Route Planning with Flexible Objective Functions

Moritz Kobitzsch[1]

Diploma Thesis

At the Department of Computer Science
Karlsruhe Institute of Technology

Referee: Prof. Dr. Peter Sanders

Advisor: Robert Geisberger

16th November 2009

Karlsruhe Institute of Technology

[1]Karlsruhe Institute of Technology, kobitzsch@kit.edu

Contents

Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenhändig angefertigt habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Karlsruhe, 16th November 2009

(Moritz Kobitzsch)

I would like to thank Veit Batz for his support when i started my work on this thesis. Also, I would like to thank Daniel Delling for his implementation of the Landmark selection methods. Finally i would like to thank my mother and my father for the support during my years at the university and for the lecturing work.

Chapter 1

German Summary

In der heutigen Zeit ist die Welt von der Globalisierung geprägt. Personen und Güter bewegen sich jeden Tag in unglaublichen Mengen von einem Ort zum anderem. Dabei kann für jede mögliche Anfrage eine eigenständige Anforderung an die Art der gewählten Strecke bestehen. Flexibilität bei Routenfindungssystemem wird deswegen immer wichtiger. Bisherige Arbeiten auf dem Gebiet befassten sich meist mit einem einzelnem zu optimierendem Kriterium. Einen ausfürlichen Überblick gibt [DSSW09]. In unserer Sicht der Dinge wird dieses in Zukunft nicht mehr ausreichen. Statt nur eine einzelne mögliche Route anzubieten könnte man sich einen Schieberegler für einen Webserver oder ein Navigationsgerät vorstellen. Bei diesem kann der Nutzer dann selbst entscheiden, wie viel extra Zeit er aufbringen möchte um mögliche Kosten einzusparen. Oder der Server könnte in Anlehnung an die Präsentation von alternativen Zugverbindungen eine Folge von im Parameter nah beieinander liegenden Routen anbieten. Der Nutzer könnte sich dann für eine dieser Routen entscheiden oder weitere Variation des Parameters anfragen.

Um solche Anforderungen zu erfüllen präsentieren wir einen Algorithmus für das parametrisierte kürzeste Wege Problem. In diesem Problem weisen wir jeder Kante eines Graphen eine lineare Kostenfunktion von der Form $w(p) = \tau + p \cdot \kappa$ mit zwei Gewichtstermen τ, κ und einem Parameter p zu. Das Ziel ist es dann den kürzesten Weg bezüglich eines gewählten Parameters p zu finden. Dieses Problem wurde bisher recht wenig betrachtet. Für unseren Algorithmus wählen wir dabei τ und κ aus $\mathbb{N}$. Bei p beschränken wir uns dabei auf ein Intervall $[0, p_{max}]$ von natürlichen Zahlen. Diese Einschränkung ermöglicht eine schnellere Vorberechnung, da insgesamt die Anzahl möglicher alternativen eingeschränkt wird. p_{max} kann dabei gewählt werden um den maximalen Einfluss des Gewichtes κ zu bestimmen.

Zum Beispiel kann mit Hilfe von Dijkstra's Algorithmus ein entsprechender kürzester Weg gefunden werden, indem man die Kostenfunktion wenn nötig auswertet. Aktuellere Methoden im Bereich der multikriteriellen kürzeste Wege Suche beschäftigen sich meist mit dem Ansatz der Pareto-optimalität. Dabei sucht der Algorithmus eine Menge von Pfaden, die nicht von einem anderen Pfad dominiert werden. Die Kantengewichte werden dabei durch Vektoren von Gewichten in der Form $(w_1, \dots w_k)$ beschrieben. Wir bezeichnen dann einen Vektor $W' = (w'_1, \dots, w'_k)$ als von $W = (w_1, \dots, w_k)$ dominiert, wenn $\forall i \in [1, k] : w_i \leq w'_i$ and $\exists i \in [1, k] : w_i < w'_i$.

Nach unserer Ansicht bestehen bei diesem Ansatz einige Probleme. Die meisten Algorithmen leiden dabei unter dem Problem, dass die Anzahl der möglichen Pareto-optimalen Pfade zu stark ansteigt um eine effiziente Vorberechnung zu erlauben. Deswegen muss meist die Anzahl der Pfade mit Hilfe von Heuristiken oder verschärften Dominanzkriterien eingeschränkt werden. Wir beschäftigen uns in dieser Arbeit mit den linearen Kantenfunktionen. In Kombination mit aktuellen hierarchischen Techniken gelingt es uns dabei für Straßennetze in kontinentaler Größe schnelle Anfragen zu ermöglichen.

Wir adaptieren für unsere Zwecke die von Geisberger et al. vorgestellten Contraction Hierarchies [GSSD08]. Dieser Algorithmus weist den Knoten eines Graphen mit Hilfe einer Heuristik eine Wichtigkeit zu. Diese Heuristik hat dabei zwar Einfluss auf die damit erreichbaren Anfragezeiten, der Algorithmus berechnet aber für jede mögliche Ordnung korrekte kürzeste Wege. Die Heuristik bedient sich dabei mehrerer Eigenschaften der Knoten und des Graphens, die sich aus der bisherigen Bearbeitung und der Struktur des Eingabegraphen ergeben. Eine mögliche Eigenschaft ist dabei zum Beispiel die Anzahl der schon kontrahierten direkten Nachbarn. Ein Knoten wird dabei als wichtig betrachtet, wenn er viele schon kontrahierte Nachbarn besitzt. Viele dieser Eigenschaften können sich dabei während der Vorberechnung verändern. Mit Hilfe der Heuristik wählt der Algorithmus dabei immer einen nächsten Knoten aus um diesen zu kontrahieren. Dabei suchen wir uns immer den zur Zeit am unwichtigsten erscheinenden Knoten. Bei der Kontraktion wird der Knoten aus dem Graphen entfernt und wenn notwendig zusätzliche Kanten eingefügt, um einen Overlay Graphen zu erhalten. Diese Kanten, die wir Shortcuts nennen, werden immer dann eingefügt, wenn ansonsten kürzeste Wege Distanzen zwischen im Graphen verbleibenden Knoten verändert werden würden. Für eine Suche wird der so entstandende Graph dann in einen Vorwärts-Graphen und einen Rückwärts-Graphen geteilt. Auf diesen beiden Graphen wird dann eine simultane Suche vom Start- und vom Zielpunkt aus gestartet. Diese Suchen treffen sich während ihres Verlaufes garantiert an einer höheren Stelle in der Hierarchie.

Für unseren Algorithmus müssen wir dabei besonders die Vorberechnung anpassen. Während der Vorberechnungsphase müssen bei der Kontraktion eines Knotens sogenannte Zeugen gefunden werden. Ein Zeuge ist dabei eine alternative Route zwischen zwei Nachbarknoten, die höchstens die gleiche Reisezeit benötig wie die über den zu kontrahierenden Knoten. Wird ein solcher Zeuge gefunden, müssen wir keine Kante in den Overlay-Graphen einfügen. Ist eine solche Kante allerdings nicht vorhanden, benötigen wir einen sogenannten Shortcut, der den Weg über den zu kontrahierenden Knoten repräsentiert. Für den flexibelen Fall ist die Situation nicht mehr so einfach. In diesem Fall können wir auch teilweise Zeugen für einen Pfad haben. Diese können dabei, auf Grund von Konkavität der minimal möglichen Kosten über alle Parameter und Pfade, im unteren Parameterbereich oder im oberen Parameterbereich als Zeuge fungieren. Das ermöglicht es uns mögliche Shortcuts auf Notwendigkeitsintervalle einzuschränken.

Da wir beweisen können, dass maximal zwei Pfade benötigt werden um einen gültigen Zeugen darzustellen für das vollständige Parameterintervall, benutzen wir eine einfache Erweiterung des Dijkstra Algorithmus für unsere Zeugensuche. Wir erwarten dabei, dass wir auch nur wenige Aufrufe unserer Erweiterung durchführen müssen um diese zwei Pfade zu finden. Der Algorithmus sucht dabei auf einem zu betrachtenden Intervall $[L, U]$ zuerst für den Parameter L den kürzesten Weg. Danach bestimmen wir den maximalen Parameterwert L', für den der gefundene Pfad ein Zeuge ist, und starten die Suche mit dem Parameterwert $L' + 1$ neu. Dieses wird so lange wiederholt bis der Algorithmus entweder bis zum Parameterwert U einen Zeugen gefunden hat, oder kein größerer Parameterwert mehr bestimmt werden kann. Im letzteren Fall beschränken wir das Notwendigkeitsintervall durch eine identische Suche, die beim Parameter U beginnt und den kleinstmöglichen Parameter sucht. Dabei konnten wir in der Praxis einen Durchschnitt von nur zwei Aufrufen des Dijkstra Algorithmus feststellen. Da wir für diesen Algorithmus einen nächsten Parameter bestimmen müssen, wählen wir als Parameter nur natürliche Zahlen.

Weil die Leistung und die Vorberechnung einer CH allerdings sehr stark von der gewählten Ordnung der Knoten abhängt, mussten wir im Verlauf der Arbeit feststellen, dass eine einzelne Knotenordnung zu zu langen Vorberechnungszeiten führte. Deswegen entwickelten wir einen Teile-und-herrsche Ansatz, der es uns ermöglicht auch große Graphen möglichst effizient zu berechnen. Dabei versuchen wir der Anteil mehrfach erledigter Arbeit möglichst gering zu halten.

Weiterhin nutzen wir für unseren Algorithmus Techniken, die auf dem Kern der berechneten Hierarchie genutzt werden. Der Kern der Hierarchie ist dabei die Menge der $|\text{Kern}|$ Knoten, die nach unserer Ordnung die höchsten Zahlen zugewiesen bekommen. Diese Techniken nutzen wir dabei sowohl auf einem kontrahierten als auch einem nicht kontrahiertem Kern. Dazu benutzen wir die Kombination des A*-Algorithmus mit Landmarken und der Dreiecksungleichung, oder auch ALT genannt. Diese Technik nutzt zentrale Punkte im Graphen um untere Schranken für die kürzesten Wege mit Hilfe der Dreiecksungleichung abzuschätzen. Mit Hilfe dieser Abschätzung kann die Suche mit einem Richtungssinn kombiniert werden.

In der Kombination gelingt es uns damit für Straßennetzwerke bis zu der Größe von Europa eine effiziente Vorberechnung durchzuführen. Wir schaffen es dabei Anfragen nach kürzesten Wegen für einen beliebig wählbaren Parameter $p \in [0, p_{max}]$ innerhalb von unter zwei Millisekunden zu beantworten. Für kleinere Straßennetze in der Größe von Deutschland schaffen wir das sogar innerhalb von mehreren hundert Microsekunden. Damit erreichen wir für Europa eine Geschwindigkeitssteigerung mit einem Faktor von über $8\,000$. Unsere Vorberechnungszeiten sind dabei für Graphen wie Deutschland im Bereich von zwei bis drei Stunden und gehen für Graphen in der Größe von Europa auf bis zu 24 Stunden.

Einer der größten Unterschiede unseres Algorithmus zu allen anderen bisher präsentierten Algorithmen, die uns bekannt sind, ist, dass wir jeweils nur einen Weg für einen festen Parameter berechnen. Allerdings gehen wir davon aus, dass wir innerhalb von linearer Zeit, abhängig von der Zahl möglicher Alternativrouten, alle möglichen Wege bestimmen können.

Damit gelingt es uns insgesamt einen Algorithmus vorzustellen, der es schafft in einem Server Szenario eine Folge von Anfragen zu beantworten, in der der Nutzer die Art der gewählten Route nach seinen Bedürfnissen beinflussen kann. Dabei könnte zum Beispiel eine Vorauswahl von möglichen Routen präsentiert werden und der Nutzer kann sich wiederum weitere Routen anfragen, die im Parameterbereich einer der präsentierten Routen liegen. Unserer Ansicht nach ist diese Art der Flexibilität sehr gut geeignet um kürzeste Wege Anfragen auf großen Straßennetzen variabel zu gestalten.

Chapter 2

Introduction

In a world that is shaped by globalization, transportation is becoming more important each day. People and goods need to travel from one location to another each day in quantities beyond imagination. Route planning devices have become a big part of our life. Nearly every cellphone supports us in finding our way with virtually no effort. But those devices are insufficient. They usually still rely on heuristics to perform their tasks and to compensate for their lack of computational power and memory. While this may be sufficient for a single person on the way to work, it is easy to find examples where heuristics are not enough. For example, in logistics, finding optimal routes can save a lot of money, since the small savings in gas add up very fast. Another benefit of the optimal routes is the decrease in CO_2 emissions that could be achieved by following optimal routes. Note that we do not solve the ressource constrained shortest path problem. Our Algorithm can only be used to find routes with lower costs if we do not have deadlines to meet. Therefore, the most prominent example would be a webserver that presents the user a number of possible routes to choose from. This could allow to present routes similar to alternative train connections. Someone could also imagine a car navigation system that provides a slider to adjust the calculated route as necessary.

In 1959 Edsgar W. Dijkstra introduced an algorithm to the problem of finding shortest paths in a static graph [Dij59]. A lot of speed-up techniques known today are based on Dijkstra's algorithm. To provide fast queries on large graphs, one can take advantage of the fact that road networks do not change a lot over time. Therefore, many techniques perform time consuming preprocessing routines to exploit the precalculated information during the queries. If we use the fastest of these algorithms, we can gain speedups of up to one million at the cost to preprocessing time and space overhead.

Most times algorithms focus only on one single criterion to calculate the shortest routes, e.g., a combination of the traversal time and estimated traversal costs at a fixed ratio. But there are a lot of possible terms we can choose from. Examples include the fuel costs, toll fees as on the German Autobahn or many more. This has been sufficient for a long time. Today we have very diverse needs on the different routes. A familiy traveling abroad may want the fastest route to keep the traveling time short. A student may want to save as much money as possible and therefore choose a cheaper road.

The route that leads to the minimal cost does not have to be the route with the fastest time. To avoid the toll fee, it might pay off to take the slower country road instead of the freeway. Even though this increases the traveling time, it may result in lower costs for the respective company. We observe that it is possible to modulate a set of different weight terms in a term we call the cost of a road. We can combine the work payments of a company driver, the fuel costs, toll payments and many more. With this observation we present an extension to the known technique of Contraction Hierarchies presented by Geisberger et al. [GSSD08]. This thesis will present a way to modify the Contraction Hierarchy (CH) algorithm to provide fast flexible queries that allow us to choose a combination of a set of weight terms at runtime.

2.1 Related Work

In the past, there has been extensive work on single-criteria speed-up techniques. These methods require to select the objective function prior to the preprocessing. We refer to [DSSW09] for an overview. In general, these objective functions can be arbitrary, but many of them achieve their best performance with the travel time metric. In this thesis, we will use flexible objective functions during our preprocessing. The flexible objective functions can be modulated by the use of linear functions depending on a parameter p. We could get to the same functionality as our algorithm if we perform any single-criteria speedup method for every parameter p and select the correct preprocessed data at query time. Obviously this will lead to a huge amount of preprocessing and data overhead.

In general we can classify the single-criteria algorithms into three categories: *hierarchical* approaches, *goal-directed* methods and *combinations* of both algorithms. Our algorithm is closely related to the Contraction Hierarchy method, which we use as a base for our algorithm. We further utilize techniques from goal directed approaches, i.e., the ALT-algorithm. In the following we will describe some of the methods of single-criteria algorithms.

2.1.1 Goal Directed Approaches

Firstly, we want to shortly describe two goal directed approaches; the ALT algorithm and the ArcFlag algorithm.

ALT

Goldberg and Harrelson presented the ALT algorithm [GH05]. It is an extension to the classic Dijkstra's algorithm. Goldberg and Harrelson augment the search with a sense for direction. In a preprocessing step, their algorithm calculates a potential function, which they add to the graph. For this, they select distinct points in the graph, which they call landmarks. In combination with the triangle inequality the algorithm can then approximate the length of a path between a source and a target. To efficiently use the triangle inequality we have to compute the distances from all nodes to the landmarks and vice versa in advance. This methods therefore comes with a large space overhead. Still, this method proves quite useful if we do not have to use it on the entire graph, but on the core of a hierarchical approach. We will utilize this algorithm in this thesis. Therefore, we provide a more detailed description in Section 3.

ArcFlags

Another goal-directed approach is the Arc Flags algorithm Lauther introduced [Lau97]. The algorithm is based on a partition of the original graph into a distinct set of regions. Lauther augments the original edges with a set of flags. Theses flags indicate for each region wether a shortest path exists, leading into the region that uses the given edge. During a search, we can use this flags to prune the search on edges that do not lead into the region of the target. To calculate the flags, shortest paths have to be precomputed starting at all border nodes of a given region. In the past some experiments showed that the combination of the ArcFlag algorithm with the ALT method can provide even better results in terms of query times.

2.1.2 Hierarchical Approaches

Hierarchical methods try to exploid some graph induced node order. For the hierarchical approaches we will focus on the Contraction Hierarchies method, since it is closely related to our algorithm. For a more detailed discussion we refer to [Sch08b].

Highway Hierarchies

This algorithm, introduced by Schultes et al. groups nodes and edges into a hierarchy of levels by alternating between two procedures [SS05]. In a contraction step, the network is contracted by removing low-degree nodes. In this *node reduction* phase, the algorithm bypasses unimportant nodes and introduces shortcut edges to preserve shortest path distances. In particular, this step removes all nodes of degree one and two. *Edge reduction* then removes *non-highway edges*, i.e., edges that only form a shortest path in a local area around a source or a target. For a more detailed description of the locality principle we refer to [Sch08b].

Highway Node Routing

Highway node routing bases on a sequence of node sets $V := V_0 \supseteq V_1 \ldots \supseteq V_L$ [SS07]. The algorithm computes a hierarchy of overlay graphs for each node level l. The level-l overlay graph consists of a node set V_l and an Edge set E_l in such a way that it preserves all shortest path distances in V_l in correspondence with the underlying Graph $G_{l-1} = (V_{l-1}, E_{l-1})$. A bidirectional query algorithm takes advantage of the multi-level overlay graphs. As we will describe later in Section 3.4.2 for the Contraction Hierarchies, which form a special case of the Highway Node Routing, the query can be performed upwards in the hierarchy.

Contraction Hierarchies

Geisberger et al. introduced the Contraction Hierarchy method [GSSD08]. A Contraction Hierarchy uses a single node order and contracts the nodes in this order. The algorithm selects the node order through a priority queue, sorted by an estimate of how important it is to contract a node next. For the query the CH splits the graph into two distinct graphs. We call these graphs the *forward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow$ defined as: $E_\uparrow := \{(u,v) \in E|$ u contracted before v $\}$ and the *backward graph* $G_\downarrow := (V, E_\downarrow)$ with $E_\downarrow$ defined as: $E_\downarrow := \{(u,v) \in E|$ u contracted after v $\}$. The query algorithm then uses a modified version of the bidirectional Dijkstra's algorithm. We perform a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. If, and only if, an s-t-path exists in the original graph, then both search spaces will meet up at a node v that has the highest order of all nodes in an s-t-path. For our algorithm we adopt this method with a significantly different contraction procedure. We will discuss the algorithm of Contraction Hierarchies more thoroughly in Section 3.

Transit-Node Routing

Transit-Node Routing is currently the fastest of the available single-criteria speedup methods [BFSS07]. This algorithm selects a set of important (*transit*) nodes. For these, a distance table is computed. Furthermore, the algorithm computes a distance table for all relevant connections between the remaining nodes and the transit nodes. Since it turns out that only about ten such *access connections* per node are needed, one can “almost” reduce the routing in large road networks to about 100 table lookups. Interestingly, by this method the local queries, for which the shortest path does not touch any transit nodes, become the difficult queries. A way of solving this problem is to introduce several layers of transit nodes. For lower level transit nodes this approach need not store routes that touch higher level transit nodes. This speedup-technique reduces routing times to a few microseconds. Since this technique utilizes a hierarchical method to calculate a good set of transit nodes, these exceptionally good query times come at the cost of preprocessing times an order of magnitude larger than the underlying speedup-technique alone.

