a topological sorting of the dependencies between the cells. We sort the cells by boundary size, to assure that the more expensive cells, i.e., those with a larger boundary, are processed first. We attach to every cell an atomic counter, that indicates the number of children cells that have not yet been computed. If this counter reaches zero, the processing can start. The bottom level cells start with a counter of zero. The higher level cells start with the number of children used in the partitioning. We spawn as many threads as the hardware can process simultaneously. Every thread iterates over the list of cells once. If it finds a cell with counter zero, it grabs the cells by atomically the counter to prevent other threads from seeing the zero counter value. The thread then processes the cell and once it is finished decreases the counter of its parent. When a thread reaches the end of the list, it puts itself into a pool of idle threads. The threads that are still processing cells look at whether this pool is non-empty between processing two target exit connections. If it is non-empty, they extract an idle thread atomically and the thread helps processing the cell. At the end of processing a cell, all threads but the main one are put back into the idle pool.

8.3 Phase 3: Answering Queries

In this section, we describe how to compute the connection subset $C_\mathcal{S}$ and how the query algorithms need to be modified.

2-way vs *k*-way. Efficiently computing $C_\mathcal{S}$ is a crucial component of an efficient implementation of our query algorithms. The input consists of several arrays of sorted data that should be merged. Three major strategies exist [94]. The first consists of iteratively performing a two-way merge to combine pairs of arrays. The other two are direct k-way merges. The idea consists of storing a pointer into each array and iteratively determining the smallest element and increasing the corresponding pointer. Determining which element is the smallest is the challenging part. There are two approaches. One can use a binary heap or one can use tournament trees. All three variants have a worst case running time of $O(n \log k)$, where n is the total number of elements. We implemented all three variants and in preliminary experiments on our data set, the iterative two-way merge was the fastest, followed by the binary heap, and the tournament heaps came last. Unfortunately, the iterative two-way merge can only compute $C_\mathcal{S}$ as a whole. The direct k-way approach allows us to perform a partial merge, i.e., only merge the first x connections, which is enough for some of our applications.

Profile Queries. We implement the earliest arrival and Pareto profile algorithms in the straight forward way. First, our algorithm computes $C_\mathcal{S}$ using an iterative two-way merge. In a second step, the Connection Scan base algorithm is applied restricted to $C_\mathcal{S}$.

Earliest Arrival Queries. Earliest arrival queries have a start and stop criterion. We therefore use a direct k-way merge to avoid computing parts of $C_\mathcal{S}$ that will not be scanned. For each of the k arrays, we run a binary search to determine the first connection not before the source time. We then start the k-way merge. We run the merging process until the stop criterion aborts the scan.

Range Queries. For range queries, we use a similar approach. We first perform the k binary and then start with the k-way merge. To determine the reachable trips, we execute a non-profile earliest arrival scan. Once the stop-criterion activates, we continue the merge until all connections departing within the desired range have been computed. We store the output of the merging process into a temporary array. We then run the profile algorithm restricted to the connections in this temporary array.

8.4 Experiments

In this section, we experimentally evaluate CSAccel. We use the experimental setup described in Section 6.4.1. We start by comparing various multilevel configuration in terms of preprocessing, earliest arrival query, and profile query running time. For one of the best configurations, we present an evaluation of range queries. We conclude with a comparison of experimental results with related work.

8.4.1 Query Experiments

In Table 8.3, we experimentally evaluate Connection Scan Accelerated for various configurations. A label X-c-l refers to a recursive partitioning of timetable X, over l levels, with c children per level. The number of lowest level cells is c^l. We report c^l in the table to give an overview over the granularity of the partition. We report the time needed to compute the multilevel partitioning with KaHip version 1.0c. In preliminary experiments, we also tried using Metis. The partition running times were significantly lower but the customization and query running times were higher. As we focus on the later two values, we therefore refrain from reporting these experiments. Further, we report the customization running times. Both, the preprocessing and customization experiments, were performed on our older Xeon E5-2670 machine with 16 physical hardware threads. The customization running times are parallelized, whereas the preprocessing running times are sequential. We also report running times for various query variants. The query experiments were run sequentially on the newer Xeon E5-1630v3 machine. We report the average running times for the earliest arrival time, the earliest arrival profile, and the Pareto profile problem settings. Journey extraction was not performed. Range query experiments are reported in Table 8.4 and discussed later in this section. We activated all optimizations of the base algorithm, i.e., start and stop criteria, source domination, limited walking, and AVX. Beside the query running times, we also report the number of connections in $C_{\mathcal{S}}$. These are the number of connections that are scanned by the profile algorithms. The earliest arrival algorithm only needs to scan a subset of these connections because of the start and the stop criteria.

The preprocessing running times roughly grow with the number of lowest level cells. This is non-surprising, as the number of partitioner invocations follows this trend. The customization running times follow the same general trend and grow with the number of cells. However, having a large number of children helps the customization but hampers the partitioning. The minimum partitioning running times are therefore achieved for Germany-2-9 and London-3-3, which have a low number of children, whereas the customization running times are minimum for Germany-8-3 and London-8-2, i.e., a high number of children. To minimize the number of connections, a recursive bisection strategy with many levels performs best. Scanning fewer connections reduces the running time spent in the Connection Scan algorithm. Query running times are

	Low	Setup [s]			Query [ms]		
Instance	Cell	Part.	Cust.	Conn [K]	EA	EA-Prof	Par-Prof
Germany-2-9	512	**2 483.7**	157.7	897.2	6.6	49.1	**75.0**
Germany-2-12	4 096	7 300.5	329.1	**751.0**	**6.2**	**47.3**	78.9
Germany-3-5	**243**	2 604.8	114.5	1 184.6	7.2	56.7	85.9
Germany-3-7	2 187	4 918.8	220.3	868.8	6.5	51.0	79.3
Germany-4-5	1 024	3 746.7	157.7	1 023.4	7.2	55.6	84.6
Germany-4-6	4 096	7 214.2	229.1	988.9	7.0	57.1	89.0
Germany-8-3	512	3 170.2	**113.6**	1 331.4	7.9	66.2	99.3
Germany-8-4	4 096	7 367.9	176.0	1 252.2	7.7	67.5	102.3
London-2-7	128	253.5	101.2	1 933.6	2.6	**91.7**	**134.4**
London-2-10	1 024	838.1	126.6	**1 920.8**	2.6	96.3	137.5
London-3-3	**27**	**124.3**	54.1	2 181.2	2.5	99.2	140.6
London-3-5	243	338.3	74.1	2 085.0	2.3	92.6	137.5
London-4-3	64	230.0	51.7	2 226.2	2.3	95.0	141.2
London-4-5	1 024	718.6	67.1	2 193.6	2.2	97.1	141.5
London-8-2	64	186.7	**32.9**	2 490.1	2.0	97.7	147.9
London-8-3	512	579.9	40.6	2 464.9	**1.9**	97.1	147.0

Table 8.3: Preprocessing and customization running times, number of lowest level cells, number of connections in filter, and average query running times for earliest arrival time, earliest arrival profile, and Pareto profile. Preprocessing and customization were run on the older machine. Customization was parallelized with 16 threads.

therefore comparatively fast for nested dissection configurations. The only exception to this trend is the earliest arrival running time on London, which is fastest for London-8-3 and London-8-2. The explanation is that the k-way merge step dominates the running time. Having more levels results in more arrays to be merged and thus increases the running time of the merge step. London-8-3 and London-8-2 have the fewest levels and therefore the fastest merge steps.

Compared to the non-accelerated running times, we observe a significant decrease in running times for every query type on the Germany instance. However, the speedups are significantly less impressive on the London instance. The explanation is that the London instance only has 4 850K connections but even for London-2-10 1 921K connections have to be scanned. The speedup is therefore very slim. In fact for the earliest arrival time problem, the base algorithm is even faster. The explanation is that it is faster to scan the few additional connections, than to perform the k-way merge.

It is very surprising that CSAccel is faster in absolute terms on the Germany instance

Instance	Pareto	Running Time [ms]
Germany-2-12	○	17.9
Germany-2-12	●	24.7
London-2-7	○	11.2
London-2-7	●	12.0

Table 8.4: Accelerated range queries average running times.

compared to the London instance. There are several reasons for this effect. London has at the time of writing nearly 9M inhabitants. This contrasts with the largest German city Berlin that has only 3.5M inhabitants. As a consequence, the London urban transit is larger than any urban transit contained in the Germany instance. Another explanation is the difference in stop modeling. The London instance has a reflexive transfer model with usually one stop per platform. The Germany instance groups nearby platforms into one stop and uses loops in the footpath graph. London is thus modeled in greater detail than Berlin. Having more stops increases computation times.

8.4.2 Range Queries

In Table 8.4, we report range query results. We restrict our exposition to Germany-2-12 and London-2-7, as we obtained very good results for these configurations for non-range profile queries. Compared to the profile query running times, we observe significant speedups. These speedups are similar to those observed when comparing profile with range queries in the non-accelerated Connection Scan base algorithm. The speedups are due to cache effects and fewer connections being scanned.

8.4.3 Comparison with Related Work

In Table 8.5, we compare various algorithms for timetable routing. Some make use of very heavy-weight preprocessing, while others are very lightweight. We compare RAPTOR [52], our Connection Scan algorithm (CSA), our multilevel extension (CSAccel), public transit labeling (PTL,Pareto-PTL), [42], Trip-Based routing (TB) [144, 145], and transfer patterns (TP) [6, 11, 8]. Two PTL variants exist: the base version (PTL), and an extension that supports optimizing transfers in the Pareto-sense (Pareto-PTL). There are also two variants of Trip-Based routing: the base variant TB [144] and a newer version [145] (TB-ST) that precomputes prefix and suffix trees. Transfer patterns were introduced in [6] and overhauled in [11]. We refer to the overhauled version as TP. Another variant called "Scalable Transfer Patterns" was introduced in [8]. We refer to it as S-TP.

				Query Running Time [ms]			
	#Stop	#Conn	Prepro	Fixed-Dep		Profile	
Algo	[K]	[M]	[min]	EA	Pareto	EA	Pareto
RAPTOR [52]	252.4	46.2	—	—	325.8	—	4 730
CSA	252.4	46.2	0.1	44.9	259.2†	1 246	2 490
CSAccel-2-12	252.4	46.2	(122)+88	6.2	24.7†	47.3	78.9
TP [11]	248.4	13.9	22 320	—	0.3	—	5.0
S-TP [8]	250.0	15.0	990	—	32.0	—	—
TB-ST [145]	247.9	27.1	13 878	—	0.156	—	0.512
TB [144]	249.7	46.1	39	—	40.8	—	301.7
RAPTOR [52]	20.8	4.9	—	—	6.4	—	680
CSA	20.8	4.9	< 0.1	1.2	10.7†	106.9	170.2
CSAccel-2-7	20.8	4.9	(4)+27	2.6	12.0†	91.7	134.4
PTL [42]	20.8	5.1	54	0.0028	—	0.074	—
Pareto-PTL [42]	20.8	5.1	2 958	—	0.0266	—	—
TB-ST [145]	20.8	5.0	696	—	1.7	—	16.1
TB [144]	20.8	5.0	6	—	1.2	—	70.0

Table 8.5: Comparison of various preprocessing-based algorithms for timetable routing. The top results are for Germany instances and the bottom results for London instances.

The various papers use different instances that are based upon the same input data. The only exception is S-TP, which uses a newer version of the Deutsche Bahn data set. Unfortunately, the papers significantly differ in how they extract a formal timetable from the input. The variations on the London instance are comparatively small and originate from differences in how data errors are repaired.

The differences on the Germany instance are more significant. S-TP is based on newer input data than TP and therefore the corresponding numbers differ. The TP instance is based on the same input as the other papers.