2.1.3 Combinations

SHARC

Combining shortcuts from contraction approaches and ArcFlags, Bauer and Delling introduced their algorithm SHARC in 2008 [BD08]. It is a generalization of the two-level ArcFlag method presented [MSS$^+$06]. The preprocessing of this approach iteratively partitions the graph into regions, bypasses unimportant nodes and removes arcs for which the algorithm has computed all flags. To the best of our knowledge an adaption of the SHARC algorithm is the most successful work on multi-criteria speed-up methods so far. However, the Pareto-SHARC only works if the number of different paths between two arbitrary nodes is small [DW09]. This is the case if the edge weight functions are either very similar or if we introduce additional restrictions on the shortest paths.

CALT

The CALT algorithm is a combination of the Contraction Hierarchy method and the ALT-algorithm described above presented by Bauer et al. [BDS$^+$08]. A more thourough discussion can also be found in [Sch08a]. It uses a Contraction Hierarchy algorithm to compute a hierarchy up to a certain level. The remaining overlay graph is called the core of the hierarchy. On this core Bauer computes a set of landmarks and the respective distances for the – in comparison to the original graph – considerably smaller overlay graph to use the the ALT-algorithm. In this thesis we will also make use of the ALT algorithm on the core of our hierarchy. We will therefore adapt the ALT method to work with the flexible objective function of our approach.

2.1.4 Time Dependent Routing

In addition to the single-criteria research the time dependent routing is a field of interest today. In the time dependent case we also utilize a variable edge weight function. In the time dependent case the real edge weight can only be deduced at the arrival time at the edge. This approach calculates an appropriate weight corresponding to the arrival at the desired edge. This differs significantly from our approach, since our edge weights do not depend on the actual traversal time of the edge. We decide on a parameter in advance and get an edge weight function for the entire search. The time dependent routing is especially useful to circumvent traffic jams on popular roads or for finding train connections. A basic assumption for the time dependent routing algorithms is the FIFO property. The FIFO property is also called the *non-overtaking property*, because it states that if an individual A leaves a node u via an edge (u, v) before another individual B, B cannot arrive at node v before A. Kaufman et al. proved that the shortest path problem in FIFO networks is polynomially solvable [KS93]. If the FIFO property is not fulfilled and we do not allow waiting, according to Orda et al. the problem becomes NP-hard [OR90].

In contrast to the single-criteria shortest path problem, much less research has been done on speedup-techniques for time dependent route planning. Multiple algorithms have been adapted to the time dependent case. To the best of our knowledge, the most successful adaptions use the Contraction Hierarchy algorithm or the SHARC algorithm [BGS08, Del09]

The results in terms of query times for SHARC and the time dependent Contraction Hierarchy are roughly comparable. Still, the time dependent Contraction Hierarchy suffers from a very high amount of preprocessing data. This results from the shortcuts being expensive in form of space consumption.

2.1.5 Multi Criteria Route Planning

So far multi-criteria route planning has not been a field of great interest. Among the approaches to the multi-criteria route planning problem that we know of most focus on the task of finding all Pareto-optimal paths. In this context a Pareto-optimal path is a Path that is not dominated by any other path. In the Pareto-shortest-path problem we process edge weigts in the form of a vector $(w_1, \dots, w_k)$. We say a weight L *dominates* another weight L' if $\forall i \in [1, k] : w_i leq w'_i$ and $\exists i \in [1, k] : w_i < w'_i$.

The straightforward approach to generate all paths is a generalization of Dijkstra's algorithm [Han79, Mar84, Möh99]. The generalization changes the label setting Dijkstra's algorithm to a label correcting algorithm. It turns out that a crucial problem for the multi-criteria algorithm is the number of labels we assign to a given node. The more labels we create, the more nodes we have to reinsert into the priority queue. It has been proven that the worst case number of labels can be exponential in V, yielding impractical running times [Han79]. Hence, [Han79] presents an FPAS for the bi-criteria shortest path problem.

To the best of our knowledge most bi-criteria algorithms presented so far use a special version of the A* search. Unfortunately these algorithms only yield small speedup factors and are therefore much less practical than the single-criteria algorithms of today. [RE09] shows an overview of some of the bi-criteria methods used so far. Most of them still suffer from large amounts of preprocessing and therefore only work on small input graphs. The most work has been performed on networks deriving from time table information. In these networks Müller-Hannemann et al. observed that the number of labels is often limited and therefore the brute-force approach for finding all Pareto-optimal paths is often feasible [MW01]. Still, these special conditions are only given for a very distinct set of graphs.

The only advanced speedup-technique we know of to be modified for multi-criteria routing is the SHARC algorithm. Delling managed to generate query times between 2 and 300 ms for inputs like the European road network, depending on different restrictions which we will discuss later [Del09]. As this clearly shows, the quality of the Pareto-approach largely depends on the underlying weight functions and the severe restrictions on the number of labels.

Even though the Pareto-optimal shortest path problem has attracted more interest in the past, the parametric shortest path problem has been studied. Karp and Orlin presented one of the first algorithms we know of to focus on the parametric shortest path problem [KO80]. They presented two algorithms with respective running times of $O(n^3)$ and $O(nm \log(n))$ for a modified shortest path problem. For a graph $G = (V, E)$ they select $E' \subseteq E$. For a constant edge weight $w(e)$ and a parameter $p \in \mathbb{R}$, they define the weight of an edge $e \in E'$ as $w(e) - p$. The weight of an edge $e \notin E'$ remains unchanged. In 1991 Young et al. improved the running times of Karp and Orlin's algorithm to $O(nm + n^2 \log(n))$ with a Fibonacci-Heap [NY91].

Both of these algorithms follow the same principle that Ziegelmann also describes in his thesis [Zie01]. But, in contrast to the cost function described above, Ziegelmann does not modify his edge weights only for a subset of edges. He modulates his parametric weight function as $w(e) := t + p \cdot c$ with $t, p, c \in [0, \infty]$. Starting at a shortest path tree T_0 that only represents a shortest path tree for a parameter value of zero, Ziegelmann calculates the maximal parameter p for which the tree T_0 is still a shortest path tree. For this Ziegelmann has to scan all non-tree edges. The tree T_0 is then modified to generate a shortest path tree T_1 for the parameter p. With this method Ziegelmann manages to generate running times of $O(m \cdot \sum_{v \in V} d_c(v))$, with $d_c(v)$ denoting the distance from the source to the node v in terms of the value c. Ziegelmann also argues that $\sum_{v \in V} d_c(v) \leq n^2 C$ for $C = max_{e \in E} c(e)$, with $c(e)$ denoting the c value of the edge e. Therefore,F he manages to provide an algorithm for finding all possible parametric shortest paths in $O(mn^2 C)$. Still, none of the described algorithms uses preprocessing to provide fast queries. Therefore, none of those seem feasible to provide the fast and flexible queries we seek.

2.2 Overview

This thesis will describe the first algorithm for providing fast and flexible queries on continental-sized road networks using linear weight functions. The thesis is structured as follows: First, we will present the basics needed to describe our algorithm. Section 3 will describe the data structures and algorithms our algorithm bases on. The section describes the definitions of graphs and priority queues and will give a short introduction to the shortest path problem. Following this, we will go on to describe a selection of speedup-techniques that have been developed for single-criteria algorithms. A special focus lies on the *Contraction Hierarchy* method presented by Geisberger et al., as we base our algorithm on the CH [GSSD08]. Furthermore, we also utilize the *ALT* method on a core of the hierarchy provided by our adapted CH. Therefore, we will give a more detailed discussion over these techniques here.

In Section 4 we will describe our contribution to the field of route planning. The section introduces the augmented problem of *flexible objective functions*. We will describe the differences between the classic single-criteria speedup-techniques and our new bi-criteria algorithm. Following an analysis of the properties of *linear weight term combinations*, we will present the description of some algorithms to modify the single-criteria methods presented in Section 3. We will describe how to adapt them to make them perform on our flexible objective functions. In this context, we will also focus on the differences to the algorithms based on Pareto-optimality that have been described recently. Furthermore, we will prove the correctness of our algorithm. In the comparison to other algorithms we will also show that Pareto-SHARC as one of the best other algorithms to provide flexible queries is only a heuristic.

In our experimental section in Section 5 we will present the results we get with our algorithm. We will analyze the algorithm in term of performance and robustness and also compare our algorithm more closely to the SHARC algorithm. To allow reproduction of our results, we will provide an overview of our exact experimental settings. After a description of the settings, we will present our main results.

In Section 6 we finally present the conclusion we drwe from our experiments. We also provide some ideas for further improvement of our algorithm.

In the appendices we give some further information and provide a closer look at the data-structures of our algorithm. The appendices contain an illustration of some of the query data-structures, as they are most important for our algorithm. For this we will provide a small example graph to give an overview of how we need to construct our search graph.

Chapter 3

Preliminaries

This chapter introduces the basic concepts and algorithms used throughout this thesis. For a more detailed discussion the reader may consult one of the many books available on the topic of algorithm theory, as "Network Flows" by Ahuja et al. [AMO93] or "Datastructures and Algorithms" by Kurt Mehldorn and Peter Sanders [KM08].

3.1 Graph Definitions

We define a *graph* $G = (V, E)$ as a set of *nodes* or vertices V and a set of *edges* or arcs $E \subseteq (V \times V)$. If (u, v) and (v, u) for $u \neq v$ represent different edges, we call the graph *directed*, else *undirected*. Usually we denote with $n = |V|$ the number of nodes and with $m = |E|$ the number of edges. The *reverse* graph $\bar{G} = (V, \bar{E})$ we define by the identical set of nodes and all the edges of E reversed: $\bar{E} = \{(v, u)|(u, v) \in E\}$.

We might assign a weight function $w : E \mapsto \mathbb{R}^+$ to the graph and thus make it a weighted graph. Note that this is the basic way to make a graph weighted. Since one of our algorithms needs integer values to function correctly, we will only use natural numbers for edge weights. Furthermore, we restrict ourselves to an interval of possible parameters $[0, p_{max}]$ to limit the amount of possible routes. Therefore, we will actually assign a linear weight function $w(e, p) : (E \times [0, p_{max}]) \mapsto \mathbb{N}$ to the graph.

We define a *path* $P := \langle n_1, \ldots, n_{k+1} \rangle$ as a sequence of edges $e_1 = (n_1, n_2), e_2 = (n_2, n_3), \ldots, e_k = (n_k, n_{k+1})$. Note that two paths $\langle s, \ldots, t \rangle$ and $\langle s, \ldots, t \rangle'$ do not have to be identical. If we want to describe an s-t-path that avoids a given node u, we write $\langle s, \ldots, t \rangle_{\not\ni u}$. We also might decide to actually give all nodes the paths uses. As a result we can define the *weight* of a path P at a parameter p as $w(P, p) = \sum_{i=1}^{k} w(e_i, p)$. In the single-criteria case, we will omit the p. We call the length of the shortest path between two nodes u and v $d(u, v, p) := \min_{\langle u, \ldots, v \rangle)}(w(\langle u, \ldots, v \rangle, p)$ the *distance* between u and v.

For a given graph $G = (V, E)$ a graph $G' = (V', E')$ with $V' \subseteq V$ forms an *overlay graph* if $\forall (u, v) \in V' \times V' : \forall p \in [0, p_{max}] : d'(u, v, p) = d(u, v, p)$ with $d'(\cdot)$ denoting the distance in G' and $d(\cdot)$ denoting the distance in G.

3.2 Priority Queues

A priority queue Q is a data structure designed to manage a set of elements. Each element is assigned a key with an associated strict weak ordering. We usually call this key the *priority* $p(q)$ for an element $q \in Q$. A priority queue supports the following operations:

- insert – inserts an element into the priority queue

- deleteMin – removes the element with the smallest priority from the queue and returns it
- decreaseKey – updates the priority of a node that already belongs to the priority queue. The new value has to be smaller than the old value.

Since the priority queue introduces a large overhead, a lot of theoretical work has focused on how to reduce the overhead of the priority queue operations.

With the use of advanced speedup-techniques the priority queue optimization has grown less important. Dominik Schultes argued that even simple and not tuned priority queues only induce a factor of around two of overhead, compared to highly tuned priority queues [Sch08b]. He bases this on the low number of priority queue accesses we need with todays fast algorithms. Therefore, it does not pay of anymore to optimize the decrease key operation, since they only occur seldom in sparse road networks.

3.3 Single-Criteria Shortest Path Problem

For a given graph $G = (V, E)$ and two nodes s, t of the single-criteria shortest path problem, the goal is to find $d(s, t)$. A well-known algorithm for finding a shortest path is Dijkstra's algorithm; pseudo-code for this algorithm is given with Algorithm 1. The algorithm iteratively computes the distance from a given node s to all other nodes $v \in V$. During a Dijkstra search, we distinguish between three states a node can take. We call a node either unreached, reached, or settled. At the beginning, all nodes besides the source-node s are unreached, and their respective distances are set to infinity ($d(v) = \infty$, $\forall v \in V/\{s\}$). We set s reached with $d(s) = 0$. The algorithm now uses a priority queue with a tentative distance for the priority key. In the beginning we use s as an initial insert into the priority queue. In each step we remove the node u with the lowest tentative distance and the node becomes settled. The distance of the node is now provably the final distance and will not be subject to change anymore. An illustration of the algorithm is given in Figure 3.1. To complete a single step of the algorithm, we look at all incident edges $(u, v) \in E$. If we have neither reached nor settled the node v, we insert it into the queue. We set its respective key to $d(u) + w(u, v)$. If we have already reached v and $d(u) + w(u, v) < d(v)$, we decrease the key of v.

It is important to realize that Dijkstra's algorithm will only produce a correct result for an input graph with $w(e) \geq 0$. Especially if the graph contains a reachable negative loop, the algorithm will not terminate. The algorithm performs in $O(|E| + |V| \cdot \log |V|)$ when we use a Fibonacci heap.

We can easily modify the algorithm to grow a shortest path tree. To do so, we just keep a parent pointer, which we update whenever we perform an insert or update operation. With these pointers we can follow a path backwards from the desired target to the source. We make no statement about the parents of unreached nodes.

All nodes reached during the search form the so-called search space. One can imagine the search space as a sphere that grows around the given source. The settled nodes are located inside the sphere, the reached nodes form the surface. This analogy shows that we can reduce the search overhead with a bidirectional search from the source and the target. When we use this so-called bidirectional Dijkstra we only have to continue the search until the search spheres intersect. We can stop a bidirectional search the moment we settle a node that we have already settled in the search in the other direction. It is not sufficient to stop the search the moment we have reached a node in both directions, since the distance may still be improved. We illustrate this incident in Figure 3.2. The node path over the

Algorithm 1: Dijkstra's algorithm

```
input  : A graph $G = (V, E)$, a source $s$, a target $t$
output : The shortest distance from $s$ to $t$
1  for $v \in V / \{s\}$ do
2      $d(v) = \infty$
3  $d(s) = 0$; min = $\perp$
4  insert $s$ in Q
5  while !Q.empty() and min $\neq t$ do
6      min = minimal element in $Q$
7      delete minimal element from $Q$
8      for $e$ = (min,$v$) $\in E$ do
9          if $d(\min) + w(\min, v) < d(v)$ then
10             $d(v) = d(\min) + w(\min, v)$
11             if $v \notin Q$ then
12                 insert $v$ in $Q$
13             else
14                 update $d(v)$ in $Q$
15 return d(t)
```

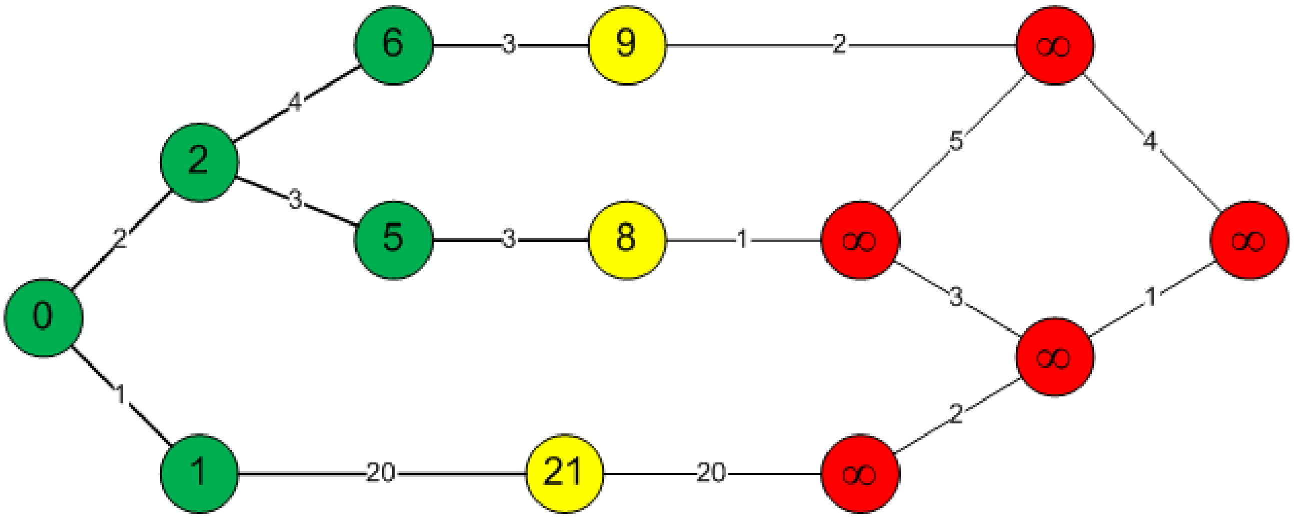


Figure 3.1: Illustration of the Dijkstra algorithm. Green nodes are settled, yellow nodes are reached. The red nodes are unreached. The label of a node shows its (tentative) distance.

only node yet reached by both searches obviously does not form a shortest path from the source to the target. Still, it suffices to stop the search the moment we settle a node in both search directions.

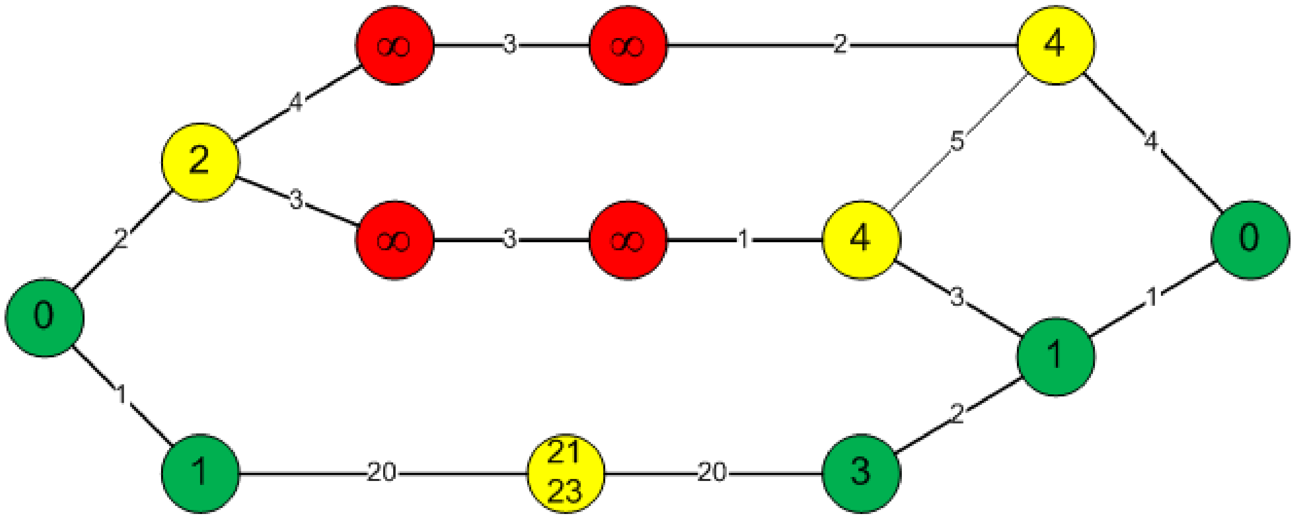


Figure 3.2: Illustration of a bidirectional Dijkstra's algorithm. Green nodes are settled, yellow nodes are reached. The red nodes are unreached. The label of a node shows its (tentative) distance.