CSAccel, TB, and TB-ST extract a two day instance. TP and S-TP extract a single day but have days-of-operation flags. Using these flags multiple days discerned. The difference between a two day instance and a one day instance with flags explains the different number of connections between TP and TB. The difference in size between TB-ST and TB originates from a different interpretation. Following our original CSAccel paper, TB extracts all connections regardless of the day of operation. This is done because some local operators do not have a schedule for every day. The downside of this approach is that several variations of the same trip appear simultaneously. For example some trips drive differently on Sundays than on workdays. Fortunately, having more connections will most likely not decrease the running times. The reported numbers of CSAccel and TB are therefore upper bounds. The difference between CSAccel and TB is the result of correcting data errors differently.

These differences in instances makes a detailed comparison difficult, if not impossible. We can only confidently compare orders of magnitude between the running times reported in the various papers. We therefore refrain from scaling running times with respect to machines as the numbers are not directly comparable anyway. Further, cache sizes can have a larger impact on the running time than the processor clock speed as demonstrated in Table 7.7. Unfortunately, cache sizes are rarely reported in papers. Scaling by processor clock speed is therefore not meaningful, even if the instances were equal.

All reported running times are sequentially. The reason that the preprocessing times seem large, stems from the fact that papers usually report parallelized running times. CSAccel is the only algorithm to split preprocessing into two phases. We therefore report its preprocessing as $(p) + c$ where p is the preprocessing and c the customization running time.

Unfortunately, we cannot report numbers for every query type and algorithm. This has various reasons. For RAPTOR, we do not report non-Pareto running times because RAPTOR does not benefit from not optimizing transfers. We report no preprocessing time for RAPTOR, because the original implementation that we use was not tuned for this criteria. For CSA, we report range query running times instead of non-profile Pareto running times. The reason is that we do not know how to implement non-profile Pareto queries in a way that significantly outperforms range queries. Range queries usually compute more journeys because they allow for a flexible departure time. In some sense the problem is therefore harder. However, the latest arrival time is bounded, which makes the problem also somewhat easier. Fortunately, both problems have similar applications and therefore we present the results in the same column. The CSA numbers are marked with a † to illustrate that range queries are computed. PTL's preprocessing can optionally optimize transfers. This explains the two PTL variants in the table. The authors evaluated the non-transfer variant for earliest arrival time and earliest arrival profiles. Unfortunately, the authors were not able to evaluate PTL on the Germany instance because of legal restrictions. Further, they did not evaluate Pareto-profile queries. The trip-based techniques TB and TB-ST, just as RAPTOR, do not benefit from not optimizing transfers in the Pareto-sense and thus no earliest-arrival-only numbers exist. The transfer patterns techniques TP and S-TP could in theory be implemented in a variant that only optimizes arrival time. This theoretical variant would probably benefit from smaller query graphs but it was, to the best of our knowledge, never implemented and thus we cannot report numbers. Unfortunately, TP was not evaluated on the London instance.

Discussion of the Germany instance. Ordering the algorithms by preprocessing running times yields: CSA, RAPTOR, TB, CSAccel, S-TP, TB-ST, and finally TP. With the exception of TB-ST and TP, the gaps between each of these techniques are large enough that we can be confident, that the differences are not solely due to differences in experimental setup. Comparing query running times is more difficult because of the various query types. Further, the differences between running times are smaller. It is thus possible that a number is only lower because of a different experimental setup. With respect to non-profile Pareto query running times, the group of fastest algorithms clearly contains TP and TB-ST. The next-slower group contains CSAccel, S-TP, and TB. The slowest group contains CSA and RAPTOR. Meaningfully comparing algorithms within a group requires a more similar experimental setup. Overall, CSAccel strikes a good trade-off between the various criteria. No query running time is above 100ms and preprocessing running times are manageable.

Discussion of the London instance. On the London instance, only PTL achieves a speedup above a factor of 11 over the CSA baseline. Given the simplicity and near-instant preprocessing running times, this makes CSA a perfect fit for this instance. PTL achieves an interesting performance trade-off when not optimizing transfers. The preprocessing time is slightly below an hour, which is still somewhat manageable. The benefit is that PTL achieves query running times are on the microsecond scale. Unfortunately, when additionally optimizing transfers the preprocessing running time of PTL becomes prohibitively large. Overall, assuming that some form of transfer optimization is required, we recommend using CSA as it is never drastically slower than the alternatives but is simple to implement and can update the timetable almost instantly.

8.5 Chapter Conclusion

The conclusions we draw from the experiments are mixed and depend on the test instance.

On the Germany instance, CSAccel can answer Pareto range queries on average in about 25ms. This is a significant improvement over the 250ms of CSA. Interactive timetable information systems with a high throughput can be constructed with an average query running times of 25ms. However, the factor 10 speedup comes at a high cost.

The obvious cost is the increased preprocessing time. CSA needs 10 seconds single core to adjust to a completely new timetable. On the other hand, CSAccel requires 2min with 16 cores. Requiring 2min to update the timetable is probably acceptable in practice but far from ideal. Further, CSAccel requires that the new timetable is sufficiently similar to the old one. CSA does not have this restriction. Again this restriction is probably acceptable in practice.

A further cost associated with CSAccel is the significant increase in code and algorithm complexity compared to CSA. Arguably the most important selling point of CSA is its simplicity. It is so simple that not even a heap-based priority queue is needed as a component. The earliest arrival CSA base is arguably even easier than Dijkstra's algorithm. CSAccel requires solving among other things a graph partitioning problem as subroutine. This is an NP-hard task and the state-of-the-art heuristics alone have a complexity far exceeding that of CSA. Depending on the application, the increase in complexity of CSAccel compared to CSA might even be worse than the increased preprocessing times.

However, for applications where query running times of 250ms are prohibitive and realtime updates are needed, CSAccel is still attractive because of the lack of alternatives. None of the other techniques achieves customization running times on the order of only a few minutes and similar query running times.

On the urban London network, the decrease in query running time of CSAccel over the CSA baseline is slim. We do not believe that it outweighs the significantly larger preprocessing costs and especially not the significant increase in code complexity. Use CSA in primarily urban networks.

An advantage of CSA is that its performance is nearly independent of the timetable structure and mostly depends on its size. On the other hand, the performance of CSAccel is heavily dependent on the timetable structure, as the differences between the test instances shows.

When setting up a new timetable information system, using CSA until the query running times get prohibitive is a good approach. CSA is easy to implement and therefore not much effort is lost when switching to other approaches. Further, chances are high that the size of your timetables will never reach the prohibitive size. For example, we have not been able to assemble a realistic timetable with only rail-bound vehicles that was large enough. The Germany test instance is only large enough because buses are included.

9 Minimum Expected Arrival Time

The Connection Scan profile framework is very flexible. In the previous sections, we have seen how the timetable can be adjusted to account for known delays. In this section, we want to plan ahead and compute a journey that is robust with respect to unknown, future delays. We do this by computing for every transfer backup journeys. For every transfer in a journey from train A to train B, we compute a list of backup trains $C_1, C_2 \ldots$ that the traveler can take if he cannot reach train B because A is delayed. If a transfer breaks, then a traveler should take the backup train with the earliest departure time that he can get.

An example of such a delay-robust journey from Karlsruhe to Berlin is depicted in Figure 9.1. We refer to the depicted graph as *decision graph*.

Furtherexamples can be generated using our proof-of-concept demonstration at `http://meatdemo.iti.kit.edu`. As we expect readers to be unfamiliar with the concept, we highly recommend to experiment with the demonstration to get a basic understanding on an intuitive level before reading on.

Computing such journeys is a very different setting than computing an earliest arrival journey with respect to a timetable aware of the realtime delay situation. All the algorithm's need to be performed in advance, when the exact delays are not yet known. To be able to do this, we assume that we have an estimation of how likely it is that a train is delayed. We refer to this estimation as *delay model*. One way to obtain such an estimation is to aggregate historic delay data. If a train was never delayed in the past, then we can assume that it is unlikely that it is delayed today. If a train was nearly always delayed, then we definitely need to have a good backup journey.

Consider a train A that is part of a very fast journey but no reasonable backup trains exist and A is likely delayed. A risk averse traveler will not want to take A because it is too risky. The algorithmically interesting question consists of identifying risky trains and avoiding them. Neither optimizing the arrival time nor the number of trains achieves this.

While developing the Connection Scan algorithms, we have discovered a surprisingly

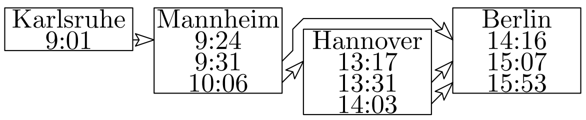


Figure 9.1: Delay-Robust journey from Karlsruhe at 9:00 to Berlin.

easy way to solve this problem. Consider the Connection Scan earliest arrival profile algorithm. Suppose that the traveler arrives with a train A, transfers at a stop X, to take a train B. We want to find the next best backup C in case that the transfer breaks. To compute this route, the Connection Scan profile algorithm looks up B's pair in X's profile. What will a traveler do, if he misses B? He will wait at X for the next train C heading into the correct direction to depart. Formulated differently, C is the backup train. Computing C is easy. The pair in X's profile after B's pair corresponds to C.

A problem remains. If no reasonable backup exists, i.e., the transfer is very risky, then the so computed backup C will arrive very late. To solve this issue, we do not store the arrival time of the next train in the profiles. Instead, we store the average over all trains in the profile weighted by the probability of the traveler taking the train. In probability theory, the expected value of a random variable is the average of all possible outcomes weighted by their probability. Following this terminology, we refer to the modified arrival times in our profiles as *expected arrival time*. If the first train B has an early arrival time but no good alternative exists, then the expected arrival time will be large. Minimizing the expected arrival time therefore solves the problem. We refer to the corresponding problem setting as *minimum expected arrival time (MEAT)* problem.

The decision graph in Figure 9.1 is tiny. Unfortunately, not all cities are as well connected as Karlsruhe and Berlin. Decision graphs between more remote areas can quickly grow in size and contain backups over numerous layers. We therefore investigate approaches to reduce the graph size and to represent it more compactly.

9.1 Related Work

There has been a lot of research in the area of train networks and delays. In contrast to our algorithm most of them compute single paths through the network instead of subgraphs containing all backups. To make this distinction clear we refer to such paths as *single-path-journeys*. The authors of [66] define the reliability of a single-path-journey and propose to optimize it in the Pareto-sense with other criteria such as arrival time or the number of transfers. The availability of backups is not considered. The authors of [29], based on delays occurred in the past, search for a single-path-journey that would have provided close to optimal travel times in every of the observed situations. Again, backups do not play a role. The authors of [83] propose to first compute a set of safe transfers (i.e. those that always work). They then develop algorithms to compute single-path-journeys that arrive before a given latest arrival time and only use safe transfers or at least minimize use of unsafe transfers. The problem with this is approach is that unsafe transfers are avoided at all costs. In the example of Figure 9.1, the direct train from Mannheim to Berlin would be missed because the transfer is unsafe. In [82], a robust primary journey is computed such that for every transfer stop a good backup single-path-journey to the target exists.

However, the backups do not have their own backups. The approach optimizes the primary arrival time subject to a maximum backup arrival time. The authors of [69] study the correlation between real world public transit schedules in Rom and compare them with the single-path-journeys computed by state-of-the-art route planners based on the scheduled timetable. They observe a significant discrepancy and conclude that one should consider the availability of good backups already at the planning stage. The authors of [9] examine delay-robustness in a different context: Having computed a set of transfer patters on a scheduled timetable in a urban setting, they show that single-path-journeys based on these patterns are still nearly optimal, even when introducing delays. The conclusion is that these sets are fairly robust (i.e., the paths in the delayed timetable often use the same or similar patterns). In [10] the authors propose to present to the user a small set of transfer patterns that covers most optimal journeys. They show that in an urban setting few patterns are enough to cover most single-path-journeys. In a different line of work, the authors of [20] investigate how a delay-perturbed timetable will evolve over time using stochastic methods. Their study shows that this is a computationally expensive task (running time in seconds) if the delay model accounts many real-world details. Using a model with such a degree of realism therefore seems unfeasible for delay-robust route planning (requiring query times in the milliseconds).