3.3.1 Pareto-Paths

Let the edge weights consist of vectors $L = (w_1 \ldots, w_k)$ with $w_i \in \mathbb{R}_0^+$. As we mentioned in Section 2.1.5 , we say a weight L *dominates* another weight L' if $\forall i \in [1,k] : w_i leq w_i'$ and $\exists i \in [1,k] : w_i < w_i'$. We define the sum of two weight vectors by $L \oplus L' = (w_1 + w_1', \ldots, w_k + w_k')$. In a Pareto-optimal shortest path search, we are interested in the set of all non-dominated path lengths. We call this set the *Pareto-set* $\rho(s,t)$.

3.4 Speedup-Techniques

3.4.1 ALT

The ALT algorithm is a modification of the A* search, which has it origins in artificial intelligence studies [MSS+06].

A* search

The A* search improves Dijkstra's algorithm by adding a sense of direction. To an arbitrary graph $G = (V, E)$ we add a potential function $\pi : V \mapsto \mathbb{R}$. We can now define a reduced edge-weight for an edge $e = (u,v)$ by $w_\pi(u,v) = w(u,v) + \pi(v) - \pi(u)$. We call a potential function $\pi(\cdot)$ feasible if the following condition holds: $w_\pi(e) \geq 0 | \forall e \in E$. For a path $P = \langle n_1, \ldots, n_{k+1} \rangle$, the potential function only changes its weight $w_\pi(P) := \sum_{i=1}^{k} w_\pi(n_i, n_{i+1})$ by the constant value of $\pi(n_1) - \pi(n_{k+1})$. If $\pi(t) \leq 0$ and w_π is feasible, the potential $\pi(u)$ forms a lower bound on the distance $d(u,t)$. Figure 3.3 illustrates a graph with modified edge weights as described here. In an A*-Dijkstra algorithm the edge weights remain unchanged. Thus, we can use a changed priority key $k(u) = d(s,u) + \pi(u)$ in the priority queue. Therefore, we prefer nodes that have a shorter approximated distance to target. This

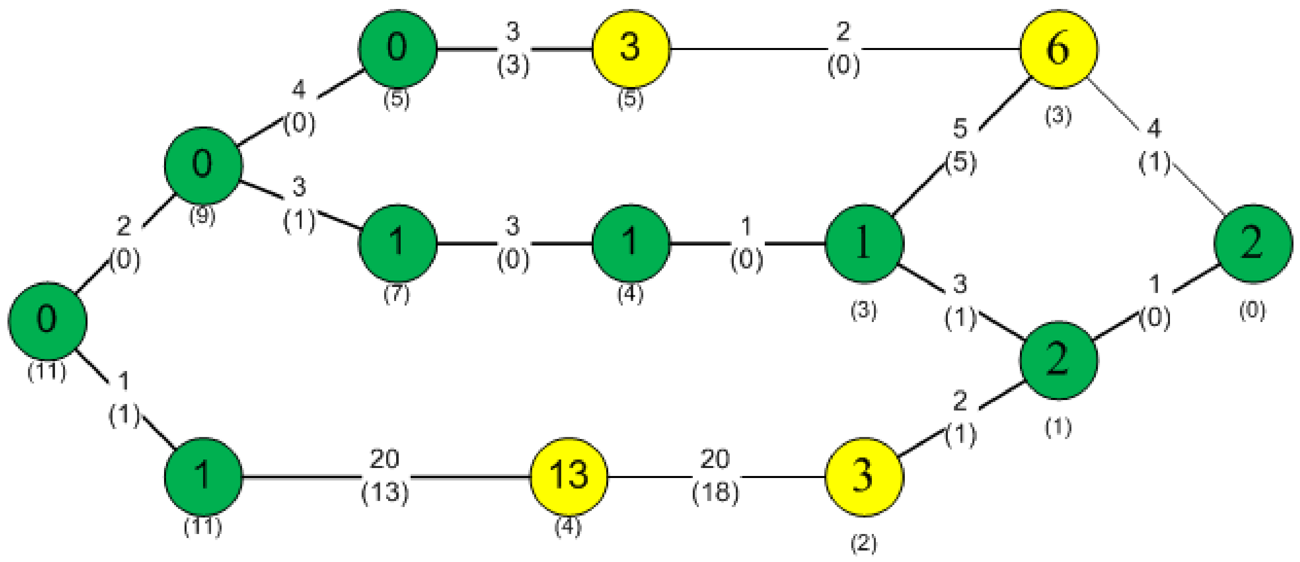


Figure 3.3: Illustration of a Dijkstra Algorithm with a potential function applied. The potential of the nodes is indicated beneath them. The reduced edge weights are shown under the original weights. Reached nodes are yellow, settled nodes green.

is obviously equivalent to running a Dijkstra's algorithm on a graph with reduced edge weights, since $d(s,u) + \pi(u) - (d(s,u) + \pi(u) - \pi(s)) = \pi(s)$, which is a constant value.

When combining the A* search with the bidirectional Dijkstra, several issues have to be addressed. We call two potential functions π_f and π_b consistent if $\pi_f + \pi_b = \text{const}$.

For a bidirectional A* search we distinguish between two different approaches, the *symmetric approach* presented by Pohl [Poh71] and the *consistent approach* proposed by Ikeda et al. [IHI+94]

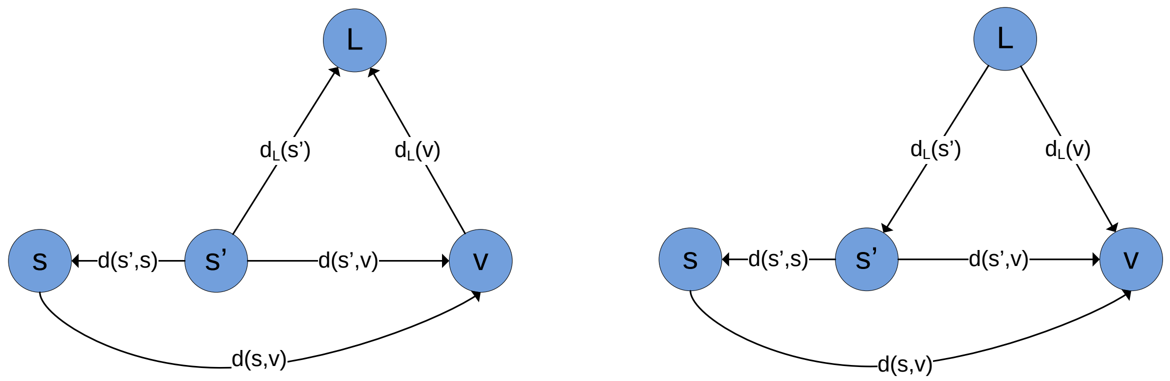
In the case of two non-consistent potential functions, we have to use a more complex stopping criterion. It is not sufficient to stop the search the moment we have settled a node in both directions, since we use different weight functions.

Symmetric Approach

When we use two non-consistent potential functions, we cannot stop the search the moment the searches have settled a node in both directions [Poh71]. Every time the search spaces meet at a node u we compute a tentative shortest path. If no shorter path has been encountered so far, the path becomes the current candidate. We stop the search when either the priority queues are empty or if we are about to settle a node that has a larger priority key than the stored tentative distance. We can prune the search at all nodes v, which we have already settled in the other direction. Since we have already found a shortest path from v to the target, we can gain no further information when we continue the search beyond v. If we use the symmetric approach, the search spaces normally meet faster. But due to the stopping criterion, the search has to continue for a long time after the search spaces have met.

Consistent Approach

With this approach we construct a consistent potential function out of two feasible potential functions π_f and π_b [IHI+94]. We define $\pi_{fc} = \frac{1}{2} \cdot (\pi_f(u) - \pi_b(u))$ and $\pi_{bc} = -\pi_{fc} = \frac{1}{2} \cdot (\pi_b(u) - \pi_f(u))$. To achieve better lower bounds we can add $\frac{1}{2}\pi_b(t)$ and $\frac{1}{2}\pi_f(s)$ to the forward and backward potential respectively. This does not compromise the consistency or the feasibility, since we only add a constant value. The consistent approach potential function is inferior to the direct potential functions that we



(a) Illustration with distances to the landmark.

(b) Illustration with distances from the landmark.

Figure 3.4: Illustration of the relations that are used during the computation of the lower bounds using proxy nodes.

use in the symmetric approach in terms of the lower bounds it provides. But due to the simpler stopping criterion we often have to settle less nodes, and thus the consistent approach often performs faster.

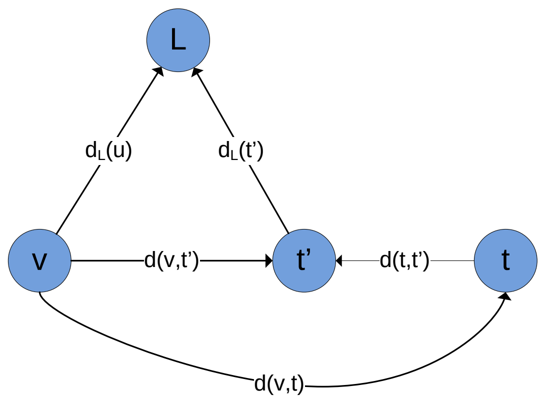
ALT algorithm

The ALT algorithm describes a way of choosing the potential function for an A* search and was proposed by Goldberg and Harrelson [GH05]. The algorithm uses a small set of nodes and the triangle inequality to calculate feasible lower bounds on $d(u, v)|u, v \in V$. We refer to this set of nodes as *landmarks*. In a preprocessing step we choose a subset of the nodes as landmarks. We also compute the distances from the landmarks to all other nodes and vice versa. Doing so, we can calculate the potentials with simple table lookups. Let $d_{\overrightarrow{L}}(u)$ denote the distance from u to a given landmark L and $d_{\overleftarrow{L}}(u)$ the distance from L to u then the triangle inequality yields $d(u, t) \geq d_{\overrightarrow{L}}(u) - d_{\overrightarrow{L}}(t)$ and $d(u, t) \geq d_{\overleftarrow{L}}(t) - d_{\overleftarrow{L}}(u)$. Goldberg and Harrelson introduced a method of how to improve the result using $\max_L(d_{\overrightarrow{L}}(u), d_{\overleftarrow{L}}(u))$ as potential function [GH05]. To avoid the overhead of using all landmarks, Goldberg et al. argue that one may use only a subset of them at the start of the algorithm and add more landmarks as necessary [GW05]. The choice of the landmark set is crucial for the performance of the algorithm. Viable approaches include the avoid algorithm and the maxCover algorithm. Usually the maxCover algorithm provides better results but has longer preprocessing times than the avoid algorithm. For a more detailed discussion on the selection methods we refer to [GW05].

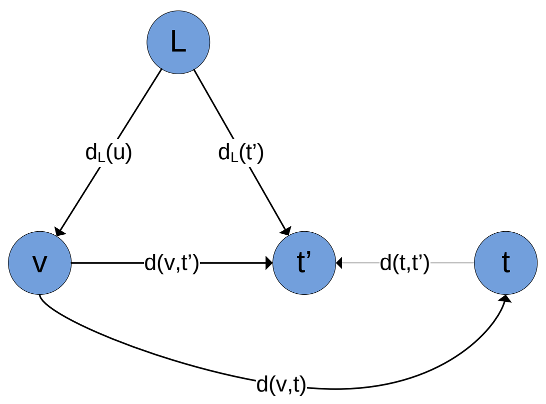
Core ALT

For the use on a hierarchy core, we have to adapt the ALT algorithm, since the source and the target of a search may not belong to the core. Since we only perform the ALT preprocessing on the core, we might not know the distances $d_{\overrightarrow{L}}(t), d_{\overleftarrow{L}}(t)$ and $d_{\overrightarrow{L}}(s), d_{\overleftarrow{L}}(s)$ respectively. To deal with this situation, we introduce so-called proxy nodes as already defined by Bauer et al. [BDS+08]. We define a *proxy node* $s' \in V_C$ of a node $s \in V/V_C$ as the node with the minimal distance from s. There are two possible candidates to choose as a proxy node to a node s. The node s' that minimizes $d(s, s')$ or the node s' that minimizes $d(s', s)$. As we will show in the following, it is sufficient to only know $d(s', s)$ to compute a lower bound on the distance $d(s, v)$.

As illustrated in Figure 3.4(a) we can find the following triangle equalitys. If we look at the nodes s', s and v the triangle inequality yields $d(s', v) \leq d(s', s) + d(s, v)$. With s, v and L respectively, the



(a) Illustration with distances to the landmark.



(b) Illustration with distances from the landmark.

Figure 3.5: Illustration of the relations that are used during the computation of the lower bounds using proxy nodes.

triangle inequality yields $d_{\overrightarrow{L}}(s') \leq d(s', v) + d_{\overrightarrow{L}}(v)$. When we combine this two inequalities we get $d(s, v) \geq d_{\overrightarrow{L}}(s') - d_{\overrightarrow{L}}(v) - d(s, s')$. Analogously we can show that
$d(s, v) \geq d_{\overleftarrow{L}}(v) - d_{\overleftarrow{L}}(s') - d(s', s)$ holds.

In the same matter we can deduce lower bounds for the distances $d(v, t)$. Figure 3.5 illustrates the same relations for the target as illustrated in Figure 3.4 for the source.

The resulting inequalities are: $d(v, t) \geq d_{\overrightarrow{L}}(v) - d_{\overrightarrow{L}}(t') - d(t, t')$ and $d(v, t) \geq d_{\overrightarrow{L}}(t') - d_{\overrightarrow{L}}(u) - d(t, t')$

To use these inequality's as a potential function we need to know the distances $d(s', s)$ and $d(t, t')$. They can either be precomputed and stored for every node, or can be calculated on the fly. In a lot of hierarchies only a few steps are needed to reach the core. Therefore, we can easily compute the needed distances on the fly. The overhead introduced by the proxy searches in the reverse graph is very small compared to the work that has to be done in the core.

Figure 3.6 illustrates the CALT algorithm. In the figure the Landmarks themselves are not shown for simplicity reasons. We represent the two phases of the search with differently colored nodes. We continue the hierarchical part until the nodes circled in bold, called entry points, have all been found. From this point on we perform a bidirectional ALT algorithm using only the nodes in the core.

3.4.2 Contraction Hierarchies

Given a graph $G = (V, E)$, a Contraction Hierarchy is a special case of Highway Node Routing with $|V|$ levels as shown by Geisberger [Gei08]. For the contraction the nodes are organized according to their respective levels. In each step, as illustrated in a simplified version in Algorithm 2, we choose the next node x according to this order and contract it. In the contraction process we inspect all paths $P = \langle u, x, v\rangle$. If a path $\tilde{P}$ exists that does not contain x with $w(P) \geq w(\tilde{P})$, we do not have to do anything. We call such a path a witness. If the path $\tilde{P}$ does not exist, we have to insert a so-called shortcut to preserve the shortest path distances.

One important part of this process is the order in which the nodes are chosen for contraction. For example, on long paths the order can result in a linear search time or a logarithmic query. The number of created shortcuts also depends on the contraction order. To determine the order in which we contract the nodes, a lot of different criteria can be used.

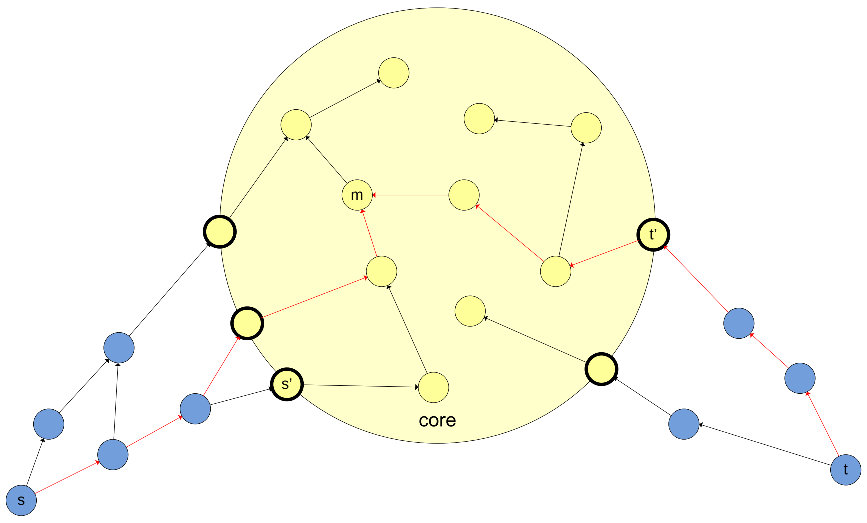


Figure 3.6: Illustration of a core search. The core-entry points are drawn in bold, the meeting point of the searches is labeled m. The source and target nodes as well as the respective proxy nodes are also labeled.

Algorithm 2: Simplified Contraction Hierarchies

```
input  : A graph $G = (V, E)$
output : A graph $G' = (V, E')$
1 while $\exists$ uncontracted node do
2     $v$ = currently most promising choice
3     for $(u, v) \in E$ with $u$ not contracted do
4         for $(v, w) \in E$ with $w$ not contracted do
5             if $(u, v, w)$ “may” be the only shortest path from $u$ to $w$ then
6                 $E := E \cup (u, w)$, $c(u, w) := c(u, v) + c(v, w)$
7 return $G$
```

Node Ordering

In the approach presented by Geisberger et al., the order is calculated during the contraction process itself [GSSD08]. A priority queue is used in their algorithm to heuristically choose a node to contract. This queue orders the nodes according to a linear combination of several terms that will change over the contraction process. Since every contraction can influence the remaining graph, a lot of priority terms have to be updated. For this the following techniques can be used.

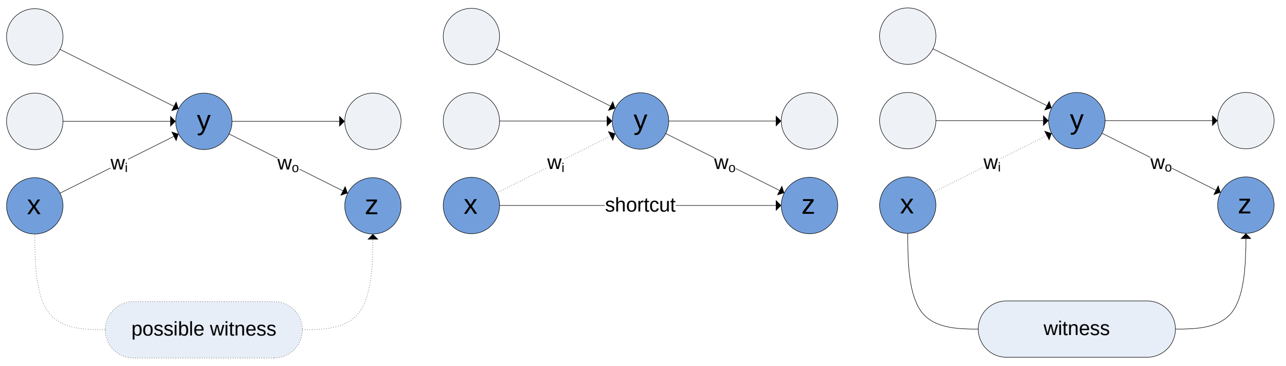
- *periodic updates*: periodically all priorities are reevaluated and the priority queue is rebuilt.
- *neighbor updates*: whenever a node x is contracted, the priority terms of all its neighbors are updated. This is a fair approximation of the direct impact a node contraction can have. Note that more nodes besides the direct neighbors can be affected by the contraction of a node.
- *lazy updates*: whenever a node x is contracted, its priority key is recalculated. If it does not exceed the priority key of the second largest element x', we contract x. Otherwise, the nodes priority is updated and the selection process reiterated. If we experience more than a certain amount of reiterations in a period of a thousand contractions, the lazy update mechanism will trigger a complete queue update. The number of lazy update iterations serves as an approximation for the actuality of the remaining node priorities.

For the actual node order Geisberger et al. proposed a wide variety of terms to classify the "importance" of a given node. In the following we want to shortly discuss the terms we also use for our algorithm. For a more detailed discussion of the subject we refer to [GSSD08].

Edge Difference: We calculate the edge difference using the searches that are used to find witness paths. This value presents the change in the number of edges for the resulting overlay graph during the contraction of a node v. Arguably this is one of the most important terms for the priority key since the number of edges has to be kept small.

Uniformity: Geisberger et al. showed that focusing on small regions will lead to more linear hierarchies where queries follow long paths. They also presented two possible approaches to more uniformly contract nodes all over the graph:

- *Deleted Neighbors*: The deleted neighbors approach counts the number of already contracted neighbors. The term deleted stems from the priority queue approach, where a contracted node has to be deleted from the priority queue as the node with the minimal key. If a node has a lot of already contracted neighbors, including nodes reached over shortcuts, it will be contracted late. We can update this parameter during the contraction by modifying the counters for all neighbors.
- *Voronoi Regions*: In the overlay graph containing only the non-contracted nodes, the *Voronoi-Region* $R(v)$ is defined as the set of nodes in the input graph that is closer to v than to any other node in the overlay graph. Geisberger et al. propose to use the square root of the size of the Voronoi-Region as a term for the priority queue key. Geisberger et al. showed that under certain conditions only $O(n \cdot \log n)$ steps of the Dijkstra algorithm are necessary to compute the Voronoi-Regions during the contraction process, thus making them reasonably efficient. Since the regions can only grow, this term does not render the lazy update process, as described above, incorrect. In the parametric shortest path problem we use the parameter 0 to compute the Voronoi regions.
- *Shortcut Original Edges Count*:The term of the number of original edges a shortcut represents is another possible node ordering term. It helps to keep a balance in the length of the shortcuts.



(a) Witness search problem

(b) Contraction without an existing witness path

(c) Contraction with existing witness path

Figure 3.7: Illustration of the witness search problem and the two possible situations.

For the node ordering we use the square root of the number of original edges. In addition to the simple count of the number of edges we also utilize a further term involving the number of original edges a shortcut represents. We calculate for every needed shortcut the quadratic mean between the number of edges of the two edges a shortcut represents. The *Shortcut Distance Mean* favors more balanced shortcuts.

- *Edges Added* and *Edges Removed*: Geisberger proposed the edge difference as a node ordering term. We actually split this term into two terms. The edges added and the edges removed. During the contraction of a node, all edges from and to the respective node are removed from the graph. We call those the removed edges. The edges we add during the witness searches for all necessary paths form the added edges. If we weight both terms equally, this results in the edge difference, as we subtract the value of the deleted edges from our priority term.

- *Search Space Size*: A further criterion is the size of the search space. With this term we may choose to favor nodes in less dense areas of the graph since the search space will contain more edges in dense areas.

Let $G = (V, E')$ be the overlay graph for the current contraction step. Following the node order determined by the priority queue, we individually contract each node y. During the contraction of a node y we look at every pair of nodes x, z with $(x, y) \in E'$ and $(y, z) \in E'$. For each pair we perform a Dijkstra's algorithm, restricted onto a local area around the node x, omitting the node y in the search. This situation is illustrated in Figure 3.7(a). If we cannot find a witness path as described above for the path $\langle x, y, z \rangle$, we add a shortcut edge to the graph. This shortcut represents the path over y. We illustrate this situation in Figure 3.7(b). We can then omit the incoming edges of y, since we insert a shortcut for every pair of edges of y that may be part of a shortest path. Finally, Figure 3.7(c) depicts the situation with an existing witness path. In contrast to the situation depicted in Figure 3.7(b), we do not have to insert the shortcut edge. The witness searches are therefore essential to keep the number of edges for the overlay graph small. If we do not search for witnesses, the graph will converge against a complete graph. Note that, due to the limitation of the witness-search, we may insert some shortcuts even though another shortest path exists. This will not affect the correctness, since we only represent an unneeded or a parallel path.

Query

To describe the query of a Contraction Hierarchy, we have to make some definitions first. Let $\mathcal{L}(\cdot)$ denote the level of the node in the Contraction Hierarchy. We divide the graph $G = (V, E)$ into a *forward graph* $G_\uparrow := (V, E_\uparrow)$ with $E_\uparrow$ defined as: $E_\uparrow := \{(u, v) \in E | \mathcal{L}(u) < \mathcal{L}(v)\}$ and a *backward graph* $G_\downarrow := (V, E_\downarrow)$ with $E_\downarrow$ defined as: $E_\downarrow := \{(u, v) \in E | \mathcal{L}(u) > \mathcal{L}(v)\}$. In the Contraction Hierarchy query we use a modified version of the bidirectional Dijkstra's algorithm. It consists of a forward search in $G_\uparrow$ and a backward search in $G_\downarrow$. [GSSD08] and [Sch08b] contain a proof that for an arbitrary shortest path $P = \langle s, \ldots, t\rangle$ a Path P' exists in the contracted hierarchy and the searches in $G_\uparrow$ and $G_\downarrow$ will meet up at node $u \in P'$ with $\mathcal{L}(u) = \max\{\mathcal{L}(v) | v \in P'\}$.

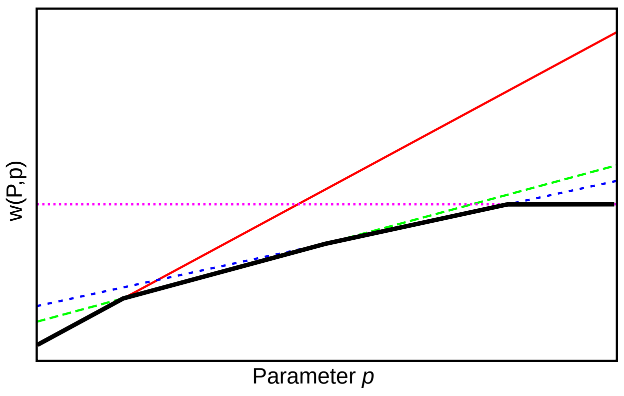
Chapter 4

Flexible Objective Functions for Contraction Hierarchies

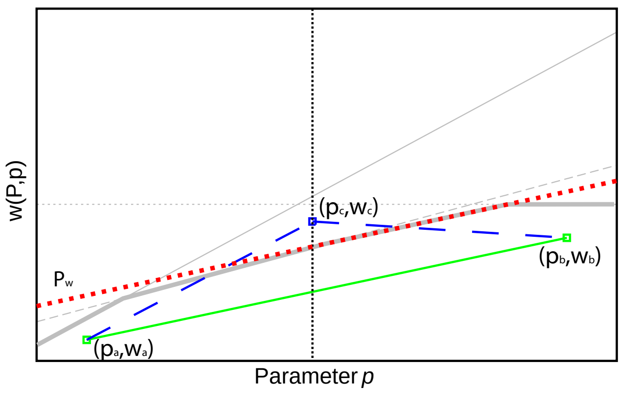
A lot of work in multi-criteria routing has been done regarding Pareto-optimality. In contrast to the Pareto-approach, which focuses on finding all possible non-dominated routes between a source and a target, we try to explore alternative methods to add flexibility to the problem of finding shortest paths. For our multi-criteria Contraction Hierarchy, in short MCCH, we try to modulate the possible choices with linear functions. We chose to concentrate on a combination of two weight terms. If we use t_1 and t_2 to denote the weight terms we can combine these terms in two ways to gain a weight function $w(p)$, where p specifies a selectable parameter. The first method, which we already described in the preliminaries, is given by $w_1(p) := t_1 + p \cdot t_2$. In this case we can choose $p \in \mathbb{R}_0^+$ or $p \in \mathbb{N}_0$. Another possibility would be to combine the terms in the following way: $w_2(p) := (1 - p) \cdot t_1 + p \cdot t_2$, for $0 \leq p \leq 1, p \in \mathbb{R}$. In this work we chose to use the first approach with the weight function w_1, but we can easily reduce the second weight function to $w_2(p) = t_1 + p \cdot (t_2 - t_1)$, which forms a similar weight function compared to the weight function we chose. By modulating the edge-weights as a linear function, we will show that we can provide fast and flexible routing for even continental sized route networks. To be more exact: We define our weight function to consist of two *weight terms*, which we will refer to by τ and κ. For the parameters we choose a distinct *parameter interval* $[0, \dots, p_{max}]$ and $p_{max} \in \mathbb{N}_0$. Therefore, we define our *weight function* as $w(p) := \tau + p \cdot \kappa$ for all $p \in [0, \dots, p_{max}]$. We will describe how to calculate the used weight terms in the experimental section.

4.1 The Multi Criteria Contraction Hierarchy Search Space

In contrast to a normal Dijkstra's algorithm we have several issues that need to be addressed. Since we utilize the Contraction Hierarchy method for our approach, we have to modify the node ordering and the contraction to suit our enhanced needs. Especially during the contraction, the witness searches need adjustment. Instead of a single shortest path we have to find a set of paths, comparable to the Pareto-set so that for every parameter p an alternative route exists. Thus, a single Dijkstra's algorithm will not suffice as a witness search. To further discuss this, we have to make a few observations first. Obviously the weight of a path P will always be a linear function, as it is the sum of linear functions. In a Cartesian coordinate system, let the x-axis represent the parameter values and the y-axis represent the weight of a path P at the parameter p: If we depict the weight function $d(s, t, p)$ for a fixed source s and target t with $d_{s,t}(p)$, we can show that $d_{s,t}(p)$ is concave. A graphical illustration of multiple weight functions and the resulting $d_{s,t}(p)$ can be found in Figure 4.1(a).



(a) Illustration of $d_{s,t}(p)$ in bold for multiple weight functions of different paths between a source and a target.



(b) Antagonism in Proof of Lemma 4.1

Figure 4.1: Illustration for Lemma 4.1

Lemma 4.1 *The function $d_{s,t}(p)$ is concave in p.*

Proof. We will prove this by showing that the area under $d_{s,t}(p)$ is convex. Let us assume that the area under $d_{s,t}(p)$ is not convex. In that case we can find two distinct points a and b with $a = (p_a, w_a)$, $b = (p_b, w_b)$ and $w_a \leq d_{s,t}(p_a) \wedge w_b \leq d_{s,t}(p_b)$, W.L.O.G. $p_a < p_b$ and we can find a $0 \leq t \leq 1, t \in \mathbb{R}$ with $c := (p_c, w_c) = (a + t \cdot (b - a))$ and $w_c > d_{s,t}(p_c)$.

Thus, we can find a path $P_w = \langle s, \ldots, t \rangle$ with $w(P_w, p_c) < w_c$, as $d_{s,t}(\cdot)$ forms the minimum over all paths between s and t. This path P_w can be chosen as the path with the minimal weight at p_c over all paths.

Since $w_a \leq d_{s,t}(p_a)$ and $w_b \leq d_{s,t}(p_b)$ hold – as $d_{s,t}(p)$ is the minimum over all paths from s to t including P_w – $w_a \leq w(P_w, p_a)$ and $w_b \leq w(P_w, p_b)$ hold. Therefore, we have exactly two common points between $w(P_w, p)$ and $(a + p \cdot (b - a)$ for $p \in [p_a, p_b]$. One of these points is located in the interval $[p_a, p_c)$ and the other is located in the interval $(p_c, p_b]$. In those intervals the common point is unique, since we compare to straight lines that fulfill the $\leq$-relation at a and b and the $>$-relation at p_c respectively. We depict this situation in Figure 4.1(b). Therefore, we have exactly two distinct common points between two straight lines. Since there can either be no, one or an infinite number of common points between two straight lines, the point c cannot exist. Therefore, we have proven $d_{s,t}(p)$ is concave. □

4.2 Construction

We can compare the construction part of a multi-criteria Contraction Hierarchy to the single-criteria *Contraction Hierarchies (CH)* introduced by Geisberger et al. [GSSD08]. We select a total order over all nodes, and contract all nodes in this order. First we will discuss the node ordering process. After this we will focus on the contraction process and introduce some approaches to realize hierarchies of huge inputs as the European road network.

4.2.1 Node Ordering

As shown by Geisberger et al., a single-criteria CH is very sensitive to the underlying node order [GSSD08]. In our approach we have to find a node order that will represent all parameters in the interval $[0, p_{max}]$. For small parameter values the time metric has the biggest influence on the shortest paths. Along with the parameter we allow to gradually increase the amount of cost that we take into account. Therefore, we chose a more general node order than the order proposed by Geisberger et al., which only focuses on single-criteria hierarchies [GSSD08]. We suspect a node order that bases mostly on uniformity, as we described in the preliminaries, to deliver the best possible results. We chose to concentrate on the priority terms *added shortcuts*, *deleted neighbors*, the *edges removed*, the size of the *search space*, the *shortcut original edges count*, the *shortcut distance mean* and the *Voronoi regions*. For the contraction process we choose the factors for *added shortcuts* and *deleted neighbors* 150, the *edges deleted* 40, the *search space* 80, the *shortcut original edges count* and the *shortcut distance mean* 10 the *Voronoi regions* 50. As we will discuss later, we could not find a single node order, which was general enough to represent all parameters in a good way. We will discuss this problem more closely in Section 4.3.1.

4.2.2 Lazy updates

As Geisberger et al. already discussed, the priority terms for the node order change during the contraction [GSSD08]. Whenever we contract a node, we update the priority terms of all of its direct neighbors. Still, with this method we do not reach all nodes with this method that might be affected. We therefore also perform lazy updates. Every time we remove the node with the currently minimal key from the priority queue, we perform a simulated contraction to calculate the current nodes priority. If the node remains the one with the highest priority we continue with the current contraction. If the node becomes less important than the following node, we reinsert it into the priority queue and continue with the next node in the same matter. Whenever we perform more than a certain number of lazy updates within a given period, we perform an update of the priority terms for the whole priority queue.

4.2.3 Contraction

In this section we will focus on the details of the contraction process, especially on how we locate witness paths. After this we will show how to improve the performance of the algorithm and how to deal with the situation of a multi-criteria Contraction Hierarchy.

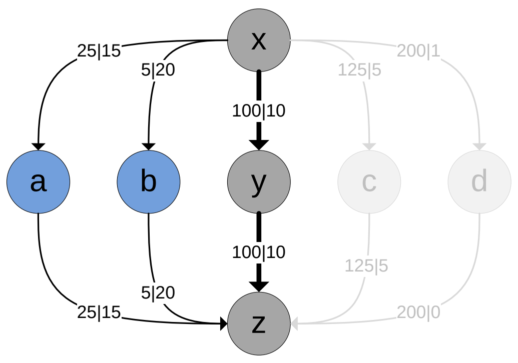
As we already discussed earlier, we search for witness paths during the contraction. A witness path is an alternative route that allows us to omit a shortcut. In the single-criteria case we need only a single path as a witness. In the multi-criteria case a single path might not suffice.

A simple implementation of the witness searches could be to perform a single-criteria witness search for every parameter $p \in [0, p_{max}]$, which is infeasible for large $|[0, p_{max}]|$. We could also perform a Pareto-search. But as we already discussed in Section 4.3, the number of labels might get to large. Instead we do something more tailored to our objective function.

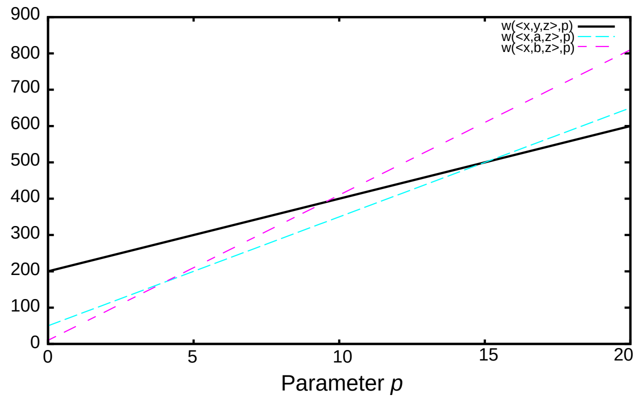
As we observed in Lemma 4.1, the minimum over all shortest paths from a source to a target forms a concave function. This observation implies another observation.

Lemma 4.2 *Let $\langle x, y, z\rangle$ be a possible shortcut. The interval $[L, U]$ can be partitioned into three, possibly empty partitions $[L, L']$, $[L'+1, U'-1]$ and $[U', U]$ with $L', U' \in \mathbb{N}$ and the following three properties:*
(a) If $[L, L']$ is not empty, then there exists a single witness path for all $p \in [L, L']$. (Figure 4.2(a), 4.2(b))
(b) If $[U', U]$ is not empty, then there exists a single witness path for all $p \in [U', U]$. (Figure 4.2(c), 4.2(d))
(c) If $[L'+1, U'-1]$ is not empty, then for all values of p in it, there exists no witness path for all $p \in [L'+1, U'-1]$ (Figure 4.2(g), 4.2(h))

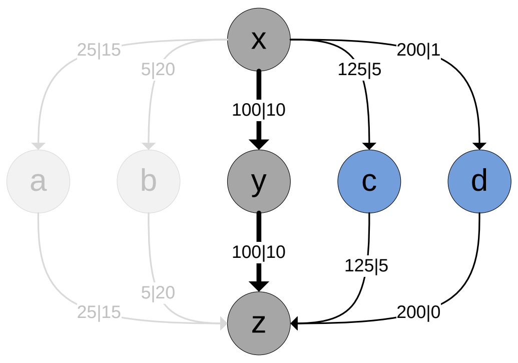
This lemma implies that two paths will suffice to decide on the size of the necessety interval of a shortcut. We depict all important possibilities in Figure 4.2. Lemma 4.2 also implies that we



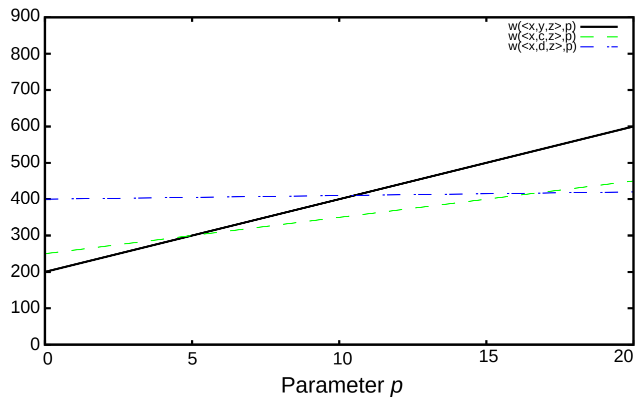
(a) Partwise witness only on lower end



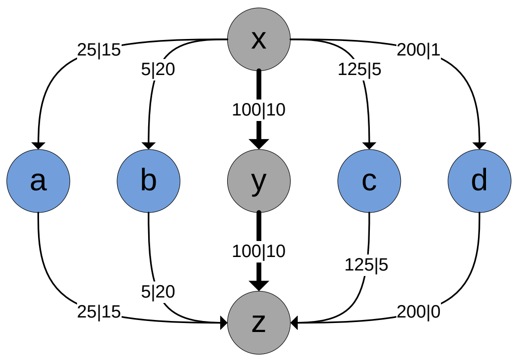
(b) Cost function for Figure 4.2(a)



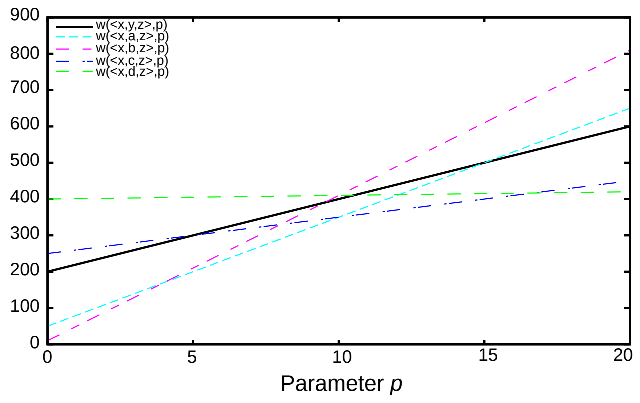
(c) Partwise witness only on upper end



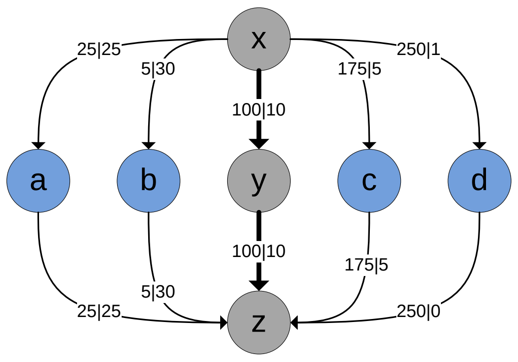
(d) Cost function for Figure 4.2(c)



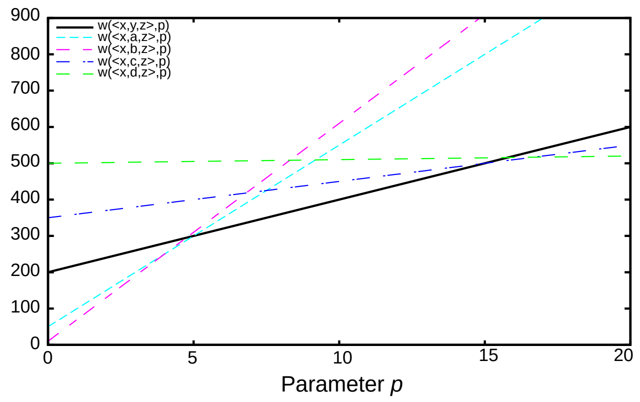
(e) Witness on whole parameter interval



(f) Cost function for Figure 4.2(e)



(g) Partwise witnesses on both ends



(h) Cost function for Figure 4.2(g)

Figure 4.2: Sample graphs with corresponding weight functions

can constrain necessary shortcuts to small parameter intervals on which they are needed. Therefore, we add a *necessity interval* $[L'+1, U'-1]$ to all edges, indicating for which parameter values the shortcut is necessary. Edges of the original graph are necessary for the whole parameter interval $[0, p_{max}]$. These edges can also be restricted with interval reduction, when we find a path during our search that is shorter for some parameters p. During the contraction of the path $\langle x, y, z\rangle$ with (x, z) necessary on the interval $[L_x, U_x]$ and (y, z) necessary on the interval $[L_y, U_y]$ we only have to search for witnesses on $[L_x, U_x] \cap [L_z, U_z]$. We observed that the limitation to the minimal necessity interval yielded a much better performance compared to the unconstrained parameter intervals.