9.2 Delay Model

Every random variable X in this work is denoted by capital letters, is continuous, non-negative, and has a maximum value $\max X$. We denote by $P[X \leq x]$ the probability that the random variable is below some constant x and by $E[X]$ the expected value of X.

A crucial component of any delay-robust routing system is choosing against which types of delays the system should be robust and how to model these delays. This choice has deep implications throughout the whole system. While a too simplistic model does not yield useful routes, a too complicated model makes routing algorithms too inefficient to be useful in interactive timetable information systems. We therefore propose a simple stochastic model. While our model does not cover every situation and is not delay-robust in every possible scenario, it works well enough to give useful routes with backups. Further, we were not able to construct a proof-of-concept implementation for a more complex model while maintaining reasonable query running times.

The central simplification is that we assume that all random variables are independent. Clearly, in reality this is not always the case. However, if delays between many trains interact then the timetable perturbation must be significant. An example of a significant perturbation is a train track that is blocked for an extended period of time. As reaction to such a perturbation, even trains in the medium or distant future need to be rescheduled (or arrive at least not on-time). The set of possible outcomes and the associated uncertainty is huge. Accounting for every outcome seems infeasible to us. We argue that if the perturbation is large then we cannot account for all possible recovery scenarios in advance. Instead, the user should replan his journey based on the realtime delay situation. Furthermore, even if we could account for all scenarios, we would still face the problem of explaining every possible outcome to the user, which is a show-stopper in practice. Our model therefore only accounts for small disturbances as we only intend to be robust against these. We believe that assuming independence for small disturbances is a model simplification that is acceptable in practice.

Formally, our model contains one random variable $\mathcal{D}_c$ per connection c. This variable indicates with which delay the train will arrive at $c_{\text{arr\_stop}}$. We assume that all connections depart on time. This assumption does not induce a significant error because it roughly does not matter whether the incoming or the outgoing train is delayed. Furthermore, we assume that every connection c has a maximum delay, i.e., $\max \mathcal{D}_c$ is a finite value. Finally, we assume that all random variables are independent. Delays between trips are independent because if they were not then the perturbation would be large. We can assume that delays within a trip are independent, as there nearly never exists an optimal decision graph that uses a trip more than once.

We assume that the changing times at stops are encoded in $\mathcal{D}_c$. An transfer with a slack time below the regular change time should have a very low success probability but it should not be zero. This way the computed decision graph will also include the very risky transfers that in practice only have a chance of working if the outgoing train departs delayed. However, as the probability is low, not much weight is attributed to them. We further assume that inter-stop footpaths are handled by contracting adjacent stops and adjusting the change time. These simplifications allows us to omit footpath and change time handling from the algorithm. Fortunately, for applications that require them, they can be incorporated analogously to how they are handled in the earliest arrival profile Connection Scan algorithm.

The only remaining modeling issue is to define what distribution the random variables $\mathcal{D}_c$ should have. An obvious choice is to estimate a distribution based on historic delay data. However, this has two shortcomings:

- It is hard to get access to delay data (we do not have it), and
- you need to have records of many days with precisely the same planned schedule.

Suppose for example that the user is in the middle of his journey and a significant perturbation occurs. The operator then adjusts the short-term timetable to reflect this and the user wants to reroute based on this adjusted data. With historic data this often is not possible because this exact recovery scenario may never have occurred in the past and almost certainly not often enough to extrapolate from the historic data.

For these reasons, we propose to use synthetic delay distributions that are only parametrized on the planned timetable. We propose to add to each connection c a

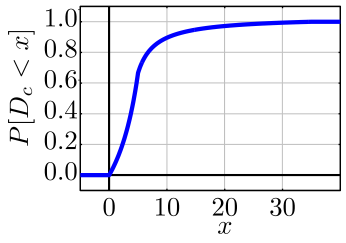


Figure 9.2: Plot showing $P[\mathcal{D}_c \leq x]$ in function of x for $m = 5$ and $d = 30$.

synthetic delay variable $\mathcal{D}_c$ that depends on the change time m of $c_{\text{arr\_stop}}$ and on a global[1] *maximum delay parameter d*. We define $\mathcal{D}_c$ as follows: $\forall x \in (-\infty, 0] : P[\mathcal{D}_c \leq x] = 0$, $\forall x \in (0, m] : P[\mathcal{D}_c \leq x] = \frac{2x}{6m-3x}$, $\forall x \in (m, m+d] : P[\mathcal{D}_c \leq x] = \frac{31(x-m)+2d}{30(x-m)+3d}$, and $\forall x \in (m+d, \infty) : P[\mathcal{D}_c \leq x] = 1$. The function is illustrated in Figure 9.2 and the rational for is given in the next section.

9.2.1 Synthetic Delay Distribution

There are many methods to come up with formulas for synthetic delays. The lack of any effectively accessible ground truth makes any conclusive experimental evaluation of their quality very difficult. The only real criteria that we have is “intuitively reasonable”. The approach presented here is by no way the final answer to the question of how to design the best synthetic delay distribution. In this section, we describe the rational for our design decisions.

We define for every connection c its delay $\mathcal{D}_c$ by defining its cumulative distribution function $f_{m,d}(x)$, where d is the maximum delay of c and m the minimum change time at $c_{\text{arr\_stop}}$. Our delays do not depend on any other parameter than m and d. We have the following hard requirements on $f_{m,d}$ resulting from our algorithm:

- $f_{m,d}(x)$ is a probability, i.e.,
- $f_{m,d}(x)$ is a cumulative distribution function and therefore
- max $\mathcal{D}_c$ should be $m + d$, i.e.,
- Our model does not allow for trains that arrive too early, i.e.,

These requirements already completely define what happens outside of $x \in (0, m + d)$. Because of the limitations of current hardware, there are two additional, more fuzzy but important requirements:

[1]d is global since we lack per-train data. Our approach can be adjusted, if such data became available.

- We need to evaluate $f_{m,d}(x)$ many times. The formula must therefore not be computationally expensive.
- Our algorithm computes a lot of $(f_{m,d}(x_1) + a_1) \cdot (f_{m,d}(x_2) + a_2) \cdots$ chains. The chain length reflects the number of rides in the longest journey considered during the computations. As 64-bit-floating points only have a limited precision, we must make sure that order of magnitude of the various values of $f_{m,d}$ do not differ too much. If they do differ a lot then the less likely journeys have no impact on the overall EAT because their impact is rounded away.

Finally there are a couple of soft constraints coming from our intuition:

- is the probability that everything works as scheduled without the slightest delay. In practice, this does happen and therefore this should have reasonable high probability. On the other hand a too high $f(m)$ can lead to problems with rounding. We set as we believe that it is a good compromise.
- We want to be continuous.
- The maximum variation should be at $x = m$.
- Initially, the function should grow and then once $x = m$ is reached the growth should slow down.

We define f_1 and f_2. For these pieces we assume $m = 5\text{min}$ and $d = 30\text{min}$ and scale them to accommodate for different values as follows:

$$f_{m,d}(x) = \begin{cases} 0 & \text{if } x < 0 \\ f_1(\frac{5x}{m}) & \text{if } 0 \le x \le m \\ f_2\left(\frac{30(x-m)}{d}\right) & \text{if } m < x < m + d \\ 1 & \text{if } m + d \le x \end{cases}$$

It remains to define f_1 and f_2. We started with a $-1/x$ function and shifted and stretched the function graphs until we ended up with something that looks “intuitively reasonable”.

$$\begin{aligned} f_1(x) &= \frac{2x}{3(10-x)} \\ f_2(x) &= \frac{31x + 60}{30(x+3)} \end{aligned}$$

The resulting function fulfills all requirements and is illustrated in Figure 9.2. To sum up: We define the $f_{m,d}$ as follows:

$$f_{m,d}(x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{2x}{6m-3x} & \text{if } 0 \leq x \leq m \\ \frac{31(x-m)+2d}{30(x-m)+3d} & \text{if } m < x < m+d \\ 1 & \text{if } m+d \leq x \end{cases}$$

9.3 Decision Graphs

In this subsection, we first introduce the notion of safe journey, then formally define decision graphs, and then introduce three problem variants: (i) the unbounded, (ii) the bounded, and (iii) the α-bounded MEAT problems. The first two are of more theoretical interest, whereas the third one has the highest practical impact. We prove basic properties of the unbounded and bounded problems and show a relation to the earliest safe arrival problem.

9.3.1 Formal Definition

A *safe* (s,τ_s,t)*-journey* is a (s,τ_s,t)-journey, such that for every transfer the time difference between the arrival of the incoming train and the departure of the outgoing train is at least the maximum delay of the incoming train. We denote by $\text{eat}(s,\tau_s,t)$ the arrival time of an optimal earliest arrival journey and by $\text{esat}(s,\tau_s,t)$ the arrival time of an optimal safe earliest arrival journey.

A (s,τ_s,t)*-decision graph* from source stop s to target stop t with the traveler departing at time τ_s is a directed loop-free multi-graph $G = (V, A)$, whose vertices correspond to stops and whose arcs correspond to legs l directed from $l_{\text{dep\_stop}}$ to $l_{\text{arr\_stop}}$. There may be several legs between a pair of stops, but they depart at different times. We formalize this as: $\forall l^1, l^2 \in A : l^1_{\text{dep\_time}} \neq l^2_{\text{dep\_time}} \vee l^1_{\text{dep\_stop}} \neq l^2_{\text{dep\_stop}}$. We require that the user must be able to reach every leg and must always be able to get to the target. Formally, we require that for every $l \in A$ there exists a $(s,\tau_s,l_{\text{dep\_stop}})$-journey j with $j_{\text{arr\_time}} \leq l_{\text{dep\_time}}$ to reach the leg, and a safe $(l_{\text{arr\_stop}}, l_{\text{arr\_time}} + \max \mathcal{D}_r, t)$-journey j' to reach the target. To exclude decision graphs with unreachable stops, we require that every stop in V except s and t have non-zero in- and out degree.

We first recursively define the *expected arrival time* $e(l)$ (short EAT) of a leg $l \in A$ and define in terms of $e(l)$ the EAT $e(G)$ of the whole decision graph G. If $l_{\text{arr\_stop}} = t$, we define $e(l) = l_{\text{arr\_time}} + E[\mathcal{D}_l]$. Otherwise, $e(l)$ is defined in terms of other legs. Denote by $q_1 \ldots q_n$ the sequence of legs in G ordered by departure time, departing at $l_{\text{arr\_stop}}$ after $l_{\text{arr\_time}}$, i.e., every leg that the user could reach after l arrives. Denote by $d_1 \ldots d_n$ their departure times and set $d_0 = l_{\text{arr\_time}}$. We define $e(l) = \sum_{i\in\{1\ldots n\}} P[d_{i-1} < \mathcal{D}_l < d_i] \cdot e(q_i)$, i.e., the average of the EATs of the connecting legs weighted by the transfer probability. This definition is well-defined because $e(l)$ only depends on $e(q)$ of legs with a later departure time, i.e., $l_{\text{dep\_time}} < q_{\text{dep\_time}}$. Further, $P[\mathcal{D}_l < d_n] = 1$. Otherwise, no safe journey to the target would exist invalidating the decision graph.

We denote by G^{first} the leg $l \in A$ with minimum $l_{\text{dep\_time}}$. This is the leg that the user must initially take at s. We define the *expected arrival time* $e(G)$ (short EAT) of the decision graph G as $e(G^{\text{first}})$. Furthermore, the *latest arrival time* $G_{\text{max arr\_time}}$ is the maximum $l_{\text{arr\_time}} + \max \mathcal{D}_l$ over all $l \in A$. By minimizing $G_{\text{max arr\_time}}$, we can bound the worst case arrival time giving us some control over the arrival time variance.