Proof (of Lemma 4.2). Let $\widetilde{w}(p) := w(\langle x, y, z\rangle, p)$ be the weight function of the possible shortcut. We select a parameter L_{max} and a parameter U_{min} as follows:

$$L_{max} := \max\left\{ l | \exists P = \langle x, \dots, z\rangle_{\not\ni y} : w(P, l) \leq \widetilde{w}(l) \wedge w(P, l+1) > \widetilde{w}(l+1) \right\},$$

$$U_{min} := \min\left\{ u | \exists P = \langle x, \dots, z\rangle_{\not\ni y} : w(P, u) \leq \widetilde{w}(u) \wedge w(P, u-1) > \widetilde{w}(u-1) \right\}.$$

The definition of L_{max} gives that we can find a single witness path for every $p \in [L, L_{max}]$. This holds, as we examine linear functions. Therefore, $\exists P = \langle x, \dots, z\rangle_{\not\ni y} : \forall p \in [L, L_{max}] : w(P, p) \leq \widetilde{w}(p)$. In the same matter the definition of U_{min} gives that we can find a single witness path for every $p \in [U_{min}, U]$.

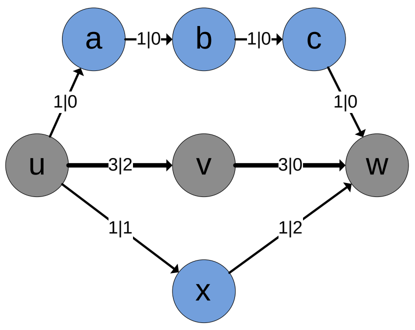
First, let us assume that $L_{max} \geq U$. This is equivalent to $U_{min} \leq L$. In this case $L' = U$. This results in $[U', U] = \emptyset$ and $[L'+1, U'-1] = \emptyset$.

Now let us assume that $L_{max} < U$ and $L_{max} + 1 \geq U_{min}$. If $L_{max} < U$, but $L_{max} + 1 \geq U_{min}$, the two paths that provided L_{max} and U_{min} cover the whole parameter interval and therefore form a multi-criteria witness. In Figure 4.2(e) these paths would be represented by the paths $\langle x, b, z\rangle$ and $\langle x, c, z\rangle$. Therefore, we can choose $L' = L_{max}$ and $U' = L_{max} + 1$ with $[L'+1, U'-1] = \emptyset$.

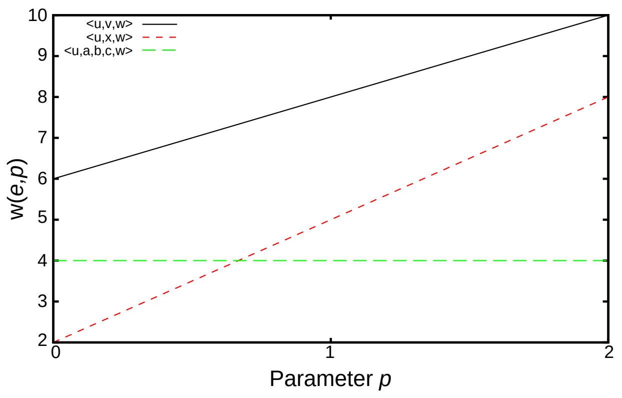
If neither is the case, we cannot find a witness for any parameter $p \in [L_{max}+1, U_{min}-1]$. Let us assume we could find a path $\widetilde{P} = \langle x, \dots, z\rangle_{\not\ni y}$ with $w(\widetilde{P}, p') \leq \widetilde{w}(p')$ and $p' \in [L_{max}+1, U_{min}-1]$. Since $p' \in [L_{max}+1, U_{min}-1]$, neither $w(\widetilde{P}, L) \leq \widetilde{w}(L)$ nor $w(\widetilde{P}, U) \leq \widetilde{w}(U)$ can hold. Otherwise the path $\widetilde{P}$ would conflict with the choice of L_{max} and U_{min}. Therefore, we have exactly two common points. One in the parameter interval $[L, p'-1]$ and one in the interval $[p'+1, U]$. Since we deal with linear function of degree one, we cannot have exactly two common points. As a result of this antagonism the path $\widetilde{P}$ cannot exist. Therefore, we cannot find a witness for any parameter $p \in [L_{max}+1, U_{min}-1]$. Thus, we can select $L' = L_{max}$ and $U' = U_{min}$. □

Note that we do not need to find the best possible paths for all parameters p. Figure 4.3(a) illustrates a possible contraction situation. Possible witnesses for the path $\langle u, v, w\rangle$ are the paths $\langle u, x, w\rangle$ and $\langle u, a, b, c, w\rangle$. Let us assume a parameter interval $[0, 2]$. Neither of the possible witness paths is smaller than the other on the whole parameter interval. Still, both paths are a valid witness paths for the path $\langle u, v, w\rangle$, as can be seen in Figure 4.3(b).

This observation allows us to propose a parameter increasing witness search to find a multi-criteria witness. If we can find a witness for every parameter p, the parameter increasing witness search will suffice to finde a witness for every p. If on the other hand we cannot find a witness, we want to restrict the necessetiy interval of the inserted edge as much as possible. Therefore, we also introduce a parameter decreasing witness search. The combination of those algorithms allows us to restrict the necessity interval at both ends. We expect this algorithm to work reasonably well, since Lemma 4.2 suggests that a few iterations of Dijkstra's algorithm may suffice on average to find a witness.



(a) Possible witness search situation for the contraction of node v.



(b) Weight functions for Figure 4.3(a)

Figure 4.3: Sample graph and corresponding weight function

Algorithm 3: Parameter Increasing Witness Search

```
input  : source $x$, target $z$, avoiding node $y$, an interval $[L, U]$
output : a parameter $p$ so that we can find an MCCH witness on $[L, p]$
1 p = L
2 while $p$ smaller or equal to $U$ do
3     //Calculate the shortest path $\langle x, \dots, z \rangle_{\not\ni y}$ for parameter $p$
4     P = PerformDijkstra($x$,$z$,$y$, $p$)
5     if $w(P, p)$ > $w((x,y,z),p)$ then
6         return $p - 1$
7     $p'$ = $\max_k$ with $w(P, k) \leq$ w(($x$,$y$,$z$),$k$)
8     $p = p' + 1$
9 return $U$
```

Algorithm 4: Parameter Decreasing Witness Search

```
input  : source $x$, target $z$, avoiding node $y$, an interval $[L, U]$
output : a parameter $p$ so that we can find an MCCH witness on $[p, U]$
1 p = U
2 while $p$ greater or equal to $L$ do
3     //Calculate the shortest path $\langle x, \dots, z \rangle_{\not\ni y}$ for parameter $p$
4     P = PerformDijkstra($x$,$z$,$y$, $p$)
5     if $w(P, p)$ > $w((x,y,z),p)$ then
6         return $p + 1$
7     $p'$ = $\min_k$ with $w(P, k) \leq$ w(($x$,$y$,$z$),$k$)
8     $p = p' - 1$
9 return $L$
```

Algorithm 5: Witness Search

```
input  : source $x$, target $z$, avoiding node $y$, an interval $[L, U]$
output : the minimal interval $[L'+1, U'-1]$ for which a shortcut is necessary
1 $L'$ = ParameterIncreasingWitnessSearch( $x$,$z$,$y$,$[L,U]$ )
2 if $L' \geq U$ then
3     return $\emptyset$
4 //At his point we know that $[L'+1, U'-1] \neq \emptyset$
5 $U'$ = ParameterDecreasingWitnessSearch( $x$,$z$,$y$,$[L,U]$ )
6 return $[L'+1, U'-1]$
```

Lemma 4.3 *If a witness exists for every $p \in [L, U]$, Algorithm 3 suffices to find a witness for the complete interval $[L, U]$. If for some $p \in [L, U]$ no witness exists, Algorithm 3 and Algorithm 4 can be used to calculate the minimal necessity interval $[L'+1, U'-1]$ as described in Lemma 4.2. Therefore, Algorithm 5 will find the minimal necessety interval for a possible shortcut.*

Proof. If the interval $[L'+1, U'-1]$ is empty following condition holds: $\forall \widetilde{p} \in [L, U] : \exists P = \langle x, \ldots, z \rangle_{\not\ni y} : w(P, \widetilde{p}) \leq w(\langle x, y, z \rangle, \widetilde{p})$. Therefore, Algorithm 3 will find a suitable P with $w(P, p) \leq w(\langle x, y, z \rangle, p)$ in every iteration. Since the weight functions are linear functions, the calculated path P is a valid witness for every $\widetilde{p} \in [p, p']$. Therefore, we only have to search for a further witness starting at $p = p' + 1$. Since Algorithm 3 increases p by at least one in every step, the algorithm terminates after at most $|[L, U]|$ steps.

If the interval $[L'+1, U'-1]$ is not empty, $\forall p \in [L, L'] : \exists P = \langle x, \ldots, z \rangle_{\not\ni y} : w(P, p) \leq w(\langle x, y, z \rangle, p)$ holds. As we argued above, Algorithm 3 will perform at most $|[L, L']|$ steps to calculate the maximal possible L'. Furthermore, $\forall p \in [U', U] : \exists P = \langle x, \ldots, z \rangle_{\not\ni y} : w(P, p) \leq w(\langle x, y, z \rangle, p)$ holds. Therefore, the parameter decreasing witness search will also terminate after $|[U', U]|$ steps and find the minmal possible U'. Since Lemma 4.2 shows that $\forall p \in [L'+1, U'-1] : \forall P = \langle x, \ldots, z \rangle_{\not\ni y} : w(P, p) > w(\langle x, y, z \rangle, p)$, Algorithm 3 and Algorithm 4 find the maximal possible L' and minimal possible U'. □

Note that these algorithms require discrete parameter intervals. The algorithms base on the possibility to select a next parameter value. This can only be done for discrete parameter values. If we are only interested in finding witnesses for all parameter values p, Algorithm 3 would suffice. If we want to constrain shortcuts to the smallest possible interval, we also have to perform a parameter decreasing witness search starting at U. This is described in Algorithm 4. The restriction to the minimal necessity interval, as we do with Algorithm 5, has proven to be very useful. It helps to speed up the contraction process as well as the query in the contracted graphs, as we only focus on edges that are necessary for a chosen parameter p. During the contraction we only perform a witness search if the intersection of the respective necessity intervals is non-empty. If the necessity intervals do not intersect, there will never be a query that has to use both edges. Therefore, we do not have to search for a witness.

We use a Dijkstra algorithm for our local search with some minor modifications. We perform a one hop backwards search in combination with hop limits. The one hop backwards search allows us to prune the search on nodes whose distance, combined with the minimal distance from the one hop backward search, cannot provide a shorter path then we have already encountered. Also, we manage to reduce the hop limit for the forward search. To keep the processing time short enough we limit the search to a certain number of hops. For hop limits we select the staged hop limits Geisberger et al. described [GSSD08]. A detailed discussion on the influence of hop-limits for the local searches can be found in [Gei08].

To further reduce the amount of possible necessary edges, we perform a *parameter interval reduction*. Whenever we add a shortcut to the graph, we check if the new edge covers an existing edge on some part of the existing edges parameter interval. In this context we do not have to consider the new edges parameter interval, as the constraints induce an additional path with even smaller edge weight. If we cover the existing edge, we adjust its necessity interval.

4.3 Comparison to Related Work

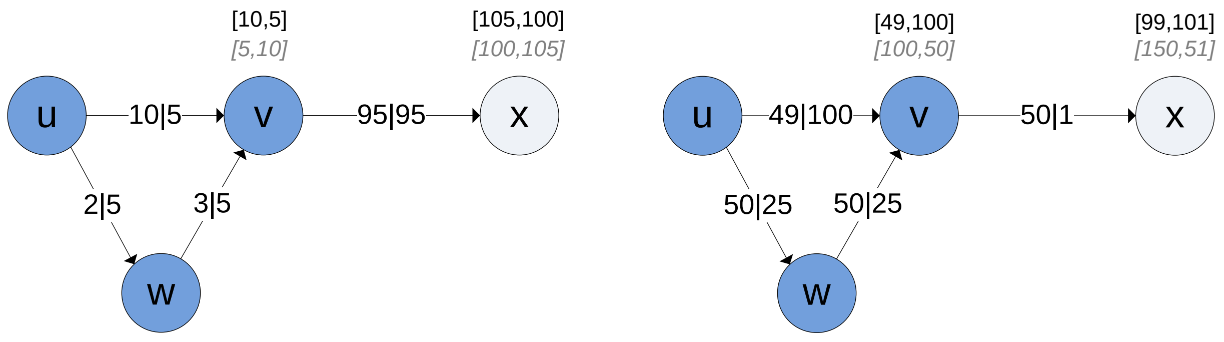
Recently, Delling and Wagner modified the algorithm SHARC to compute Pareto-optimal shortest paths [DW09]. In contrast to our approach, Delling and Wagner introduced a different method for calculating multi-criteria shortest paths [DW09].

They use a straight forward generalization of Dijkstra's algorithm to compute a Pareto-set $\rho(s,t)$. They maintain a list of labels at each node. For each step they extract the label with the smallest minimum component. Similarly to Dijkstra's algorithm, they calculate a tentative distance to all adjacent nodes. If the weight vector is not dominated, they insert it into the list of labels. By this method, the Dijkstras algorithm loses its label settling property. As we have already mentioned, it has been proven in that the worst case can result in a number of labels exponential in $|V|$ [Han79]. But even if the number of labels does not become exponential in $|V|$, the number of labels can rise too fast. Without any restrictions, even small graphs as the *Karlsruhe* road network will yield query times around 80 ms and a number of around 50 target labels with the full Pareto-SHARC. To overcome this problem, Delling and Wagner have to drastically reduce the number of possible labels. They introduce two new optimization thresholds ϵ and γ. ϵ limits the maximal length of the Pareto-paths. Let $L_t = (W, w_1, \ldots, w_{k-1})$ and W denote the time component of the vector; they say a vector L dominates another vector L' if $W \cdot (1+\epsilon) < W'$. In addition to the length limitation they introduce the factor γ. The factor γ introduces some kind of pricing. They only accept longer paths if the new path induces a significant improvement in the other parameters. They formulate this with the inequality $W/(W' \cdot \gamma) < \sum_{i=0}^{k-1} w_i' / \sum_{i=0}^{k-1} w_i$.

Both label reduction techniques fail to preserve the sub-path optimality criterion. Figure 4.4 gives examples for both domination criteria, where a label would be present at node x but the necessary sub-path is dominated at node v. We will exemplary describe Figure 4.4(b) in detail. The example for the ϵ domination criterion as depicted in Figure 4.4(a) can be described analogous. For a path $\langle u, \ldots, v \rangle$ we have two possibilities. Firstly, the direct edge $P = \langle u, v \rangle$, and secondly the path $P' := \langle u, w, v \rangle$. For simplification let γ be fixed at 1.0. At v the γ-condition evaluates as follows: $\frac{50}{100} > \frac{49}{100}$, which is true, and the Path P' will not be considered as it is dominated by the direct edge (u, v). At x the γ-condition for the concatenation of P' and (v, x), in comparison to the concatenation of P and (v, x), evaluates to $\frac{51}{101} > \frac{99}{150}$ which is false. Thus the alternative route via w should be considered at x. Since the algorithm itself relies on the subpath optimality, multi parameter SHARC is only a heuristic.

Furthermore, these restrictions have to be chosen very strictly to allow SHARC to precalculate continental size road networks. SHARC can only allow routes of up to a 5% longer traveling time for the western European road network to succeed with the preprocessing in a reasonable amount of time. Following the increased amount of labels, a Pareto-SHARC with a up to 5% longer traveling time will generate querys with an increased time of around 400 ms. For reasonable query times, Delling has to restrict the alternative routes to routes less than 1% longer than the shortest route. Only by turning to the combinations of the two heuristics, SHARC manages to process continental size road networks with routes up to 50% longer than the shortest route in terms of traveling time.

In our approach we only induce a single-criterion to constrain the number of possible paths. We restrict the number of valid parameters. This can roughly be compared to the epsilon threshold used



(a) Example for the failure of the ϵ restriction (b) Example for the failure of the γ restriction

Figure 4.4: Illustration of the heuristically characteristic of the SHARC label reduction techniques

in the SHARC approach by Delling and Wagner. Still, the restriction to a parameter interval $[0, p_{max}]$ will guarantee sub-path optimality and will therefore result in exact shortest paths.

In our opinion, the Pareto-approach suffers from some shortcomings and will therefore not succeed in supplying the flexibility we need today. As we already mentioned earlier, the Pareto-approach is often feasible on a small graphs, mostly derived from time table information [MW01]. On larger graphs the Pareto-approach can result in a number of labels exponential in $|V|$. Even if the number of labels will not be of exponential size, the number of labels that are Pareto-optimal will often grow too large to provide fast queries. Therefore, approaches that use Pareto-optimal paths usually have to introduce a lot of restrictions to reduce the number of possible labels. In contrast to the Pareto-approach we take a different route. Let $G = (V, E)$ be a graph. We add flexibility to the graph by modulating the edge weights as a linear combination of several weight terms.

The advantage of our approach, compared to the classical Pareto-approach, is that our graph becomes a single-criteria graph the moment we select a parameter p. By this way we successfully avoid the problem of multiple labels per node. Still, this advantage makes a comparison between the two approaches very difficult, as we only compute one possible path whereas SHARC computes all possible shortest Paths at once.

4.3.1 Gradual Parameter Splitting

Geisberger et al. already discussed the problem that Contraction Hierarchy is very sensitive in terms of node ordering changes [GSSD08]. The node order has to be selected very carefully to achieve the intended results. In our preliminary experiments we discovered that a node order for a large parameter interval will result in too many shortcuts. Due to the dense overlay graphs in the late phases of the contraction process, the witness searches in the top levels of our hierarchy will lead to preprocessing times too large for any practical use. As already mentioned in Section 2.1, we could perform our algorithm on multiple distinct parameter intervals. This would leave us with a set of different hierarchies to choose from. But we can do better; in our experiments we observed a lot of shortcuts to be necessary for all parameters. These shortcuts would be present in each of the distinct hierarchies and the respective contraction process would have to be repeated multiple times. It is obvious that for a lot of nodes some shortcuts will always be necessary. More than half the edges of our complete hierarchy proved to be necessary for all the distinct hierarchies we calculated. For example, the edges of the original graph will be present in nearly every hierarchy. Also, if for example a shortcut $\{x, (y), z\}$ is necessary for the whole parameter interval, the shortcut will be present in every hierarchy that contracts the node y before x and z. Especially in the lower levels of the hierarchy the number of edges that are necessary for the whole parameter interval was very large. Most prominent reasons are dead-end streets or inner city areas where we have the same average speed on every road. For example on the American road network, where we have a lot of similar roads in the inner city areas, a lot of roads should result in nearly identical cost functions. In those areas we should get much less alternative routes and therefore more shortcuts that are valid on the whole parameter interval.

In our algorithm we try to avoid unnecessary work as much as possible. For this, we use a classical divide and conquer approach. We select a threshold t as criterion for a possible split. This threshold may be compared against the degree of the graph or against some other criterion. We contract the graph for the whole parameter interval $[L, U]$ until the threshold mechanism triggers. Then we divide the set of parameters in distinct parts $[L, M], [M + 1, U]$ and perform the further contraction only for the respective parameter intervals. This method can then recursively split the single parameter interval, when another split may seem advisable. Since the node ordering process is so sensitive, our method may still result in a suboptimal hierarchy for the parameter intervals but will prevent a lot of work from being performed multiple times. Our experiments showed that we can contract a large part of the graph for the full parameter interval $[0, p_{max}]$. This is partly a result of the dead-end streets and inner city areas or similar other criterias we mentioned above, since these areas should give the same shortest paths for all parameters. Only if we allow a lot of traveling speed changes, the parameter becomes interesting.

All in all it seems sufficient to split at a relatively high level. Even though we have only a small number of nodes left compared to the first part that we contracted for the whole parameter interval at once, the small amount of node levels still seemed to leave enough room for the node order to adapt. An indicator for this is that the top level intervals of a splitted hierarchy will only provide a low number of common nodes. Note that we perform recursive splitting for each parameter interval independently. Due to the diverse and specific needs for the different parameter values, we can split at different hierarchy levels to adapt to the specific “difficulty” of a parameter interval. This adapts to the needs that varying parameter values will lead to distinct numbers of shortcuts in different parameter intervals. Especially in parameter intervals that provide a large variety of possible routes we need to generate more individual node orders.

To limit the amount of main memory we need during the contraction process we utilize some disk space too. Whenever we split at a certain level, we write the delta of our last contraction phase to disk. When we need to reset the state of our contraction we can load the current overlay graph back from disk. This saves a lot of main memory without slowing down the contraction process too much. If we decide to provide more main memory, we could easily avoid this disk usage. For a contraction process, the threshold is a tuning parameter. The lower we choose the threshold the more our algorithm converges against a set of single-criteria CH. Note that we also adapted the threshold for different parameter values. This is needed, since we have to predict the future of our contraction process. Even though the contraction could be working fine at a given moment we could generate too “difficult” overlay graphs. Therefore, the threshold needs to be chosen very carefully. Still, we want to minimize the work performed multiple times and may therefore adapt the threshold on parameter intervals we expect to be less “difficult”.

Splitting Approaches

We can choose several criteria for triggering a split. We have to decide on the moment of splitting and on how to divide the given parameter interval. Note that the number of shortcuts is not directly related to the size of the parameter interval. The complexity is more dependent on the variety of routes between two given nodes that a parameter interval contains. A more detailed discussion of this can be found in the experimental section. Possible criteria are the average degree of the graph or the number of edges that are not necessary for the whole parameter interval, which we call *partial edges*. In our experiments the number of *partial edges* proved to be a viable method, because it is a good indicator for how far the respective parameter values veer away from a single-criteria CH. Note that a single-criteria CH will keep the number of edges small. The further we get from a single-criteria CH for a given parameter, the more edges we will add for the respective parameters. But even a single-criteria CH will result in very dense graphs in the top levels of the hierarchy. Therefore, the degree of the graph may not be an indicator as good as the number of partial edges. Thus we decided to use the number of partial edges as a splitting criterion. Note that this method is very simple. We could possibly get even better results if we also look at the kind of partial edges we create. Edges that are necessary for nearly the complete parameter interval may not have an impact as strong as edges that are only necessary for a very small parameter interval. For the question on how to perform the split we propose two different methods.

- split in half: the direct and most simple version is to split the parameter interval in half. This has shown to be a stable approach. But it might result in slower queries. If the relative traveling time and energy cost values do not change in a similar fashion over the whole parameter interval, we may get better results if we take the varying "difficulties" into account. We might end up choosing to large intervals. Also, it might end up in splitting to similar parts of the parameter interval.

- variable-split: instead of just splitting the parameter interval in half, the split can be performed again taking into account the number of edges that are only necessary on a part of the parameter interval. We first calculate the number of edges that become valid at a given parameter. We split at the parameter that allows more than half of the edges to become necessary. During our experiments such a parameter always existed. This can be explained through the property of the cost function. Most shortcuts become necessary at a certain parameter and will remain necessary. If we cannot find such a splitting parameter on a certain data set, we perform a split as in the split in half approach. Still, our experiments showed that this situation does not occur for our chosen weight functions.

In the following we will refer to those hierarchy parts for a parameter interval we did not split as *top level intervals*.

4.4 Goal Direction for Incompletely Contracted Hierarchies

Experiments have shown that the number of edges that are created will grow very fast in the upper levels of the hierarchy. This will result in extremely long preprocessing times for a complete hierarchy. To cope with this, we may choose to perform the contraction only up to a certain level L. The remaining $|N| - L$ nodes form the so-called core of the hierarchy. Our experiments showed that we can thus save a significant amount of preprocessing time. Note that this effect grows especially strong, since we perform the contraction of the highest levels of the hierarchy more often because of the splitting process. Therefore, the already more "difficult" contraction phases have to be calculated much more often than those at lower levels. If we choose not to contract each of the top level intervals of the splitting approach, we can save a significant amount of preprocessing time. Due to the splitting process this generates a core for every top level interval. On each of theses cores we can utilize other speedup-techniques as shown by Schieferdecker in [Sch08a]. In the following we will now discuss how we can utilize the ALT algorithm for our contraction process.

ALT for Linear Objective Functions

We present two different methods to adapt the ALT algorithm for the bi-criteria case. For the preprocessing we use an implementation for single-criteria ALT processing provided by Daniel Delling. We therefore extract every core twice with different weights.

Traveling Time and Energy Cost Core

In analogy to the linear weight function we utilize for our graph we can deduce lower bounds and therefore potential functions as explained above. To utilize the ALT algorithm for our linear weight functions we use multiple inequalities. We generate landmarks and the respectively landmark distances both with regard to the time (τ) and the cost (κ). For each of the weight terms we can then calculate a lower bound on the distances $d(s,u)$ and $d(u,t)$ respectively. In the following, we will denote with $d_\tau(s,u)$ the lower bound in the core using time weights and with $d_\kappa(s,u)$ the lower bound in the core using cost weights. The bounds for the target node we name in the same way. With those lower bounds we can then calculate a lower bound for a given parameter p in the same way we calculate the edge weight function itself. Let $d(s,u,p)_\tau$ denote the time component of the distance from s to u for the parameter p and $d(s,u,p)_\kappa$ denote the energy cost component respectively; for all parameters p the following inequality holds: $d(s,u,p)_\tau \geq d_\tau(s,u)$ and $d(s,u,p)_\kappa \geq p \cdot d_\kappa(s,u)$. Therefore, we can approximate $d(s,u,p) \geq d_\tau(s,u) + p \cdot d_\kappa(s,u)$.

Lower Bound and Upper Bound Core

As an alternative method we propose the *lower and upper bound core*. As we have shown in Lemma 4.1, the linear edge weights form a concave function. The straight line between $w(p_L)$ and $w(p_U)$ on an interval $[L,U]$ will provide a lower bound for each parameter $p \in [L,U]$. We can therefore approximate the minimum distance for a given parameter with a linear combination of the lower bound distances and the upper bound distances for a given core. Let $C[L,U]$ be the core for the top level interval between L and U of our splitting approach. Let us denote the approximations in the core using the lower bound weights and the upper bound weights with $d_L(s,u)$ and $d_U(s,u)$ respectively. For any given parameter $p \in [L,U]$ and any two nodes $s,u \in C[L,U]$ we can now approximate with Lemma 4.1 $d(s,u,p) \geq (1-\frac{p-L}{U-L}) \cdot d_L(s,u) + \frac{p-L}{U-L} \cdot d_U(s,u)$.

Due to this combination of the potential functions, we can use one pair of time and cost landmarks for a given core instead of calculating the landmarks on a per parameter basis. This comes at the cost of some quality, since we find a lower bound on a lower bound. Still, it has proven a viable method and helps to improve the quality of the core search. Especially in the lower and upper bound approach the landmarks give correct distances for the parameters L and U for every core $C[L,U]$. As already discussed in Section 3 we may need to utilize proxy nodes for source and target combinations that are not contained in the core for a given parameter. Note that for different set of parameters we may or may not need proxy nodes, since the different node orders also generate different cores. During a CALT search we use a single proxy node for the search that has the minimum combined distance from the core to the source node. By this method we avoid the need to calculate landmarks and core graphs for every parameter. We only have to compute one core with two different weight functions per top level interval. Since our top level intervals contain up to 80 parameters, we save up to a factor of 40 in space and preprocessing time. This comes at the cost of multiple quality penalties on the potential functions. Still, our experiments showed that we can improve the query times of a core approach with the potential functions presented here.

4.5 MCCH Query

In this section we will first describe the differences between a multi-criteria Contraction Hierarchy query and a single-criteria CH query, including a more detailed illustration of the core-based query. Then we will proof the correctness of the algorithm and present some techniques to speed up the Query.

4.5.1 MCCH compared to Single-Criteria CH

For a completely contracted MCCH and a given parameter, the MCCH query is identical to a CH query. For a given parameter we can evaluate the weight function, and thus the graph can be looked at like a graph with a constant weight function. Therefore, we can perform a standard Contraction Hierarchy search if we evaluate the edge functions as needed. The only difference to the standard query is that not all edges stored are relevant for the current search. Edges that are not necessary for the given parameter do not have to be considered in the search.

Core Query. We perform the core approach, as Bauer describes it, in two phases [BDS+08]. In the first phase we perform a bidirectional CH query in the forward and the backward graph respectively stopping at uncontracted nodes. To identify these nodes we can store a flag for each top level interval of the split up hierarchy. This we call the first phase of the search. If we could not finish the search during the first phase, we move on to the second phase. In the second phase we perform a core based bidirectional ALT algorithm. We split the second phase itself into multiple parts. The first phase is the calculation of the proxy nodes. To find these we have to perform another search upwards in the hierarchy. We perform a backward search from the source node and a forward search from the target node until the first core node is settled. Note that it is not guaranteed to find a path to the core in the respective searches. If that is the case we use a special node as proxy node with the distance form and to all landmarks as well as the proxy distance set to zero. This can reduce the search to a simple bidirectional Dijkstra's algorithm if neither a source proxy nor a target proxy can be found. After the calculation of the proxy nodes we use a bi-direction ALT algorithm with the consistent approach [IHI+94].

4.5.2 Correctness Proof

During our query we utilize the single-criteria Contraction Hierarchy query on our multi-criteria Contraction Hierarchy. In contrast to the single-criteria CH query we only use a subset of the graphs edges. An edge belongs to the search graph if its necessity interval contains the current parameter p. The proof that our algorithm performs correct shortest path queries therefore bases on the correctness of the single-criteria CH query algorithm. A detailed proof of the correctness of the single-criteria CH query can be found in [Gei08].

Theorem 4.4 *For every parameter a single-criteria CH query can find a shortest path in a MCCH.*

To prove Theorem 4.4 we first have to prove two other lemmas:

Lemma 4.5 *For every path $P = \langle u, \dots, v \rangle$ that is valid on an interval $[a, b]$ we can find for every parameter $p \in [a, b]$ a path $P' := \langle u, u_1, \dots, u_h, v \rangle$ with $\forall i \in [1, h] : \mathcal{L}(u_i) \geq \min(\mathcal{L}(u), \mathcal{L}(v))$ and $w(P', p) \leq w(P, p)$ with $\mathcal{L}$ denoting the level of a node as defined in Section 3.4.2*

Proof. Let $p \in [a, b]$. Assume that we have a path $P = \langle u = u_0, u_1, \ldots, u_h, u_{h+1} = v\rangle$ with $\exists u_i$: $\mathcal{L}(u_i) < \min(\mathcal{L}(u), \mathcal{L}(v))$. Pick $\widetilde{u}$ with $\forall i \in [1, h] : \mathcal{L}(\widetilde{u}) \leq \mathcal{L}(u_i)$. Since $\mathcal{L}(\widetilde{u})$ is minimal over all u_i, $\widetilde{u}$ was contracted before its direct neighbors x and y. Therefore, $\forall p \in [a, b] : \exists \widetilde{P} = \langle x, \ldots, y\rangle_{\not\ni \widetilde{u}}$: $w(\widetilde{P}, p) \leq w(P, p) \wedge \forall y \in \widetilde{P} : \mathcal{L}(y) > \mathcal{L}(\widetilde{u})$. This holds, as we insert a shortcut for every $p \in [a, b]$ for which we cannot find a path $\widetilde{P}$ as described. Thus, we can either replace $\widetilde{u}$ with a path consisting out of nodes with higher level then $\widetilde{u}$ or omit it in the path if we had to insert a shortcut for the respective parameter. Since the number of levels is limited, we can construct the desired path P' in a limited number of steps. □

Lemma 4.6 *For every edge (u, v) that is restricted to a parameter interval $[a, b]$ and a parameter $p \notin [a, b]$, we can find a path $P = \langle u, \ldots, v\rangle$ in the graph created by our algorithm with $w(P, p) \leq w((u, v), p)$ and $\forall x \in P : \mathcal{L}(x) \geq \min(\mathcal{L}(u), \mathcal{L}(v))$.*

Proof. Let $e = (u, v)$ be an edge restricted to an interval $[a, b]$ and $p \notin [a, b]$. Furthermore, let $|\cdot|$ denote the number of edges of the original graph a given edge represents. First let us assume that $|e| = 1$. Since the edge is an edge of the original graph and is restricted to an interval following holds: $\forall p \in [0, a - 1] \cap [b + 1, U] : \exists P = \langle u, \ldots, v\rangle : w(P, p) \leq w(e, p)$. This is the case, since we only restrict edges of the original graph with parameter interval reduction. Therefore, those alternative paths must exist for every parameter p in the respective intervals. Due to Lemma 4.5 we can replace the path P with a path P' that fulfills $\forall x \in P' : \mathcal{L}(x) \geq \min(\mathcal{L}(u), \mathcal{L}(v))$. This forms the basis for our induction step.

Now let us assume $|e| = N$ with $N > 1$. Since $N > 1$ the edge is created as a shortcut from two edges e_{in} and e_{out} that are restricted to the intervals $[a_{in}, b_{in}]$ and $[a_{out}, b_{out}]$ with $[a_{in}, b_{in}] \cap [a_{out}, b_{out}] \supseteq [a, b]$. Now $|e_{in}| < N$ and $|e_{out}| < N$ holds. With induction we can replace both edges with a path P_{in} and P_{out} as described above. If $p \in [a_{in}, b_{in}]$ or $p \in [a_{out}, b_{out}]$ we can directly use e_{in} or e_{out}. The concatenation of P_{in} and P_{out} therefore gives a path of the desired form for p. □

Proof (of Theorem 4.4). Let $((G = (V, E), <))$ be a multi-criteria Contraction Hierarchy. Let $s, t \in V$ be a *source* and a *target* node and let p be the desired parameter. By the definition of a shortcut, it represents an existing path in the graph. The single-criteria CH query finds only paths of the form

$$\langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle$$

$$u_i < u_{i+1} \text{ for } i \in \mathbb{N}, i < p, \text{ and } u_j > u_{j+1} \text{ for } j \in \mathbb{N}, p \leq j < q$$

Let $p \in [0, p_{max}]$. Let $P = \langle s = u_0, u_1, \ldots, u_p, \ldots, u_q = t\rangle$ be an aribtrary path that is not of the described form. Therefore $\exists u_k, k \in [1, q] : \mathcal{L}(u_{k-1}) > \mathcal{L}(u_k) < \mathcal{L}(u_{k+1})$. Let the edge (u_{k-1}, u_k) be restricted to the interval $[a_{in}, b_{in}]$ and the edge (u_k, u_{k+1}) be restricted to the interval $[a_{out}, b_{out}]$. If $p \notin [a_{in}, b_{in}]$, Lemma 4.6 implies that we can subsitute the edge with a path P' that is valid for p and consists of nodes with higher or equal levels. In the same way we can find a path P'' if $p \notin [a_{out}, b_{out}]$. With the use of Lemma 4.5 we can replace the concatenation of P' and P'' with a further path P''' that consists of nodes with higher levels than u_k.

Let $M_P := \{u_k | \mathcal{L}(u_{k-1}) > \mathcal{L}(u_k) < \mathcal{L}(u_{k+1})\}$. We can now construct a path $\widetilde{P}$ for every p, with $M_{\widetilde{P}} = \emptyset$ and $w(\widetilde{P}, p) \leq w(P, p)$. For this we recursively select the node u_m with $\forall u_j \in M_P$: $\mathcal{L}(u_m) \leq \mathcal{L}(u_j)$. As explained above we can replace the sub-path (u_{m-1}, u_m, u_{m+1}) with a path consisting of higher level nodes. Since we have a limited set of levels this algorithm will terminate. Thus, the Contraction Hierarchy query can find a path of equal or less weight for every arbitrary source-target-pair and every parameter p. □

4.5.3 Dealing with Split Hierarchies

One of the problems we face during a contraction is that we cannot find a node order that will result in a viable hierarchy for a large parameter interval. One of the methods we propose is the divide and conquer approach. This will result in multiple node orders for different levels in the hierarchy, including multiple sets of edges. Therefore, we have to augment the definitions of the forward and the backward graphs. The forward graph of a Multi Criteria Contraction Hierarchy contains all edges that are directed upwards in a hierarchy for a given parameter. The backward graph will contain all edges that are directed downwards in a hierarchy for a given parameter. During a query we can recognize the edges not directed upwards for the given parameter by looking at its necessity interval. To save memory, we can combine all edges that are identical in all aspects except the necessity interval. If the two intervals of those identical edges lie next to each other, we can only insert one edge with the combined necessity interval. These edges can be found with a simple sort of all edges of a given source node. The split also will lead to multiple cores. Therefore, we have to store multiple sets of landmarks and landmark distances. According to the number of splits that occur during the divide and conquer approach we have to store a bit for every node, indicating if it is in the core of the respective parameter interval.

If we use the core approach together with the splitting approach, we have to store multiple sets of landmarks and distances. To recognize core nodes we can store a set of flags in the nodes of the graph indicating if the node is in the core of the respective hierarchy. We can then choose the appropriate set of landmarks and respective distances and perform the ALT search.