The *unbounded* (s, τ_s, t)*-minimum expected arrival time* (short MEAT) problem consists of computing a (s, τ_s, t)-decision graph G minimizing $e(G)$. The bounded (s, τ_s, t)-*MEAT* problem consists of computing a (s, τ_s, t)-decision graph G, minimizing $e(G)$ subject to a minimum $G_{\text{max arr\_time}}$. As a compromise between bounded and unbounded, we further define the α-bounded MEAT problem: We require that $G_{\text{max arr\_time}} - \tau_s \leq \alpha\,(\text{esat}\,(s, \tau_s, t) - \tau_s)$, i.e., the maximum travel time must not be bigger than α times the delay-free optimum. The bounded and 1-bounded MEAT problems are equivalent.

9.3.2 Decision Graph Existence

Lemma 4. There is a (s, τ_s, t)-decision graph G, if and only if there exists a safe (s, τ_s, t)-journey j.

Proof. If there exists a (s, τ_s, t)-decision graph G then by the decision graph definition we know that there exists a safe $(G^{\text{first}}_{\text{arr\_stop}}, G^{\text{first}}_{\text{arr\_time}} + \max \mathcal{D}_{G^{\text{first}}}, t)$-journey j'. Prefixing j' with G^{first} yields the required (s, τ_s, t)-journey j.

Conversly, if there exists a (s, τ_s, t)-journey j, we can construct a (non-optimal) (s, τ_s, t)-decision graph G that contains exactly the same legs as j. □

A direct consequence of this lemma is that the minimum $G_{\text{max arr\_time}}$ over all (s, τ_s, t)-decision graphs G is equal to $\text{esat}(s, \tau_s, t)$. Using this observation, we can reduce the bounded MEAT problem to the unbounded MEAT problem. Formally stated:

Lemma 5. An optimal solution G to the bounded (s, τ_s, t)-MEAT problem on timetable T is an optimal solution to the unbounded (s, τ_s, t)-MEAT problem on a timetable T' where T' is obtained by removing all connections c from T with $c_{\text{arr\_time}}$ above the $\text{esat}(s, \tau_s, t)$.

Proof. There are two central observations needed for the proof: First, every (s, τ_s, t)-decision graph on timetable T' is a (s, τ_s, t)-decision graph on the strictly larger timetable T. Second, every safe (s, τ_s, t)-journey in T' is an earliest safe (s, τ_s, t)-journey in T. Suppose that a (s, τ_s, t)-decision graph G' on T' would exist with a suboptimal $G'_{\text{max arr\_time}}$, then there would also exist a safe (s, τ_s, t)-journey j' in T' with a suboptimal $j'_{\text{arr\_time}}$, which is not possible by construction of T', which is a contradiction. □

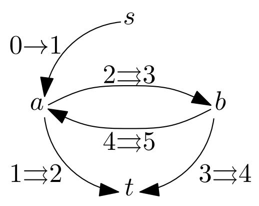


Figure 9.3: A timetable T_p has 4 stops: s, a, b and t. The arrows denote connections. An arrow is annotated with its departure time and arrival time. A simple arrow ($\rightarrow$) denotes a single non-repeating connection. A double arrow ($\rightrightarrows$) is repeated every 4 time units, i.e. $1 \rightrightarrows 2$ is a shorthand for $1 + 4i \rightarrow 2 + 4i$ for every $i \in \mathbb{N}$. All connections are part of their own trip and have the same delay variable $\mathcal{D}$. We define $P[\mathcal{D} = 0] = p$ (with $p \neq 0$) and $P[\mathcal{D} < 1] = 1$.

Having shown how to explicitly bound $G_{\max \text{arr\_time}}$, it is natural to ask what would happen if we dropped this bound and solely minimized $e(G)$. For this, we consider the timetable T_p with an infinite connection set illustrated and defined in Figure 9.3. T_p is constructed such that it does not matter whether the user arrives at a at moments $1 + 4\mathbb{N}$ or at b at moments $3 + 4\mathbb{N}$ as the two states are completely symmetric with the stops a and b swapping roles. By exploiting this symmetry, we can reduce the set of possibly optimal $(s, 0, t)$-decision graphs to two elements: the decision graph G^1 that waits at a and never goes over b, and the decision graph G^2 that oscillates between a and b. The corresponding expected arrival times are $e(G^1) = p\,(2 + E[\mathcal{D}]) + (1 - p)\,(7 + E[\mathcal{D}])$ and $e(G^2) = p\,(2 + E[\mathcal{D}]) + (1 - p)\,\big(3 + e(G^2)\big)$. The later equation can be resolved to $e(G^2) = E[\mathcal{D}] - 1 + \frac{3}{p}$. We can solve $e(G^1) < e(G^2)$ in terms of p. The result is that G^1 is better if $p < \frac{\sqrt{43}-4}{9} \approx 0.28$. If they are equal, then G^1 and G^2 are equivalent, otherwise G^2 is better.

This has consequences even for timetables with a finite connection set. One could expect that to compute a decision graph, it is sufficient to look at a time-interval proportional to its expected travel time: It seems reasonable that a connection scheduled to occur in ten years would not be relevant for a decision graph departing today with an expected travel time of one hour. However, this intuition is false in the worst case: Consider the finite sub-timetable T' of the periodic timetable T_p that encompasses the first ten years (i.e., we “unroll” T_p for ten years). For $p{>}0.28$, an optimal $(s, 0, t)$-decision graph will use all connections in T', including the ones in ten years (as G^2 would). Fortunately, the bounded MEAT problem does not suffer from this weakness: No connection arriving after $\text{esat}(s, 0, t)$ can be relevant. Therefore, even on infinite networks the bounded MEAT problem always admits finite solutions. This property is the main motivation to study the bounded MEAT problem.

9.3.3 Non-dominated Pairs and Decision Graphs

In this section, we only consider decision graphs and journeys arriving at a fixed target stop t. All lemmas and definition are therefore with respect to t. To simplify our notation, we omit t in this section.

We consider, for every connection c, the pair $p_c = (c_{\text{dep\_time}}, e(G))$ where G is a decision graph that minimizes the expected arrival time, subject to c being the first connection, i.e., $G^{\text{first}}_{\text{enter}} = c$. Denote by O the outgoing connections of a stop. Every connection has an associated pair, which can be dominated within O. This allows us to define when a connection is dominated: It is dominated when its pair is dominated. A leg l is dominated if l_{enter} is dominated. Non-dominated connections have an important role in the computation of optimal decision graphs as the following lemma shows.

Lemma 6. For every source stop s and source time τ_s, if there exists a decision graph, then there exists an optimal decision graph, such that for every leg l of G the entry connection l^{enter} is non-dominated at $l^{\text{enter}}_{\text{dep\_stop}}$.

Proof. We know that an optimal decision graph H exists as we required the existence of a decision graph. If $H^{\text{first}}_{\text{enter}}$ is dominated, then there is another optimal decision graph associated with the dominating connection. Without loose of generality we can therefore assume that $H^{\text{first}}_{\text{enter}}$ is non-dominated.

Suppose that H contained some other leg l such that l^{enter} is dominated. Further denote by l' an incoming leg from which the traveler might transfer to l. l' must exist because l is not the first leg in the decision graph. As l is dominated, removing it and all legs that can only reached via l from H improves $e(l')$, which in terms improves $e(H)$, which is a contradiction to H being optimal. □

9.4 Solving the Minimum Expected Arrival Time Problem

The unbounded MEAT problem can be solved to optimality on finite networks, and by extension also the bounded and α-bounded MEAT problems. We first describe an algorithm to optimally solve the unbounded MEAT problem. By applying this algorithm to a restricted timetable we solve the bounded and α-bounded MEAT problems.

9.4.1 Solving the Unbounded problem

Our algorithm works in two phases:

- Compute the minimum expected arrival times for all connections c,
- extract a desired (s, τ_s, t)-decision graph.

The first phase is a variant of the earliest arrival profile Connection Scan algorithm. The second phase is an extension of the journey extraction algorithm.

9.4.1.1 Phase 1: Computing all Expected Arrival Times

Recall the basic Connection Scan profile framework depicted in Algorithm 7.1 and especially the earliest arrival time instantiation depicted in Algorithm 7.2. We first describe the algorithmic differences to the later and then explain why the proposed algorithm is correct. In the context of this subsection c always refers to the connection being scanned.

The first key idea consists of replacing all earliest arrival times with minimum expected arrival times. This works similarly to the profile Pareto-optimization where all earliest arrival times were replaced by vectors. The stop data structure becomes an array of dynamic arrays of pairs of departure time and expected arrival time. The trip data structure becomes an array of expected arrival times. The computation of the expected arrival time, when arriving at the target τ_1, is only modified in a minor way: We need to add $E\mathcal{D}_c$, a constant, to the arrival time of the connection. The arrival time when the traveler remains sitting τ_2 is computed in exactly the same way by reading the value of $T[c_{\text{trip}}]$. The computation of the arrival time when changing trains τ_3 is significantly modified and is described below. The value of τ_c is still computed as the minimum of τ_1, τ_2, and τ_3. τ_c is the minimum expected arrival time over all decision graphs starting in c. Formulated differently, τ_c is the minimum $e(G)$ over all decision graphs such that $G^{\text{first}}_{\text{enter}} = c$. Incorporating τ_c into the trip data structure T and the stop profiles S works completely analogous to the earliest arrival profile algorithm.

The computation of τ_3, i.e., the computation of the arrival time when transferring trains is changed. The reason for this change is that the arriving train c has a random arrival time between $c_{\text{arr\_time}}$ and $c_{\text{arr\_time}} + \max \mathcal{D}_c$. Our algorithm starts by determining, using a sequential scan, all pairs $p^1 \dots p^k$ in the profile $S[c_{\text{arr\_stop}}]$ that might be relevant. These are all pairs departing between $c_{\text{arr\_time}}$ and $c_{\text{arr\_time}} + \max \mathcal{D}_c$ and the first pair after $c_{\text{arr\_time}} + \max \mathcal{D}_c$. These correspond to all outgoing trains that are worth taking. It then computes τ_3 as the weighted sum over the expected arrival times of all p^i. A pair is weighted by the probability of the incoming being delayed in such a way that the traveler will take it. Formally this means: p^1 is weighted by the probability $P[c_{\text{arr\_time}} + \max \mathcal{D}_c \leq p^1_{\text{dep\_time}}]$ and all other p^i are weighted by $P[p^{i-1}_{\text{dep\_time}} \leq c_{\text{arr\_time}} + \max \mathcal{D}_c \leq p^i_{\text{dep\_time}}]$. Formulated differently, τ_2 is the average over the expected arrival time of the non-dominated outgoing trains, weighted by the probability of the traveler transferring to them.

The correctness of our algorithm relies on optimal decision graphs not containing any dominated legs as shown in Lemma 6. The domination test in the profile insertion filters dominated pairs and pairs which appear several times. In the later case there are two or more connections that depart at the same time and have the same expected arrival time. In this case, it does not matter which we insert into the decision graph but we may only insert one. Our algorithm picks the connection that appears last in the connection array.

It remains to show, why our strategy of selecting all outgoing non-dominated connections during the evaluation is optimal. This directly follows from the pairs being ordered. One does not want to skip earlier pairs because they have lower expected arrival times than the later trains. One cannot remove the later trains because it is not guaranteed that the earlier trains can be reached. Connection not in the profile are dominated. From Lemma 6 follows that we can ignore dominated connections.

9.4.1.2 Phase 2: Extracting Decision Graphs

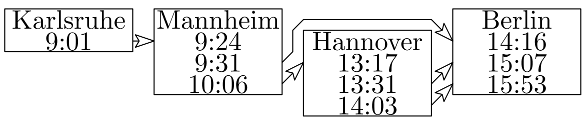
We extract a (s, τ_s, t)-decision graph $G = (V, A)$ by enumerating all legs in A. The stop set V can then be inferred from A. At the core, our algorithm uses a min-priority queue that contains connections, ordered increasing by their departure time. Initially, we add the earliest connection in the profile of s to the queue. While the queue is not empty, we pop the earliest connection c^1 from it. Denote by $c^2 \dots c^n$ all subsequent connections in the trip c^1_{trip}. The desired leg $l = (c^1, c^i)$ is given by the first i such that $e(c^1) \neq e(c^{i+1})$ (or $i = n$ if all are equal). We add l to G. If $c^i_{\text{arr\_stop}} \neq t$, we add the following connections to the queue: (i) All connections in the profile of $c^i_{\text{arr\_stop}}$ departing between $c^i_{\text{arr\_time}}$ and $c^i_{\text{arr\_time}} + \max \mathcal{D}_{c^i}$, and (ii) the first connection in the profile of $c^i_{\text{arr\_stop}}$ departing after $c^i_{\text{arr\_time}} + \max \mathcal{D}_{c^i}$.

9.4.2 Solving the α-Bounded Problem

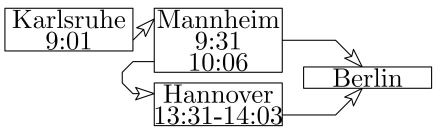
We assume that the connection set is stored as an array ordered by departure time. To solve the α-bounded (s, τ_s, t)-MEAT problem, we perform the following steps: (i) Run a binary search on the connection set to determine the earliest connection c^{first} departing after τ_s. (ii) Run a one-to-one Connection Scan from s to t that assumes all connections c are delayed by $\max \mathcal{D}_c$ to determine $\operatorname{esat}(s, \tau_s, t)$. (iii) Let $\tau_{\text{last}} = \tau_s + \alpha \cdot (\operatorname{esat}(s, \tau_s, t) - \tau_s)$ and run a second binary search on the connection set to find the last connection c^{last} departing before τ_{last}. (iv) Run a one-to-all Connection Scan from s restricted to the connections from c^{first} to c^{last} to determine all $\operatorname{eat}(s, \tau_s, \cdot)$. (v) Run Phase 1 of the unbounded MEAT algorithm scanning the connections from c^{last} to c^{first} skipping connections c for which $c_{\text{arr\_time}} > \tau_{\text{last}}$ or $\operatorname{eat}(s, \tau_s, c_{\text{dep\_stop}}) \leq c_{\text{dep\_time}}$ does not hold. (vi) Finally, run Phase 2 of the unbounded MEAT algorithm, i.e., extract the (s, τ_s, t)-decision graph.

9.5 Decision Graph Representation

In the previous section, we described how to compute decision graphs. In practice this is not enough and we must be able to represent the graph in a form that the user can effectively comprehend. The main obstacle here is to prevent the user from being



(a) Expanded



(b) Compact

Figure 9.4: Decision graph representations from Karlsruhe at 9:00 to Berlin.

overwhelmed with information. A secondary obstacle is how to actually layout the graph. In this section, we solely focus on reducing the amount of information. The presented drawings were created by hand. In the demonstration we use GraphViz [68].

9.5.1 Expanded Decision Graph Representation

Figure 9.4(a) illustrates the expanded decision graph drawing style. It subdivides each node v into slots $s_{v,1} \dots s_{v,n}$ that correspond to moments in time that an arc arrives or departs at v. The slots in each node are ordered from top to bottom in chronological order. Each arc (u, v) connects the corresponding slots $s_{u,i}$ and $s_{v,j}$. To determine his next train the user has to search for the box, corresponding to his current stop and pick the first departure slot after the current moment in time. The arrows guide him to the box corresponding to his next stop.

9.5.2 Compact Decision Graph Representation

The scheduled arrival time of trains is an information contained in the expanded decision graph that is not strictly necessary. A traveler decides what outgoing train to take when he arrives. At that moment, he can look at any clock to figure out the precise arrival time. The scheduled arrival time, recorded in the timetable, is not needed for his decision.

Figure 9.4(b) illustrates the compact decision graph drawing style. It exploits this observation by removing the arrival time information from the representation. Each arc (u, v) connects the corresponding departure slot $s_{u,i}$ directly to the stop v instead of a slot. Time slots that only appear as arrival slots are removed. If two outgoing arcs of a node u have the same destination and depart subsequently, they are grouped and only displayed once. The compact decision graph is never larger than the expanded one and most of the time significantly smaller.

9.5.3 Relaxed Dominance

Decision graphs exist that contain legs that have near to no impact on the EAT. Removing them increases the EAT by only a small amount, resulting in an almost optimal decision graph that can be significantly smaller. To exploit this, we introduce a *relaxation tuning parameter* β. Formulated in terms of the framework depicted in Algorithm 7.1, we only insert a new pair into the profile $S[x]$, if the expected arrival time of the earliest pair of $S[x]$ is at least β time units later than τ_c.

9.5.4 Displaying only the Relevant Subgraphs

In many scenarios, we have a canvas of fixed size. If even the compact relaxed decision graph is too large to fit, we can only draw parts of it. We observe that the decision graph extraction phase does not rely on the actual distributions of the delay variables $\mathcal{D}_c$ but only on $\max \mathcal{D}_c$. It extracts all connections departing in an interval I, plus the first connection directly afterwards. The full decision graph is extracted when $I = [c_{\text{arr\_stop}}, c_{\text{arr\_stop}} + \max \mathcal{D}_c]$. Reducing the size of I reduces the number of legs displayed, while still guaranteeing that backup legs exist. For example a smaller partial decision graph is extracted, if we only follow the connections departing in $I = [c_{\text{arr\_stop}}, c_{\text{arr\_stop}} + \kappa]$ for $\kappa = 1/2 \cdot \max \mathcal{D}_c$. Valid values for κ are from 0 to $\max \mathcal{D}_c$. We refer to κ as *display window*. Given an upper bound γ on the number of arcs in the compact or expanded representation, we use a binary search to determine the maximum display window κ and draw the corresponding subgraph. In the worst case, the display window has size zero. In this case, the decision graph degenerates to a single-path-journey.

9.6 Experiments

For our experiments, we used a single core of a Xeon E5-2670 at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. This is the "older" machine used in the experiments of the previous sections. We implemented the algorithm in C++ and compiled it using GCC 4.7.1 with -O3.

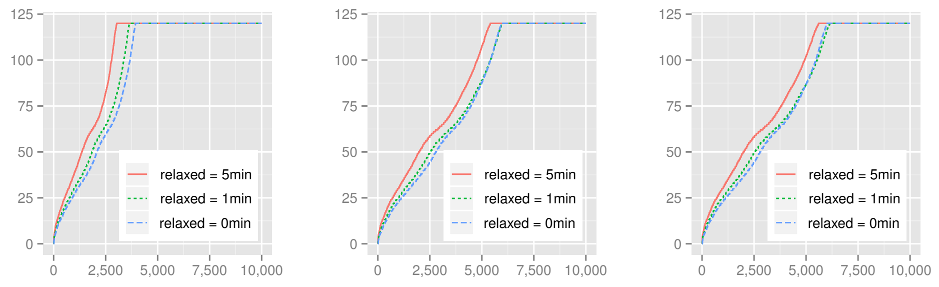
The timetable is based on the data of `bahn.de` during winter 2011/2012. This is the same primary data source as used for the experiments of Section 8.4. However, we extracted a different formal timetable. We extracted every vehicle except for most buses as we mainly focus on train networks. Not having buses explains the significant instance size difference compared to the Germany instance of the previous sections. Not having buses allows us to get the running times onto a manageable level.

		Unbounded				2.0-Bounded				1.0-Bounded			
		Time	Stops	Legs	Arcs	Time	Stops	Legs	Arcs	Time	Stops	Legs	Arcs
0min-Relax	Avg	6 452	12	98	42	138	12	87	35	26	9	45	19
	33%	6 209	7	22	10	84	7	22	10	16	7	15	7
	66%	7 407	13	70	31	162	13	69	31	27	10	40	19
	95%	7 635	25	349	125	312	24	330	119	66	19	149	57
	Max	7 805	280	35 450	28 848	817	173	5 540	4 703	288	38	1 607	366
1min-Relax	Avg	5 122	12	88	39	116	12	73	31	25	9	39	17
	33%	4 628	8	26	12	75	8	25	12	16	6	14	7
	66%	6 026	13	66	31	136	13	64	30	26	10	36	17
	95%	6 368	24	284	110	249	24	257	100	64	18	123	52
	Max	6 595	50	12 603	6 558	685	50	1 576	478	240	37	1 390	289
5min-Relax	Avg	4 180	11	66	33	100	11	51	25	24	9	29	15
	33%	3 845	8	24	12	66	8	23	11	15	6	13	6
	66%	4 808	13	53	26	115	12	51	25	25	10	30	15
	95%	5 028	22	178	82	216	22	155	74	61	17	84	42
	Max	5 159	54	6 640	3 220	553	54	760	285	196	34	590	183

Table 9.5: The time (in ms) needed to compute a decision graph and its size. Arcs is the number of arcs in the compact representation. The number of rides corresponds to the number of arcs in the expanded representation. The maximum delay parameter is set to 1h. We report average, maximum and the 33%-, 66%- and 95%-quantiles.

#Stop	16 991
#Conn.	55 930 920
#Trip	3 965 040

Table 9.6: Instance Size.



(a) 1.0-Bounded (b) 2.0-Bounded (c) Unbounded

Figure 9.7: Display windows in minutes (y-axis) for each of the 10 000 test queries (x-axis), ordered increasingly. The maximum delay parameter is set to 2h.

Further, it allows us to focus on long-distance trains where delays have a significantly larger impact than in high-frequent inner-city transit. We removed footpaths longer than 10 min, connected stops with a distance below 100 m, and then contracted stops connected through footpaths adjusting their minimum change times resulting in an instance without footpaths. Not having footpaths again benefits query running times. We pick the largest strongly connected component to make sure that there always exists a journey (assuming enough days are considered). We extract one day of maximum operation (i.e. extract everything regardless of the day of operation and remove exact duplicates). We then replicated this day 30 times to have a timetable spanning about one month of operation. The detailed sizes are in Table 9.6. We ran 10 000 random queries. Source and target stop are picked uniformly at random. The source time is chosen within the first 24h. We filter queries out that have an time above 24h.

Our experimental results are presented in Table 9.5. The compact representation is smaller by a factor of 2 in terms of arcs than the expanded one. As expected, a larger relaxation parameter gives smaller graphs. Increasing the α-bound leads to larger graphs and running times grow. The running times of unbounded queries are proportional to the timespan of the timetable (i.e. 30 days). On the other hand, the running times of bounded queries depend only on the maximum travel time of the journey. This explains the gap in running time of two orders of magnitude. As the maximum values are significantly higher than the 95%-quantile, we can conclude that the graphs are in most cases of manageable size with a few outlines that distort the average values. Upon closer manual inspection, we discover that most outliers with large decision graphs connect remote rural areas, where even no “good” delay-free journey exists. We can therefore not expect to find any form of robust travel plan.

In Figure 9.7, we evaluate the value of the display window such that the extracted graphs have less than 25 arcs in the compact representation. Recall that this modifies what is displayed to the user. It is still guaranteed that backups exist. As the 1.0-bounded graphs are smaller than 2.0-bounded graphs we can display more, explaining the larger display window. The difference between 2.0-bounded graphs and unbounded graphs is small. A greater relaxation parameter also reduces the graph size and thus allows for slightly larger display windows. If there is no “good” way to travel, the decision graphs degenerate to single-path-journeys.