For the landmark variant of our search we want direct access to the landmarks of a node. For this we need to map the core nodes to the numbers 0 to $|core| - 1$. Let $\mathcal{M}_\Theta(\cdot)$ denote the mapping of a node in the top level interval Θ. We decided to store the mapping directly in the respective node. Since the cores of the respective top level interval only have a few common nodes, we manage to map the nodes of each core to the numbers 0 to $|core| - 1$ in such a way that following condition holds:

$$\forall u : u \in \Theta \wedge u \in \Theta' \Rightarrow \mathcal{M}_\Theta(u) = \mathcal{M}_\Theta(u).$$

Due to the small number of common nodes between two top level intervals, we can find such a mapping with a simple greedy approach. We sort the nodes by the number of top level intervals a node belongs to and process the mappings in the resulting order. For every node we select the first number that is not used in any of the containing cores. If we only have nodes left that belong to one of the intervals, we can map these nodes directly to the ids left for the respective core. If we run out of free ids during the mapping process of the common nodes, we map to ids larger then $|core| - 1$. This would lead to holes in the storage space for our landmark distances. Still, this case did not occur during our experiments and we were always able to map our nodes directly into the interval $[0, |core| - 1]$. Even if this holes would occure, we expect the number to be very small. Therefore, the holes would not harm the perfomance of our algorithm.

4.5.4 Speeding up Multi-Criteria Contraciton Hierarchy Queries

Due to the flexibility and the splitting approach we generate a lot of edges that are only necessary for a small fraction of the allowed parameters. Without compromising the correctness, we could use all those edges in our search. However, our experiments showed that we would use an unnecessarily large amount of edges, which we do not need to find a shortest path. We could also climb downwards in the hierarchy of our current parameter when we use an edge from a different node order. Since a single-criteria CH generates its high speedups through the upwards-only searches, we would severely increase our query times if we would use all unnecessary edges. Therefore, we have to skip all edges that are not necessary for our current parameter value. Since we have to skip a lot of edges, we use a bucketing approach. Each bucket will represent a smaller parameter interval, i.e., one of the top level intervals of our hierarchy. For every bucket, we store an *edge id array*, containing the edge ids of edges in the edge array. This extra array introduces another level of indirection. For every node wee store, analogous to an adjacency array, offsets into the edge id array. In our experiments we found a lot of edges belonging to all buckets. These edges result in a lot of edge ids that have to be stored for each bucket. Especially edges of the original graph would be present in all buckets and necessary for nearly all parameters. For these edges we augment our bucket approach even further. We add an additional offset to every node that points into a second edge array, the *global edge array*. This offset we use exactly as a normal adjacency array for the second array. This results in arbitrary edge ids. A single id could belong to the indirectly accessed edges or the edge array that we can access without the use of a further level of indirection. To circumvent this problem, we use the sign bit of the edge ids. An edge-id with a negative sign will belong to the bucket edges and an edge-id with a positive sign will belong to the global edge array. This additional edge array saves a lot of main memory. Since more than half of the edges belong to all buckets, we invest $4 * n$ bytes for the extra offset in every node to save $4 * |Buckets| \cdot m/2$ bytes of unnecessary edge ids in the edge id arrays. In the appendix we present a graphical illustration of our bucketing approach for a sample graph.

Chapter 5

Experiments

Test Instances. We present experiments performed on road networks from the year 2006 provided by the PTV AG. The German road network consists of 4 692 751 nodes and 21 612 382 edges. The European road network consists of 29 764 455 nodes and 135 315 556 edges. For some preliminary experiments we also used some of the smaller countries of Europe.

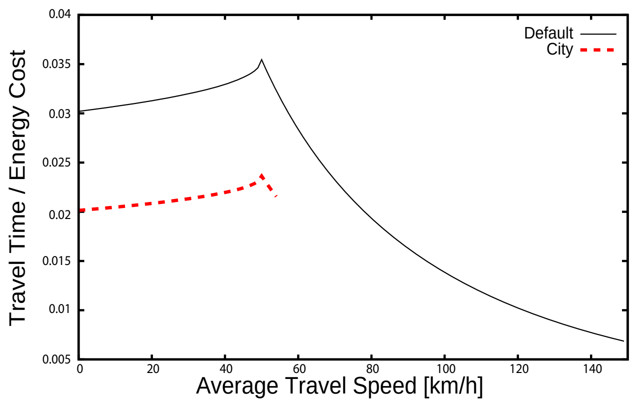
Environment. Experiments have been done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2×1 MB L2 cache. Only the contraction of the European road network has been done on one core of an Intel Xeon 5345 processor clocked at 2.33 GHz with 16 GB main memory and 2×4 MB L2 cache. We run SuSE Linux 11.1 (kernel 2.6.27) and use the GNU C++ compiler 4.3.2 with optimization level 3.

In our experiments we will first focus on the contraction times and query times of the completely contracted hierarchies. Then we will go on to discuss the CALT version of the hierarchy.

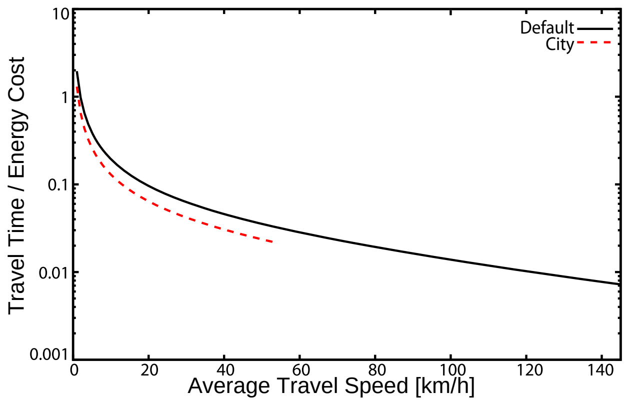
Methodology. For our experiments we use a number of $1\,000$ queries for every parameter value p of the respective parameter interval. If we do not specify a parameter interval explicitly, we use the parameter interval $[0, 1\,024]$. For these queries we choose a source and target pair uniformly at random. We will present *preproc*essing times in the format $[hh : mm.ss]$. We may split the preprocessing times in compute and I/O-time. For the CALT algorithm we will present average preprocessing times on a per *top-level*-basis. Query times will be given in milliseconds $[ms]$ and the respective speedup as a comparison to a standard Dijkstra algorithm. Space consumption will be given absolutely in $[B/node]$. If not specified we use one bucket for every top level interval of our hierarchy. For split methods we distinguish between the **H**alf split and the **V**ariable split method. We may use **G** and **E** as an abbreviation for the German and the European road network respectively. If we use a core approach, we give the number of nodes in the core and indicate if the core is **C**ontracted or **U**ncontracted. For the landmark selection we distinguish between the **A**void and the **M**ax-cover algorithm.

5.1 Experimental Preliminaries

Weight functions. For our experiments we used a linear combination of the travel time and the approximated energy cost for traversing a road. We synthesized a cost function through the approximated mechanical work for a standard car under ideal conditions. To generate the energy costs of a road we utilize a combination of the standard formula for air resistance $F_w(v) := A \cdot c_w \cdot \rho/2 \cdot v^2$ and the roll resistance $F_r = d/R \cdot F_N$, where A denotes the projected plane in square meters, c_w the drag



(a) Illustration of the relation between the travel times and the costs generated by $C(v,l)$ for the different traveling speeds.



(b) Illustration of the relation between the travel times and the costs generated by $\widetilde{C}(v,l)$ for the different traveling speeds.

Figure 5.1: The two different cost functions

coefficient, ρ the air density, v the average traveling speed of the road, d/R the roll resistance coefficient and F_N the weight of the car in Newton. We use $A = 2.67m^2$, $F_N = 1\,500kg$ and $d/R = 0.015$ as well as $c_w = 0.3$. These values are approximated to represent a common car as the Audi A3. For ρ we use the air density for 21 degrees Celsius $\rho = 1.2kg/m^3$.

We can now define a cost function $C(v,l) = l \cdot c \cdot (F_w(v) + F_r)/\eta$, with l denoting the length of the road, v the average traveling speed, c the cost of energy and η the efficiency of the combustion engine and drive train. We use $c = 0.041€/\text{MJ}$, which corresponds to a price of about $1.42€/l$ and $\eta = 0.25$.

Since the only parameter of the roads is the average traveling speed we would result end up favoring the roads with minimal average traveling speed. This effect would be even greater since the speed is a quadratic parameter. Preliminary experiments with a standard car showed that the optimal traveling speed was at around $50km/h$. We adapt our cost function to this by defining a new cost function $\widetilde{C}(v,l)$ as follows:

$$\widetilde{C}(v,l) = \left\{ \begin{array}{ll} C(50 + \sqrt{50 - v}, l) & \text{if } v < 50 \\ C(v,l) & \text{otherwise} \end{array} \right\}$$

Finally we adapt our cost function $\widetilde{C}(v,l)$ for inner city roads. Inner city traveling introduces a lot of stop-and-go situations through traffic lights, right-of-way rules or drivers backing up into a parking space. Therefore, it seemed reasonable to use higher cost for an inner city road compared to the cost of an outer city road. We compensate for this by multiplying $\widetilde{C}(v,l)$ by a factor of 1.5 on inner city roads.

In Figure 5.1(a) we can see how the travel time of an edge compares to the cost of an edge as provided by $C(v,l)$ for the different average speeds. In comparison Figure 5.1(b) shows the relations between the travel time and the cost provided by $\widetilde{C}(v,l)$. The $\widetilde{C}(v,l)$ is more realistic since we will not favor the slowest possible route. Still, our experiments showed that our algorithm can even process cost functions that will favor the slowest possible route.

Parameter Interval. As we described in Section 4.2.3 we use a parameter-increasing witness path search in our contraction process. The algorithm's correctness relies on the ability to select the next parameter value. To achieve this we utilize integer values for our traveling time and energy cost values. This not only allows us to use Algorithm 3 but also gives us calculations that do not rely on floating point accuracy. Our experiments showed that we only need relatively small parameter values for a relatively large variety of possible routes. Preliminary experiments revealed that a for our cost function only parameters $p \in [0, 0.1]$ proved promising. For a parameter of 0.1 we will already allow routes of 65% longer traveling time on average compared to the optimal traveling time. To still allow a fine granular selection for integer parameters we scale the time values by a factor of 100. In addition we divide the cost values by 100, which results in an effective scale of $10\,000$ and allows $1\,000$ parameters in the interesting interval. Since we already use an approximation for our cost term, we can perform this division. The values would only suggest accuracy that we cannot provide with our cost function.

In Figure 5.3(a) we illustrate the average relations for travel time and energy cost values over all parameters. As can be seen the strongest variation takes place for $p \leq 600$. Figure 5.3(a) illustrates a graphical overview of the changes in the cost and time component for all processed parameter values. At $p = 1\,000$ we are on average only 1% off the optimal energy cost value. Also, the relative improval over all parameter values gets too small compared to the lengthening of the traveling time. Therefore, the parameter interval is sufficient for our calculations. Unless otherwise stated, we use an interval size of $1\,024$ to generate equal-sized intervals for the half-split method.

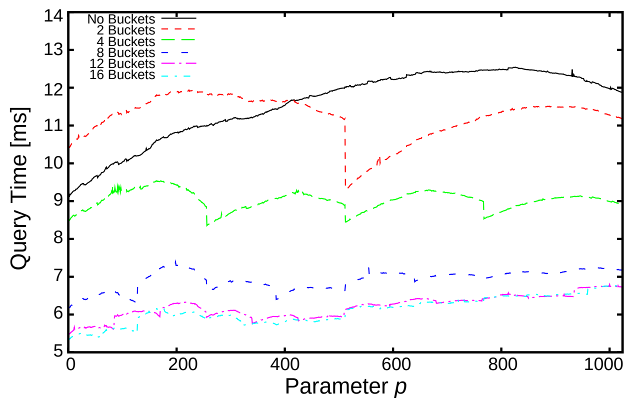
To illustrate the robustness of our algorithm we will also present some results on parameter intervals of the size of 24, 512 and $2\,048$.

The step size of our algorithm is one important tuning parameter. Figure 5.3(b) illustrates the number of different routes we get with three different step sizes. As the figure shows we only omit a low number of paths if we lower the resolution of the parameter interval. But our experiments also revealed that we need an only 2% longer preprocessing time but get 10% more possible routes. Compared to Pareto-SHARC that has to limit the number of possible labels to around 5 with heuristics, we manage to generate a more diverse number of routes while still maintaining very reasonable preprocessing times. Despite those results we expect a higher speedup if we increase the step size very drastically.

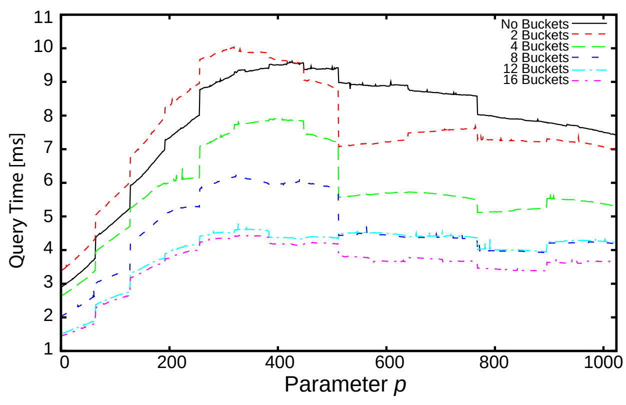
5.2 Bucketing

Due to the restriction of edges to the smallest parameter interval for which they are necessary we have to scan a lot of unnecessary edges. To prevent this we have introduced a bucketing structure in Section 4.5.4. The number of edges stored for every node is high enough to gain advantage from a bucketing structure. Figure 5.2(a) illustrates the positive effects and the drawbacks of the bucketing approach for a hierarchy contracted for all parameters at once. In Figure 5.2(b) we depict the same situation for a hierarchy contracted with our splitting approach. As the figure shows, we even get a minor increase in query times for some part of the parameter interval with the low number of two buckets. This is a result of the additional level of indirection that we add to the graph. The positive effect comes into effect if we add more buckets. For the number of four buckets we already exceed the query times of the no-bucket version on the whole parameter interval. For a number of eight buckets we manage to reduce the query times by nearly 50%. A further increase in the number of buckets yields only a small improvement in query times. For the split contraction we can see the same effect. We also get the reduction of the query time for a high number of buckets of around 50%. Especially in the upper half of the parameter interval we see a good improvement, even for a low number of buckets.

The effect of the increased space consumption can be lowered severely by the separation into the bucket edges and the directly accessed edges. Especially for a parameter interval contracted as a whole the percentage of edges that belong to all buckets is incredibly high. Since we need a high

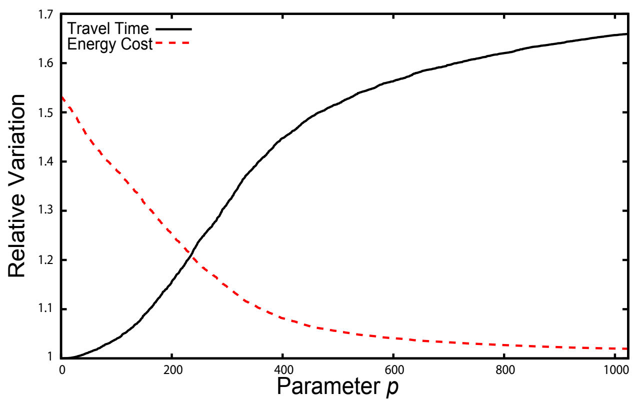


(a) Contraction without splitting.

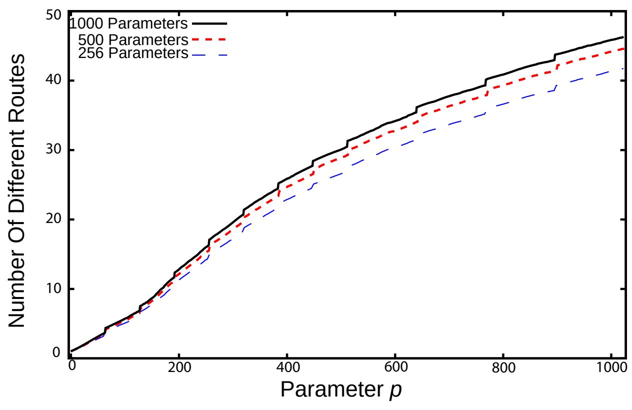


(b) Contraction with the half-split method.

Figure 5.2: Illustration of query times for different number of buckets with equally-sized bucket intervals for the German road network.



(a) Illustration of the relative variations of the travel time and the travel cost for the German road network



(b) Illustration of the number of different routes we encounter up to a certain parameter for the European road network. The smaller parameter intervals are scaled by a factor of 2 and 4 respectivly to represent the same interval.

Figure 5.3: Parameter Interval Illustrations

amount of buckets to get the positive effect in full, we would end up storing to much data in all buckets for the method to stay reasonable. The separation of edges into the bucket edges and the directly accessed edges instead has proven to be very space-efficient. For the gradual parameter split the bucketing method becomes even more valuable. Due to the number of different node orders the number of edges valid for only a small number of parameter values p is considerably larger.

Note that the bucketing itself has some drawbacks. The added level of indirection reduces the possible speedup of approach severely. If we used our algorithm to calculate a separate hierarchy for every single bucket, we would reduce the query times by about 40% at the cost of larger memory consumption and longer preprocessing time. This would lead to a very large amount of space overhead, which we hope to avoid. In the following we will show that we can get decent query times even with a single hierarchy.

5.3 Completely Contracted Hierarchies

As already mentioned, our preliminary experiments revealed that the contraction process would take too long for a complete hierarchy for all parameters. In Table 5.1 we present a number of different size graphs with complete contractions without splitting. As the table shows, the preprocessing times will rise super-linearly in the size of the graph. When the size of the input graph exceeds a certain size the preprocessing times will suffer from a combinatorical explosion as we get in the Pareto-approaches. We observed that the number of edges will get too large on higher levels for a complete contraction over the whole parameter interval. As Table 5.1 shows, the contraction time rises extremly fast between Portugal and Italy. The difference from Italy to Germany is even more drastic. A 24% increase in the number of nodes yields a 500% increase of preprocessing time. As Geisberger et al. discussed the Contraction Hierarchy method is very sensible in terms of node ordering [GSSD08]. The node order is designed to keep the number of edges in the top level intervals small while also providing fast queries. If we allow flexibility, the optimal node orders for different parameters will be very diverse. Geisberger showed this behaviour in [Gei08]. As a result of this we get a very suboptimal node order for a complete contraction over all parameters, since we have to account for a lot of different needs. This leads to a more dense series of overlay graphs. This effect accumulates throughout the contraction process to finally result in extremely long preprocessing times.

Table 5.1: Preprocessing and query performances for different *graph*s of different node numbers.

	nodes	param	preproc	space	query	speed	nodes	edges
graph			[hh:mm]	[B/node]	[ms]	up	settled	relaxed
Switzerland	693 122	1 024	0:05	90	0.7	270	262	1 841
Portugal	1 244 546	1 024	0:14	91	1.0	350	337	2 454
Italy	3 757 721	1 024	2:14	97	2.2	529	445	4 183
Germany	4 692 751	1 024	9:52	99	7.8	260	823	9 731

5.4 Gradual Parameter Splitting

Our experiments showed that we cannot rely on a single complete contraction for our whole parameter interval. To circumvent this we proposed the method of gradual parameter interval splitting. In Section 4.3.1 we discussed two possible splitting criteria. In Table 5.3 we illustrate amongst other things a contraction for the parameter interval $[0, 24]$ for the European road network, which roughly represents an increase of 2.5% in travel time. A comparision of the contraction for the interval $[0, 24]$, to the contraction of the complete interval Table 5.3 reveals that we would require a factor of five longer preprocessing times when using our algorithm for a number of 43 parallel hierarchies. This time may be reduced if we use longer intervals, as long as we do not make the intervals too large.

If we even choose to rely on a number of parallel Contraction Hierarchies, we would need 1024 parallel Contraction Hierarchies. We performed a number of distinct contractions for fixed parameters and in advance evaluated cost functions with a single-criteria CH.

We computed the single-criteria Contraction Hierarchies for the parameters $p = 0$, $p = 1000$ and for $p = 300$, which has shown to be one of the most difficult parameters from Figure 5.5(a). The results can be found in Table 5.2. The results showed a large variation between the different parameter values. The preprocessing time varied by about 100% and the query time even by 270%. Only the space consumption appeared to be quite constant with a variation of less than 3% and around 22 B/node.