9.7 Chapter Conclusion

We described the Minimum Expected Arrival Time (MEAT) problem and described an efficient CSA-based algorithm to solve it. This demonstrates that the CSA-framework is very flexible and can be adapted to complex problem settings. The achieved query running times of 100ms on average are fast enough for interactive systems. This is further demonstrated by our proof of concept implementation accessible at `http://meatdemo.iti.kit.edu`.

However, the fast query running times were bought by removing most buses from the instance. For the full Germany instance, the running times are prohibitively large. Fortunately, decision graphs make most sense in long-distance travel, where most high-frequency local bus lines do not play a role. The size of the computed decision graphs can become large. careful engineering it is possible to sufficiently reduce their size to a manageable value.

Overall, we believe that the MEAT problem and our CSA-based algorithm are a promising basis on which an innovative timetable information system can be built.

10 Conclusion

10.1 Summary

In this thesis, we study various variations of the adaptive routing problem using the Algorithm Engineering methodology. The objective is to efficiently compute a route from a source location to a target location. We study road- and timetable-based networks. We consider realtime and predicted changes to the network.

To solve the road-based realtime routing problem, we introduce a technique called Customization Contraction Hierarchies (CCH). In an extensive theoretical and experimental analysis, we demonstrate that CCH has good worst case performance bounds and works well in practice on realworld data. We further demonstrate that the CCH running time performance, with the exception of the path extraction, is independent of graph weights used. This means that the technique can efficiently adapt to any traffic situation. This even includes absurd unrealistic corner cases.

We further propose a heuristic approach named TD-S to predicted changes in road graphs. In an experimental study on a recent production-grade instance, we show that TD-S is fast and finds paths that are small enough for practical purposes. A technique that always finds a shortest path would be superior, however, our study also shows that none of the competitor algorithms achieves this goal either.

To handle timetable-based routing problems, we introduce the Connection Scan Algorithm (CSA). It is a simple and very flexible algorithm. One of its strengths is that the preprocessing phase essentially consists of sorting the vehicles in the timetable. As sorting is an efficient operation, most CSA-based algorithms are capable of handling realtime changes to the timetable. While CSA has many very nice properties, the average query running times on large instances can be larger than 100ms. For certain applications, these query running times can be less than ideal. We therefore combine CSA with a multilevel partitioning scheme and obtain CSAccel. CSAccel trades many of the advantages of CSA such as simplicity for a decreased average query running time. Depending on the application, this trade-off might be more useful than the basic CSA algorithm.

In addition to many routing related algorithms, we study the graph bisection problem. This problem appears as a subproblem to the CCH preprocessing. An improved graph bisection directly translates to improved CCH performance with respect to every criteria. We introduce a novel graph bisection algorithm named FlowCutter and perform a detailed experimental analysis. Our study demonstrates that FlowCutter is capable of finding high-quality cuts in graphs originating from various applications. FlowCutter finds small, balanced cuts in road graphs but is not limited to this application.

We further show that CCH and tree decompositions are deeply coupled. By exploiting this relation, we can use FlowCutter to compute tree decompositions. We entered FlowCutter to the 2016 PACE challenge [39] and won[1] in the relevant category.

10.2 Outlook

Further Research into Contraction Hierarchies. The highway dimension theory [1] and skeleton dimension theory [96] provide partial explanations for why a CH with witness search works well. However, they are not consistent with the theoretical bounds that we achieved for a CH without witness search. Both depend on the weighted diameter, while our theory depends on the unweighted diameter. Further, the highway dimension theory does not explain why a CH works well on tree graphs. An interesting avenue for further research is therefore, in our opinion, to investigate, whether there exists a graph measure d that depends on the graph and the weights and has the following properties:

- $d \leq \text{tw}$, where tw is the tree width.
- There exists a contraction order, such that the query running time without path extraction of the corresponding CH with witness search is bounded by $O((d \log n)^2)$.

The idea of the first property is that having additional information cannot make the CH performance worse. The second property is obtained by replacing tw with d in the running time that we have proven.

We can significantly accelerate CH query running times in the CCH context by using elimination trees. Unfortunately, the definition of this tree is inherently independent of the weights. It therefore seems worthwhile to investigate whether a definition variation exists that depends on the weights.

In Chapter 2, our experiments clearly show that the number of edges in a weight-dependent CH can be smaller than in a weight-independent CCH. However, it is unknown how big this gap can get. Is there a non-trivial, worst-case bound on the gap?

Further Research into Road-based Routing with Predicted Congestions. In Chapter 5, we introduce a heuristic named TD-S that computes routes in road-based networks with predicted congestions. It works well in practice. However, it is very simplistic and it seems as if one should be able to do better with more complex algorithms.

[1] FlowCutter won in the category for sequential algorithms. It came second in the category for parallel algorithms.

The fact that TD-S+P seems to be the only viable approach in practice on a current Europe instance to compute profiles, suggests that there is room for improvement.

Many existing, more complex techniques operate by attaching travel time profile functions to edges. The representations of these functions becomes large and therefore requires a lot of memory. Our experiments show that there usually do not seem to be many optimal paths throughout a day for a given source and target location. An approach that therefore tries to store the variations in the optimal paths instead of the variations in the travel times might thus require significantly less memory.

Further Research into Graph Bisection and FlowCutter. We demonstrate that small, balanced edge cuts and node separators can be found using FlowCutter in unweighted graphs. It is open whether FlowCutter can be extended to weighted graphs. Another avenue for further research is to investigate whether FlowCutter can be used to bisect hypergraphs.

Our research has demonstrated that FlowCutter can be used to compute small tree decompositions of large graphs. There exist a lot of NP-hard problems that are fixed-parameter tractable (FPT) in the tree width [23] such as for example the maximum independent set problem or the minimum vertex cover problems. It seems thus worthwhile to investigate whether FlowCutter can be combined with such FPT algorithms to obtain an algorithm that works well in practice. Unfortunately, we do not expect that executing the FPT algorithm after executing FlowCutter will be fast enough. While the obtained decomposition widths are often small compared to the graph sizes, they seem too large to execute as algorithm whose running time exponentially depends on the decomposition width. We therefore expect, that a further refinement with problem specific heuristics is necessary.

Further Research into Timetable-Based Networks. All timetable routing papers known to us, artificially restrict how far a passenger can walk when switching trains. Ideally, the input of timetable information systems should be the timetable and a large connected walking graph. In this graph, it should be possible to walk between arbitrary positions. For example, it should be possible to walk from Karlsruhe to Berlin. The system should be able to figure out, without further input or configuration, that walking from Karlsruhe to Berlin without taking a train is not a good route. In current generation systems, this is achieved either by bounding the maximum walking time by some arbitrary configuration-dependent constant or by having a highly disconnected walking graph. We use the later. Both approaches seem less than ideal. Unfortunately, without either of these restrictions, the query running times or the space consumptions tend to drastically increase. For example, our algorithm would require a quadratic matrix. Developing an algorithm that does not suffer from this weakness would thus significantly advance the state of the art in this area.

At first glance, unrestricted walking does not seem like an important feature in practice. The existing approaches seem to get the job done. However, from an algorithmic point of view, a taxi is no different from a passenger that can walk very fast. Being able to efficiently plan multimodal routes that encompass taxis and trains is relevant in practice. Computing multimodal routes with taxis, trains, and walking is difficult. This insight is not new and was already observed in the past. For example, the reported query running times of [41] drastically increase when they consider taxis.

Bibliography

[1] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. “Highway dimension and provably efficient shortest path algorithms.” In: *Journal of the ACM* 63.5 (Dec. 2013), 41:1–41:26.

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. “Hierarchical Hub Labelings for Shortest Paths.” In: *Proceedings of the 20th Annual European Symposium on Algorithms (ESA’12)*. Vol. 7501. Lecture Notes in Computer Science. Springer, 2012, pp. 24–35.

[3] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. “Highway Dimension, Shortest Paths, and Provably Efficient Algorithms.” In: *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’10)*. SIAM, 2010, pp. 782–793.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. *Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge*. Vol. 588. American Mathematical Society, 2013.

[6] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. “Fast Routing in Very Large Public Transportation Networks using Transfer Patterns.” In: *Proceedings of the 18th Annual European Symposium on Algorithms (ESA’10)*. Vol. 6346. Lecture Notes in Computer Science. Springer, 2010, pp. 290–301.

[7] 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*. Vol. 9220. Lecture Notes in Computer Science. Springer, 2016, pp. 19–80.

[8] Hannah Bast, Matthias Hertel, and Sabine Storandt. “Scalable Transfer Patterns.” In: *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX’16)*. SIAM, 2016, pp. 15–29.

[9] Hannah Bast, Jonas Sternisko, and Sabine Storandt. “Delay-Robustness of Transfer Patterns in Public Transportation Route Planning.” In: *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’13)*. OpenAccess Series in Informatics (OASIcs). 2013, pp. 42–54.

[10] Hannah Bast and Sabine Storandt. “Flow-Based Guidebook Routing.” In: *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX’14)*. SIAM, 2014, pp. 155–165.

[11] Hannah Bast and Sabine Storandt. “Frequency-Based Search for Public Transit.” In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, Nov. 2014, pp. 13–22.

[12] Gernot Veit Batz. *KaTCH open source code*. Uploaded to github at `https://github.com/GVeitBatz/KaTCH`. 2016.

[13] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. “Minimum Time-Dependent Travel Times with Contraction Hierarchies.” In: *ACM Journal of Experimental Algorithmics* 18.1.4 (Apr. 2013), pp. 1–43.

[14] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. “Preprocessing Speed-Up Techniques is Hard.” In: *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC’10)*. Vol. 6078. Lecture Notes in Computer Science. Springer, 2010, pp. 359–370.

[15] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. “Search-Space Size in Contraction Hierarchies.” In: *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP’13)*. Vol. 7965. Lecture Notes in Computer Science. Springer, 2013, pp. 93–104.

[16] Reinhard Bauer, Gianlorenzo D’Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner. “The Shortcut Problem – Complexity and Algorithms.” In: *Journal of Graph Algorithms and Applications* 16.2 (2012), pp. 447–481.

[17] Reinhard Bauer and Daniel Delling. “SHARC: Fast and Robust Unidirectional Routing.” In: *ACM Journal of Experimental Algorithmics* 14.2.4 (Aug. 2009). Special Section on Selected Papers from ALENEX 2008, pp. 1–29.

[18] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. “Dynamic Time-Dependent Route Planning in Road Networks with User Preferences.” In: *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA’16)*. Vol. 9685. Lecture Notes in Computer Science. Springer, 2016, pp. 33–49.

[19] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. “Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected.” In: *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’09)*. OpenAccess Series in Informatics (OASIcs). 2009.

[20] Annabell Berger, Andreas Gebhardt, Matthias Müller–Hannemann, and Martin Ostrowski. “Stochastic Delay Prediction in Large Train Networks.” In: *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’11)*. Vol. 20. OpenAccess Series in Informatics (OASIcs). 2011, pp. 100–111.

[21] Annabell Berger, Martin Grimmer, and Matthias Müller–Hannemann. “Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks.” In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA’10)*. Vol. 6049. Lecture Notes in Computer Science. Springer, May 2010, pp. 35–46.

[22] Annabell Berger and Matthias Müller–Hannemann. *Subpath-Optimality of Multi-Criteria Shortest Paths in Time- and Event-Dependent Networks*. Tech. rep. University Halle-Wittenberg, Institute of Computer Science, 2009.

[23] M.W Bern, E.L Lawler, and A.L Wong. “Linear-time computation of optimal subgraphs of decomposable graphs.” In: *Journal of Algorithms* 8.2 (1987), pp. 216–235.

[24] Jean Blair and Barry Peyton. “An Introduction to Chordal Graphs and Clique Trees.” In: *Graph Theory and Sparse Matrix Computation*. Vol. 56. The IMA Volumes in Mathematics and its Applications. Springer, 1993, pp. 1–29.

[25] Hans L. Bodlaender. “A Tourist Guide through Treewidth.” In: *Acta Cybernetica* 11 (1993), pp. 1–21.

[26] Hans L. Bodlaender. “Treewidth: Structure and Algorithms.” In: *Proceedings of the 14th International Colloquium on Structural Information and Communication Complexity*. Vol. 4474. Lecture Notes in Computer Science. Springer, 2007, pp. 11–25.

[27] Hans L. Bodlaender, John R. Gilbert, Hjalmtyr Hafsteinsson, and Ton Kloks. “Approximating Treewidth, Pathwidth, Frontsize, and Shortest Elimination Tree.” In: *Journal of Algorithms* 18.2 (Mar. 1995), pp. 238–255.

[28] Hans L. Bodlaender and Arie M. C. A. Koster. “Treewidth computations I. Upper bounds.” In: *Information and Computation* 208.3 (2010), pp. 259–275.

[29] Kateřina Böhmová, Matúš Mihalák, Tobias Pröger, Rastislav Šrámek, and Peter Widmayer. “Robust Routing in Urban Public Transportation: How to Find Reliable Journeys Based on Past Observations.” In: *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. OpenAccess Series in Informatics (OASIcs). 2013, pp. 27–41.

[30] Paul Bonsma. “Most balanced minimum cuts.” In: *Discrete Applied Mathematics* 158.4 (2010), pp. 261–276.

[35] Soma Chaudhuri and Christos Zaroliagis. “Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms.” In: *Algorithmica* 27.3 (Jan. 2000), pp. 212–226.

[36] Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. “Engineering Graph-Based Models for Dynamic Timetable Information Systems.” In: *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*. Vol. 42. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 46–61.

[37] K. Cooke and E. Halsey. “The Shortest Route Through a Network with Time-Dependent Internodal Transit Times.” In: *Journal of Mathematical Analysis and Applications* 14.3 (1966), pp. 493–498.

[38] Bruno Courcelle. “The monadic second-order logic of graphs. I. Recognizable sets of finite graphs.” In: *Information and Computation* 85.1 (Mar. 1990), pp. 12–75.

[39] Holger Dell, Thore Husfeldt, Bart M. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances Rosamond. “The First Parameterized Algorithms and Computational Experiments Challenge.” In: *11th International Symposium on Parameterized and Exact Computation*. Leibniz International Proceedings in Informatics. 2016, 30:1–30:9.

[40] Daniel Delling. “Time-Dependent SHARC-Routing.” In: *Algorithmica* 60.1 (May 2011), pp. 60–94.

[41] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. “Computing Multimodal Journeys in Practice.” In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 260–271.

[42] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. "Public Transit Labeling." In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Lecture Notes in Computer Science. Springer, 2015, pp. 273–285.

[43] Daniel Delling, Daniel Fleischer, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. "An exact combinatorial algorithm for minimum graph bisection." In: *Mathematical Programming* 153.2 (Nov. 2015), pp. 417–458.

[44] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Customizable Route Planning." In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, pp. 376–387.

[45] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Customizable Route Planning in Road Networks." In: *Transportation Science* (May 2015).

[46] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. "Robust Distance Queries on Massive Networks." In: *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Vol. 8737. Lecture Notes in Computer Science. Springer, Sept. 2014, pp. 321–333.

[47] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. "Exact Combinatorial Branch-and-Bound for Graph Bisection." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 30–44.

[48] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. "Graph Partitioning with Natural Cuts." In: *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011, pp. 1135–1146.

[49] Daniel Delling, Bastian Katz, and Thomas Pajor. "Parallel Computation of Best Connections in Public Transportation Networks." In: *ACM Journal of Experimental Algorithmics* 17.4 (July 2012), pp. 4.1–4.26.

[50] Daniel Delling and Giacomo Nannicini. "Core Routing on Dynamic Time-Dependent Road Networks." In: *Informs Journal on Computing* 24.2 (2012), pp. 187–201.

[51] Daniel Delling, Thomas Pajor, and Renato F. Werneck. "Round-Based Public Transit Routing." In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012, pp. 130–140.

[52] Daniel Delling, Thomas Pajor, and Renato F. Werneck. "Round-Based Public Transit Routing." In: *Transportation Science* 49.3 (2015), pp. 591–604.

[53] Daniel Delling and Dorothea Wagner. “Time-Dependent Route Planning.” In: *Robust and Online Large-Scale Optimization*. Vol. 5868. Lecture Notes in Computer Science. Springer, 2009, pp. 207–230.

[54] Daniel Delling and Renato F. Werneck. “Faster Customization of Road Networks.” In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA’13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 30–42.

[55] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, eds. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Vol. 74. DIMACS Book. American Mathematical Society, 2009.

[56] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. “A case for time-dependent shortest path computation in spatial networks.” In: *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS’10)*. 2010, pp. 474–477.

[57] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. *Connection Scan Algorithm*. Tech. rep. ArXiv e-prints, 2017.

[58] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. “Intriguingly Simple and Fast Transit Routing.” In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA’13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 43–54.

[59] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Customizable Contraction Hierarchies*. Tech. rep. ArXiv e-prints, 2014.

[60] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Customizable Contraction Hierarchies.” In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA’14)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 271–282.

[61] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Customizable Contraction Hierarchies.” In: *ACM Journal of Experimental Algorithmics* 21.1 (Apr. 2016), 1.5:1–1.5:49.

[62] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Delay-Robust Journeys in Timetable Networks with Minimum Expected Arrival Time.” In: *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’14)*. Vol. 42. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 1–14.

[65] Edsger W. Dijkstra. “A Note on Two Problems in Connexion with Graphs.” In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[66] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. "Multi-Criteria Shortest Paths in Time-Dependent Train Networks." In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Vol. 5038. Lecture Notes in Computer Science. Springer, June 2008, pp. 347–361.

[67] Stuart E. Dreyfus. "An Appraisal of Some Shortest-Path Algorithms." In: *Operations Research* 17.3 (1969), pp. 395–412.

[68] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. "Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools." In: *Graph Drawing Software*. Springer, 2003, pp. 127–148.

[69] Donatella Firmani, Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. "Is Timetabling Routing Always Reliable for Public Transport?" In: *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. OpenAccess Series in Informatics (OASIcs). 2013, pp. 15–26.

[70] Lester R. Ford Jr. and Delbert R. Fulkerson. "Maximal flow through a network." In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[71] Forschungsgesellschaft für Verkehrswesen. *Richtlinien für Lichtsignalanlagen (RiLSA)*. 2010.

[72] Luca Foschini, John Hershberger, and Subhash Suri. "On the Complexity of Time-Dependent Shortest Paths." In: *Algorithmica* 68.4 (Apr. 2014), pp. 1075–1097.

[73] Delbert R. Fulkerson and O. A. Gross. "Incidence Matrices and Interval Graphs." In: *Pacific Journal of Mathematics* 15.3 (1965), pp. 835–855.

[74] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. "Some Simplified $\mathcal{NP}$-Complete Graph Problems." In: *Theoretical Computer Science* 1 (1976), pp. 237–267.

[75] Robert Geisberger. "Contraction of Timetable Networks with Realistic Transfers." In: *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Vol. 6049. Lecture Notes in Computer Science. Springer, May 2010, pp. 71–82.

[76] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks." In: *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*. Vol. 5038. Lecture Notes in Computer Science. Springer, June 2008, pp. 319–333.

[77] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. "Exact Routing in Large Road Networks Using Contraction Hierarchies." In: *Transportation Science* 46.3 (Aug. 2012), pp. 388–404.

[78] Alan George. "Nested Dissection of a Regular Finite Element Mesh." In: *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[79] Alan George and Joseph W. Liu. "A Quotient Graph Model for Symmetric Factorization." In: *Sparse Matrix Proceedings*. SIAM, 1978, pp. 154–175.

[80] Alan George and Joseph W. Liu. "The Evolution of the Minimum Degree Ordering Algorithm." In: *SIAM Review* 31.1 (1989), pp. 1–19.

[81] John R. Gilbert and Robert Tarjan. "The analysis of a nested dissection algorithm." In: *Numerische Mathematik* 50.4 (July 1986), pp. 377–404.

[82] Marc Goerigk, Sascha Heße, Matthias Müller–Hannemann, and Marie Schmidt. "Recoverable Robust Timetable Information." In: *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. OpenAccess Series in Informatics (OASIcs). 2013, pp. 1–14.

[83] Marc Goerigk, Martin Knoth, Matthias Müller–Hannemann, Marie Schmidt, and Anita Schöbel. "The Price of Robustness in Timetable Information." In: *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*. Vol. 20. OpenAccess Series in Informatics (OASIcs). 2011, pp. 76–87.

[84] Andrew V. Goldberg and Chris Harrelson. "Computing the Shortest Path: A* Search Meets Graph Theory." In: *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*. SIAM, 2005, pp. 156–165.

[85] Michael Hamann and Ben Strasser. *Graph Bisection with Pareto-Optimization*. Tech. rep. ArXiv e-prints, 2015.

[86] Michael Hamann and Ben Strasser. "Graph Bisection with Pareto-Optimization." In: *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX'16)*. SIAM, 2016, pp. 90–102.

[87] Martin Holzer, Frank Schulz, and Dorothea Wagner. "Engineering Multi-Level Overlay Graphs for Shortest-Path Queries." In: *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*. SIAM, 2006, pp. 156–170.

[88] Martin Holzer, Frank Schulz, and Dorothea Wagner. "Engineering Multilevel Overlay Graphs for Shortest-Path Queries." In: *ACM Journal of Experimental Algorithmics* 13.2.5 (Dec. 2008), pp. 1–26.

[89] John E. Hopcroft and Robert E. Tarjan. "Efficient Algorithms for Graph Manipulation." In: *Communications of the ACM* 16.6 (June 1973), pp. 372–378.

[90] Cecil Huang and Adnan Darwiche. “Inference in belief networks: A procedural guide.” In: *International Journal of Approximate Reasoning* 15.3 (Oct. 1996), pp. 225–263.

[91] Ananth V. Iyer, H. Donald Ratliff, and Gopalakrishnan Vijayan. “Optimal node ranking of trees.” In: *Information Processing Letters* 28.5 (Aug. 1988), pp. 225–229.

[92] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.” In: *SIAM Journal on Scientific Computing* 20.1 (1999), pp. 359–392.

[94] Donald E. Knuth. *The Art Of Computer Programming, Sorting and Searching*. Vol. 3. Addison-Wesley, 1998.

[95] Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. “Engineering Oracles for Time-Dependent Road Networks.” In: *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX’16)*. SIAM, 2016.

[96] Adrian Kosowski and Laurent Viennot. “Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons.” In: *Proceedings of the 28th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA’17)*. SIAM, 2017, pp. 1462–1478.

[97] Ulrich Lauther. “An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background.” In: *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*. Vol. 22. IfGI prints, 2004, pp. 219–230.

[98] Frank Thomson Leighton and Satish Rao. “Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms.” In: *Journal of the ACM* 46.6 (1999), pp. 787–832.

[99] Richard J. Lipton, Donald J. Rose, and Robert Tarjan. “Generalized Nested Dissection.” In: *SIAM Journal on Numerical Analysis* 16.2 (Apr. 1979), pp. 346–358.

[100] Transport for London. *London data store*. `http://data.london.gov.uk`. 2011.

[101] Catherine C. McGeoch, Peter Sanders, Rudolf Fleischer, Paul R. Cohen, and Doina Precup. “Using Finite Experiments to Study Asymptotic Performance.” In: *Experimental Algorithmics – From Algorithm Design to Robust and Efficient Software*. Vol. 2547. Lecture Notes in Computer Science. Springer, 2002, pp. 93–126.

[102] Matthias Müller–Hannemann and Mathias Schnee. "Efficient Timetable Information in the Presence of Delays." In: *Robust and Online Large-Scale Optimization*. Vol. 5868. Lecture Notes in Computer Science. Springer, 2009, pp. 249–272.

[103] Matthias Müller–Hannemann and Mathias Schnee. "Finding All Attractive Train Connections by Multi-Criteria Pareto Search." In: *Algorithmic Methods for Railway Optimization*. Vol. 4359. Lecture Notes in Computer Science. Springer, 2007, pp. 246–263.

[104] Matthias Müller–Hannemann and Mathias Schnee. "Paying Less for Train Connections with MOTIS." In: *Proceedings of the 5th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'05)*. OpenAccess Series in Informatics (OASIcs). 2006.

[105] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. "Timetable Information: Models and Algorithms." In: *Algorithmic Methods for Railway Optimization*. Vol. 4359. Lecture Notes in Computer Science. Springer, 2007, pp. 67–90.

[106] Matthias Müller–Hannemann and Karsten Weihe. "Pareto Shortest Paths is Often Feasible in Practice." In: *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*. Vol. 2141. Lecture Notes in Computer Science. Springer, 2001, pp. 185–197.

[107] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. "Bidirectional A* Search on Time-Dependent Road Networks." In: *Networks* 59 (2012). Best Paper Award, pp. 240–251.

[108] Ariel Orda and Raphael Rom. "Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length." In: *Journal of the ACM* 37.3 (1990), pp. 607–625.

[109] Léon Planken, Mathijs de Weerdt, and Roman van Krogt. "Computing All-pairs Shortest Paths by Leveraging Low Treewidth." In: *Journal of Artificial Intelligence Research* 43 (2012), pp. 353–388.

[110] Alex Pothen. *The complexity of optimal elimination trees*. Tech. rep. Pennsylvania State University, 1988.

[111] PTV AG – Planung Transport Verkehr. *http://www.ptv.de*. 1979.

[112] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. "Efficient Models for Timetable Information in Public Transportation Systems." In: *ACM Journal of Experimental Algorithmics* 12.2.4 (2008), pp. 1–39.

[113] Michael Rice and Vassilis Tsotras. "Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions." In: *Proceedings of the VLDB Endowment* 4.2 (Nov. 2010), pp. 69–80.

[114] Peter Sanders. “Algorithm Engineering – An Attempt at a Definition.” In: *Efficient Algorithms*. Vol. 5760. Lecture Notes in Computer Science. Springer, 2009, pp. 321–340.

[115] Peter Sanders and Dominik Schultes. “Engineering Highway Hierarchies.” In: *ACM Journal of Experimental Algorithmics* 17.1 (2012), pp. 1–40.

[116] Peter Sanders and Christian Schulz. “Advanced Multilevel Node Separator Algorithms.” In: *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA’16)*. Vol. 9685. Lecture Notes in Computer Science. Springer, 2016, pp. 294–309.

[117] Peter Sanders and Christian Schulz. “Think Locally, Act Globally: Highly Balanced Graph Partitioning.” In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA’13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 164–175.

[118] Peter Sanders and Dorothea Wagner. “Algorithm Engineering.” In: *Informatik Spektrum* 36.2 (Apr. 2013), pp. 187–190.

[119] Alejandro A. Schæffer. “Optimal node ranking of trees in linear time.” In: *Information Processing Letters* 33 (Nov. 1989), pp. 91–96.

[120] Aaron Schild and Christian Sommer. “On Balanced Separators in Road Networks.” In: *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA’15)*. Lecture Notes in Computer Science. Springer, 2015, pp. 286–297.

[121] Frank Schulz, Dorothea Wagner, and Karsten Weihe. “Dijkstra’s Algorithm On-Line: An Empirical Case Study from Public Railroad Transport.” In: *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE’99)*. Vol. 1668. Lecture Notes in Computer Science. Springer, 1999, pp. 110–123.

[122] Frank Schulz, Dorothea Wagner, and Karsten Weihe. “Dijkstra’s Algorithm On-Line: An Empirical Case Study from Public Railroad Transport.” In: *ACM Journal of Experimental Algorithmics* 5.12 (2000), pp. 1–23.

[123] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. “Using Multi-Level Graphs for Timetable Information in Railway Systems.” In: *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX’02)*. Vol. 2409. Lecture Notes in Computer Science. Springer, 2002, pp. 43–59.

[124] A. J. Soper, Chris Walshaw, and Mark Cross. “A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning.” In: *Journal of Global Optimization* 29.2 (2004), pp. 225–241.

[125] Sabine Storandt. “Contraction Hierarchies on Grid Graphs.” In: *Proceedings of the 36rd Annual German Conference on Advances in Artificial Intelligence*. Lecture Notes in Computer Science. Springer, 2013, pp. 236–247.

[126] Ben Strasser. “Delay-Robust Stochastic Routing in Timetable Networks.” Diploma Thesis. Karlsruhe Institute of Technology, July 2012.

[127] Ben Strasser. *Intriguingly Simple and Efficient Time-Dependent Routing in Road Networks.* Tech. rep. ArXiv e-prints, 2016.

[128] Ben Strasser. *Source code of PACE 2016 FlowCutter submission.* Uploaded to github at `https://github.com/ben-strasser/flow-cutter-pace16`. 2016.

[129] Ben Strasser. *Source code of RoutingKit.* Uploaded to github at `https://github.com/RoutingKit/RoutingKit`. 2016.

[130] Ben Strasser. *TD-S experimental open source code.* Uploaded to github at `https://github.com/ben-strasser/td_p`. 2016.

[133] Ben Strasser and Dorothea Wagner. “Connection Scan Accelerated.” In: *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX’14).* SIAM, 2014, pp. 125–137.

[134] Ben Strasser and Dorothea Wagner. “Graph Fill-In, Elimination Ordering, Nested Dissection and Contraction Hierarchies.” In: *Gems of Combinatorial Optimization and Graph Algorithms.* Springer, Dec. 2015, pp. 69–82.

[135] Nathan Sturtevant. “Benchmarks for Grid-Based Pathfinding.” In: *Transactions on Computational Intelligence and AI in Games* 4.2 (May 2012), pp. 144–148.

[136] Nathan Sturtevant. *Pathfinding Benchmarks.* `http://www.movingai.com/benchmarks/`. 2014.

[138] William F. Tinney and J.W. Walker. “Direct solutions of sparse network equations by optimally ordered triangular factorization.” In: *Proceedings of the IEEE* 55.11 (Nov. 1967), pp. 1801–1809.

[139] Dorothea Wagner and Frank Wagner. “Between Min Cut and Graph Bisection.” In: *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (MFCS’93).* Vol. 711. Lecture Notes in Computer Science. London, UK: Springer, 1993, pp. 744–750.

[140] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. “Efficient Route Planning on Public Transportation Networks: A Labelling Approach.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD’15).* ACM Press, 2015, pp. 967–982.

[141] Michael Wegner. “Finding Small Node Separators.” Bachelor Thesis. Karlsruhe Institute of Technology, Oct. 2014.

[142] Fang Wei. “TEDI: efficient shortest path query answering on graphs.” In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD’10).* ACM Press, 2010.

[143] Jörg D. Weisbarth. “Shortest-Path Cover auf eingeschränkten Graphklassen.” Bachelor thesis. Karlsruhe Institute of Technology, May 2012.

[144] Sascha Witt. “Trip-Based Public Transit Routing.” In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA’15)*. Lecture Notes in Computer Science. Accepted for publication. Springer, 2015, pp. 1025–1036.

[145] Sascha Witt. “Trip-Based Public Transit Routing Using Condensed Search Trees.” In: *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’16)*. Vol. 54. OpenAccess Series in Informatics (OASIcs). Aug. 2016, 10:1–10:12.

[146] Tim Zeitz. “Weak Contraction Hierarchies Work!” Bachelor Thesis. Karlsruhe Institute of Technology, 2013.

List of Coauthored Publications

Adi Botea, Ben Strasser, and Daniel Harabor. "Complexity Results for Compressing Optimal Paths." In: *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. AAAI Press, 2015, pp. 1100–1106.

Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. "Fast Quasi-Threshold Editing." In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*. Lecture Notes in Computer Science. Springer, 2015, pp. 251–262.

Lars Briem, H. Sebastian Buck, Holger Ebhart, Nicolai Mallig, Ben Strasser, Peter Vortisch, Dorothea Wagner, and Tobias Zündorf. "Efficient Traffic Assignment for Public Transit Networks." In: *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Lecture Notes in Computer Science. Springer, 2017.

Lars Briem, H. Sebastian Buck, Nicolai Mallig, Peter Vortisch, Ben Strasser, Dorothea Wagner, and Tobias Zündorf. "Modelling public transport in mobiTopp." In: *The 6th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications*. Elsevier B.V., 2017.

Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. *Connection Scan Algorithm*. Tech. rep. ArXiv e-prints, 2017.

Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. "Intriguingly Simple and Fast Transit Routing." In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Vol. 7933. Lecture Notes in Computer Science. Springer, 2013, pp. 43–54.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Customizable Contraction Hierarchies*. Tech. rep. ArXiv e-prints, 2014.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies." In: *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Vol. 8504. Lecture Notes in Computer Science. Springer, 2014, pp. 271–282.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner. "Customizable Contraction Hierarchies." In: *ACM Journal of Experimental Algorithmics* 21.1 (Apr. 2016), 1.5:1–1.5:49.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Delay-Robust Journeys in Timetable Networks with Minimum Expected Arrival Time.” In: *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS’14)*. Vol. 42. OpenAccess Series in Informatics (OASIcs). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2014, pp. 1–14.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner. “Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs.” In: *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2015, 66:1–66:4.

Julian Dibbelt, Ben Strasser, and Dorothea Wagner. *Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs*. Tech. rep. abs/1509.03165. ArXiv e-prints, 2015.

Michael Hamann and Ben Strasser. *Graph Bisection with Pareto-Optimization*. Tech. rep. ArXiv e-prints, 2015.

Michael Hamann and Ben Strasser. “Graph Bisection with Pareto-Optimization.” In: *Proceedings of the 18th Meeting on Algorithm Engineering and Experiments (ALENEX’16)*. SIAM, 2016, pp. 90–102.

Bastian Katz, Ignaz Rutter, Ben Strasser, and Dorothea Wagner. “Speed Dating: An Algorithmic Case Study Involving Matching and Scheduling.” In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA’11)*. Vol. 6630. Lecture Notes in Computer Science. Springer, 2011, pp. 292–303.

Ben Strasser. *Intriguingly Simple and Efficient Time-Dependent Routing in Road Networks*. Tech. rep. ArXiv e-prints, 2016.

Ben Strasser, Adi Botea, and Daniel Harabor. “Compressing Optimal Paths with Run Length Encoding.” In: *Journal of Artificial Intelligence Research* 54 (2015), pp. 593–629.

Ben Strasser, Daniel Harabor, and Adi Botea. “Fast First-Move Queries through Run-Length Encoding.” In: *Proceedings of the 5th International Symposium on Combinatorial Search (SoCS’14)*. AAAI Press, July 2014, pp. 157–165.

Ben Strasser and Dorothea Wagner. “Connection Scan Accelerated.” In: *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX’14)*. SIAM, 2014, pp. 125–137.

Ben Strasser and Dorothea Wagner. “Graph Fill-In, Elimination Ordering, Nested Dissection and Contraction Hierarchies.” In: *Gems of Combinatorial Optimization and Graph Algorithms*. Springer, Dec. 2015, pp. 69–82.

Nathan Sturtevant, Jason Traish, James 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 6th International Symposium on Combinatorial Search (SoCS’15)*. AAAI Press, June 2015.