Table 5.2: Single-criteria Contraction Hierarchy performance for our preevaluated weight function on the German road network

method	parameter	preproc	space	query
		[s]	[B/node]	[ms]
aggressive	0	335	20.3	0.179
economical	0	112	21.9	0.249
aggressive	300	616	20.9	0.734
economical	300	225	22.1	0.919
aggressive	1 000	589	21.1	0.693
economical	1 000	228	22.2	0.927

About the same results can be seen in Figure 5.5(a) for the query times of our multi-criteria Contraction Hierarchy. Still, one cannot compare the results of a single-criteria CH directly to our multi-criteria CH. Not only the strong dependency in terms of the edge weight function hinders any comparison, but also the additional factors we get from edge weight evaluation. Also, we need to store the necessity interval as well as two edge weight terms for every edge as well as additional data for our bucketing approach resulting in more cache faults. In addition to these effects, the bucketing method requires an additional level of indirection. If we chose to compare our algorithm to the single-criteria CH, we need a factor of 8.4 more space. We could therefore drastically reduce the needed space in comparison to the $1\,024$ parallel contraction hierarchies we would need for the same results. This even reduces the space overhead if we only compute a single-criteria contraction hierarchy for every one of our top level intervals. In terms of preprocessing time we also manage to reduce the time in comparison to $1\,024$ different parallel Contraction Hierarchies. Still, our results are not as good as for the space consumption. We could manage to calculate a number of 21 parallel single-criteria CH with the economical method [GSSD08] in the time it takes to preprocess a core of the size $5\,000$ with our algorithm, or even 42 parallel contraction hierarchies in the time it takes to completely contract our graph. In terms of query times our best results with a completely contracted core of $10\,000$ nodes even outperforms the results of the single-criteria contraction hierarchy without CALT, which needs 0.82 ms on average. Note that we only approximated this value assuming a similar curve as in Figure 5.5(a).

In Table 5.3 we give an overview of the results we get with completely contracted hierarchies. We use one bucket for every top level interval of the hierarchy.

Table 5.3: Preprocessing and query performance for different *graph*s without the usage of the CALT algorithm

	param/	preproc [hh:mm]		#	space	query	speed	nodes	edges	
graph	split	compute	I/O	split	[B/node]	[ms]	up	settled	relaxed	scanned
Ger	1 024/V	2:38	34	14	193	3.8	565	673	6 320	10 594
Ger	1 024/H	2:35	27	11	171	4.1	759	676	6 379	11 527
Eur	1 024/V	23:47	2:18	13	164	6.1	2 355	769	8 419	20 384
Eur	1 024/H	24:09	2:02	11	151	6.1	2 379	771	8 385	21 113

As Table 5.3 shows, and as could already be seen in the Figures 5.2, the split approach manages to reduce the query times compared to the complete contraction. We also reduce the contraction time to about 25% of the time needed to contract the complete parameter interval at once. The variable and the half-split method perform about the same in terms of preprocessing time, while the variable-split method yields slightly better query times. Due to the higher number of splits performed by the variable-split method the memory consumption may turn out slightly higher. If memory is not the deciding factor, we favor the variable-split method. In other cases the half-split method might be the method of choice. The noticeable difference in the query times is more a result of the scanned edges. The variable-split method seems to keep the number of edges smaller and therefore seems to result in better speed-ups. For the European road network the both methods do not seem to differ very much.

5.5 CALT for multi-criteria Contraction Hierarchies

Even though we managed to perform the contraction of even continental-sized networks in reasonable time, the query times so far are much higher compared to single-criteria speed-up techniques. In the following we present the benefits the CALT algorithm introduces in terms of query times. We will present experiments for CALT on contracted and uncontracted cores. We will also discuss the amount of data that we need for using the techniques.

Preprocessing. The CALT algorithm proved to be a viable method to generate even better speed-ups than just with contraction. The required preprocessing times for the CALT algorithm are very small compared to the contraction part of our hierarchy. This is a result of performing the speedup-technique only on the core of our hierarchy. This graph, even though it is a much denser graph, only consists of a very small amount of nodes. This allows a fast preprocessing time for the ALT algorithm. Furthermore, we expect already a small number of landmarks to provide very good lower bounds, since we have to cover only a small set of nodes. In addition to the small graph, our method to use only two weight terms to preprocess a single core per top level interval helps to drastically reduce the preprocessing times. Instead of $1\,024$ different cores, we only need to compute one core with two weight functions for every top level interval. This results in an amount of 22 to 32 landmark calculations. Therefore, we get preprocessing times 30 times faster than the preprocessing for $1\,024$ landmarks. An overview of preprocessing times for different core methods and landmarks can be found in Table 5.4.

Performance. Table 5.4 we provide a detailed analysis of our experimental results. For the preprocessing times we split the times into pure processing times and I/O time. This is reasonable, as we can save most of the time spent on disk access if we implement in memory compression or make more main memory available. Each of these methods could save the hard drive accesses generated by our splitting approach that writes out contracted parts of the hierarchy. As for now, the I/O time may make up a large portion of the preprocessing time of up to 30%. For the CALT algorithm we will give average values on a per top level basis.

Under these conditions, we obtain query times comparable to some single-criteria algorithms if we combine the landmark approach with a contracted hierarchy. We can provide query times well below one millisecond on average for the German road network. For the European road network we obtain query times of around 1.8 ms. The table also shows the large improvement we get by using landmarks. A contraction of the graph leads to a lot of very long shortcuts. The landmarks allow us to omit a large amount of nodes due to the lower bound of the shortest path distance. Even on an uncontracted core, the benefit from the landmark approach is very large. We get a factor of four compared to the core approach without landmarks. We even outperform the complete contraction by a factor of two. Especially on the contracted core the CALT algorithm proved to be very useful. If we compare the results for a contracted core with the results in Table 5.4, we can see a reduction by 80% percent in terms of query time for the CALT algorithm. This comes at the cost of only a few minutes extra preprocessing time. We can see that the slightly better landmarks of the max-cover approach do not reduce the query times significantly. Also, the number of 64 landmarks instead of 32 does not significantly reduce the query times for a small core. On the larger cores, a number of 64 landmarks can yield an improvement of about 3%. Compared to the long computing times for the contraction, the extra preprocessing time 64 landmarks increases the preprocessing time only by 2.1%. Therefore even the small speedup of 3% justifies the larger number of landmarks. We therefore recommend a number of 64 avoid landmarks for larger cores of about $5\,000$ nodes and 32 avoid landmarks for the smaller cores. For our algorithm we use an active set of four landmarks. We choose two landmarks each for the lower and upper bound landmark sets. As landmarks we choose the two landmarks that give us the largest potential values for the source and target proxy respectively. Since the source and the target may not belong to the core, we have to select proxy nodes to compute lower bounds. The selection of the nodes with the smallest distance to the source seems only natural, as they provide a good approximation for the distance to the actual source and target.

An obvious property of our data is that we provide the best query times for $p = 0$. This meets our expectations, as the single-criteria contraction hierarchy also performs best on the travel-time metric and results in considerable longer query times and preprocessing times for other metrics.

Our algorithm scales well with the size of the parameter interval. Doubling the size of the parameter interval only increases the preprocessing time by 25% and the space by less than 5% without really affecting the query time. We also did some experiments on the European road network with settings that worked well for the German road network. We could not perform a thorough investigation, as the preprocessing times were too long. Still, we managed to provide very good speed-ups of up to $8\,000$. These better speed-ups are expected for larger graphs, as we also observe the same behavior for the single-criteria CH. The space consumption for the European road network is even better, as it is less dense in comparison to the difficult German road network.

Table 5.4: Preprocessing times and performance for different numbers of landmarks and different landmark methods.

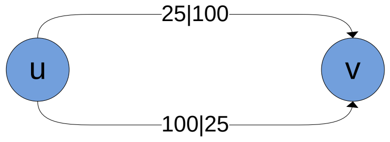
	graph/core	# parameter	# splits	lm	#	preprocessing contr [hh:mm]	IO [hh:mm]	LM [mm.ss]	space [B/node]	query [ms]	outside core[ms]	speedup	settled	relaxed
3 000 CORE	GH/3 000/U	1 024	12	A/L	16	1:18	27	0.03	166	2.00	0.47	1 021	944	7 233
	GH/3 000/U	1 024	12	A/L	32	1:18	27	0.07	170	1.80	0.47	1 150	878	6 126
	GH/3 000/U	1 024	12	M/L	32	1:18	27	2.22	170	1.74	0.47	1 181	861	5 876
	GH/3 000/U	1 024	12	A/L	64	1:18	27	0.15	175	1.69	0.47	1 219	843	5 612
	GV/3 000/U	1 024	14	A/L	32	1:23	30	0.07	190	1.70	0.48	1 196	884	6 199
5 000 CORE	GH/5 000/U	1 024	11	A/L	16	1:05	27	0.05	165	2.10	0.29	967	1 014	7 522
	GH/5 000/U	1 024	11	A/L	32	1:05	27	0.10	169	1.80	0.29	1 141	919	6 180
	GH/5 000/U	1 024	11	M/L	32	1:05	27	3.08	169	1.72	0.29	1 187	899	5 912
	GH/5 000/U	1 024	11	A/L	64	1:05	27	0.21	174	1.64	0.29	1 255	861	5 445
	GH/5 000/U	2 048	14	A/L	32	1:29	34	0.10	213	1.80	0.30	1 111	922	6 189
	GV/5 000/U	1 024	13	A/L	32	1:09	30	0.10	183	1.80	0.30	1 137	987	6 286
EUROPE	EH/10 000/U	24	0	A/L	32	2:16	16	0.42	50	1.48	0.17	9 435	1 238	9 709
	EH/10 000/U	512	6	A/L	32	6:28	1:51	0.42	109	3.50	0.45	4 017	1 611	16 496
	EH/5 000/U	1 024	10	A/L	64	10:54	1:53	0.30	144	3.90	0.89	3 600	1 583	19 422
	EH/10 000/U	1 024	10	A/L	64	9:02	1:51	0.54	145	3.90	0.45	3 580	1 671	18 627
CONTR. CORE	GH/3 000/C	1 024	12	A/L	32	2:35	27	0.10	173	1.20	0.47	1 749	648	3 624
	GH/5 000/C	1 024	12	A/L	32	2:35	27	0.19	175	0.90	0.29	2 207	576	2 937
	GH/10 000/C	1 024	12	A/L	32	2:35	27	0.40	179	0.70	0.15	2 863	504	2 256
	GH/10 000/C	1 024	12	A/L	64	2:35	27	1.00	179	0.66	0.15	3 123	475	1 978
	GV/10 000/C	1 024	13	A/L	64	2:38	34	1.20	193	0.74	0.15	2 768	505	2 240
	EV/10 000/C	1 024	14	A/L	64	23:47	2:18	1.30	165	1.80	0.56	8 039	872	7 625
	EH/10 000/C	1 024	12	A/L	64	24:09	2:02	1.00	152	1.70	0.45	8 184	855	7 264
	GH/3 000/U	1 024	12	-/-	-	1:18	27	-	164	8.5	0.47	240	2 620	42 606

Table 5.5: Performance of the time and cost core approach.

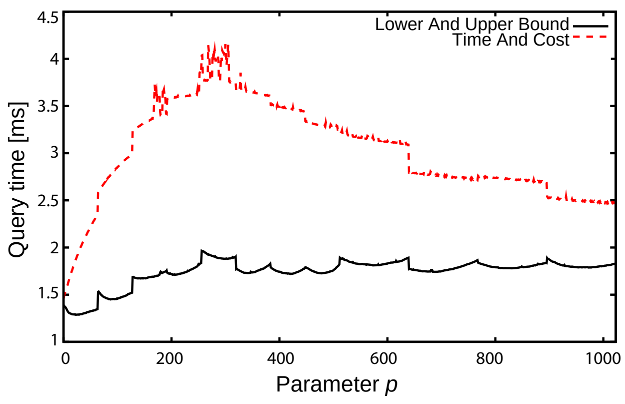
graph/core	# parameter	# splits	landmark method	# [B/node]	space [ms]	query	speedup	settled	relaxed
GH/3 000/U	1 024	12	A/T	16	166	3.0	696	1 198	12 325
GH/3 000/U	1 024	12	A/T	32	170	2.8	747	1 141	11 416
GH/3 000/U	1 024	12	A/T	64	175	2.7	780	1 102	10 855

Comparison of Lower and Upper bounds to Time and Cost weighted cores. In the Tables 5.4 and 5.5 we can see a large discrepancy between the query times for the lower and upper bound weighted cores and the time and cost weighted cores. Our experiments showed a clear reason for this difference. The problem with the time and cost approach is that we end up using only one lower bound to approximate the whole interval instead of two bounds as our preferred approach does. The lower and upper bound weighted core therefore approximates the lower bounds better. Figure 5.4(a) shows a small example where the lower and upper bound core are far superior to the time and cost core. Let us assume the parameter interval to be set to $[0, 10]$. In this example we would approximate the distance from u to v with the line $25 + p \cdot 25$ for the time and cost core. This only yields an acceptable lower bound for very small values of p. With the lower bound core we would approximate with $\frac{10-p}{10} \cdot 25 + \frac{p}{10} \cdot (100 \cdot 250) = 25 + p \cdot 32.5$. For $p = 10$ this leads to a lower bound of 275 for the time and cost core and a lower bound of 350 for the lower and upper bound core. The discrepancy of 75 is relatively small for this simple example. In larger graphs this discrepancy will get even larger. Since the time and cost core generates less accurate lower bounds as the lower and upper bound cores, the CALT algorithm has to search a considerably larger graph. This effect can be seen in the larger query times and larger search space in Table 5.5 compared to Table5.4. Especially in the parts of the hierarchy that have a large variation of possible routes, the time and cost core are outperformed by the lower and upper bound core by far, as the lower and upper bound core provides a more suited approximation on the right end of every parameter interval. This can be seen in Figure 5.4(b). If we combine Figure 5.4(b) with Figure 5.3(a), we can see that we get more comparable query times for the two core approaches in the parameter values that provide only a small variation as shown in Figure 5.3(a). The lower and upper bound core with 32 or 64 avoid landmarks – depending on the size of the core – are therefore the method of choice for the landmark-based MCCH algorithm. An interesting side note is that Table 5.4 shows a stronger effect for the switch from 32 to 64 landmarks for the time and cost core approach. Still, the time and cost core cannot reach the performance of the lower and upper bound core.

Summary. If we perform no landmark processing, the variable-split method will outperform the half-split method. In combination with the landmark approach, the half-split method gave the slightly better query times. It seems that somehow the half-split approach benefits more strongly from the CALT algorithm. Since the space consumption and preprocessing times are also better, we recommend the half-split method in a combination with a core of at least 10 000 nodes with 64 avoid landmarks for the best query times. Still, additional experiments showed that for some other cost functions the variable-split method might be superior to the half-split method. Especially if the effect we depict in Figure 5.3(a) is even stronger, the variable-split method might outperform the half-split method. As the most promising result we can present our core method. The core method proves very useful, as it significantly reduces the preprocessing time. Since we generate a large number of possible routes in the top levels of the hierarchy and the graph becomes very dense, the preprocessing time for the topmost levels is considerably large. Therefore, we can save a lot of preprocessing time with the core approach even if we only use a small core. In combination with the landmark approach we get even



(a) Illustration of a difficult core situation for the time and cost core.



(b) Query times over all parameter values

Figure 5.4: Illustration of differences between the time and cost core compared to the lower and upper bound core.

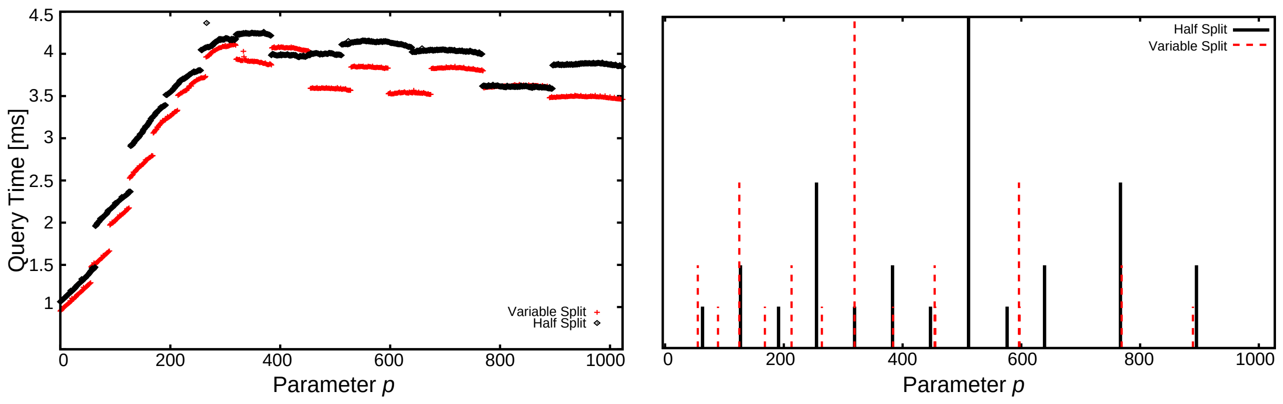
better query times than we get with the complete contraction without core. Still, our best results are, as Table 5.4 presents, with a contracted core and landmarks.

5.6 Pareto-SHARC vs MCCH

Even though the SHARC algorithm only manages to process continental-size road networks heuristically and we perform exact shortest path queries, we perform pretty well in comparison. Our comparison to the Pareto-SHARC algorithm bases on the contractions for the European road network for the parameters intervals $[0, 24]$ and $[0, 512]$. These contractions roughly represent the elongation in travel time Pareto-SHARC allows for the the pure ϵ-heuristic with $\epsilon = 0.02$ and the combination with the γ-heuristic with $\gamma = 1.0$ and $\epsilon = 0.5$ respectively. We present an overview of the comparison in Table 5.6. For the $[0, 24]$ parameter interval our algorithm performs an about as fast precalculation as the pure ϵ-heuristic of the Pareto-SHARC algorithm. This is even more impressive, since we use an updated version of the European road network with three times the number of edges. We cannot compare the query times directly since Pareto-SHARC actually calculates all possible non-dominated paths for its tightened domination criteria. But even if we perform our algorithm five times, which resembles the number of various routes SHARC can provide without resulting in extremely large preprocessing and query times, we can perform a faster query. For the parameter interval $[0, 512]$ we can only compare ourselves to the combination of the γ- and the ϵ-heuristic, since SHARC does not manage to preprocess continental-size road networks without both heuristics in less than 24 hours if they set ϵ to 0.1 or higher. But even compared to the double heuristic, which again limits the number of available paths to around five, we perform very well. Our preprocessing time takes about the same time as preprocessing for the Pareto-SHARC algorithm and a number of five subsequent queries takes only about half the time of a single Pareto-SHARC query. An additional bonus is that we manage to provide far more than the five different routes provided by the SHARC algorithm. Our approach offers a far more flexible choice of routes as we manage to provide 45 different routes on average.

Table 5.6: Comparison of the multi-criteria CH to SHARC on the European road network

MCCH				SHARC			
parameter	core	preprocessing [HH:MM]	query [ms]	ϵ	γ	preprocessing [HH:MM]	query [ms]
24	10 000	2:32	1.48	0.02	-	4:10	48.1
512	10 000	9:24	3.50	0.50	-	>24:00	-
512	10 000	9:24	3.50	0.50	1.0	7:12	35.0



(a) Illustration of the query times for the different splitting methods.

(b) Illustration of the splits performed for Figure 5.5(a)

Figure 5.5: Comparison of the half-split method and the variable-split method for Germany in a complete contraction.

Chapter 6

Conclusion and Outlook

6.1 Conclusion

In this thesis we present the first algorithm for fast flexible queries on continental-sized road networks. Although our algorithm is the first to provide such queries, we achieve query times comparable to single-criteria algorithms. We have managed to engineer the preprocessing to be very time- and space-efficient. As flexibility grows more important each year, algorithms – like the one we presented – also gain in importance. We managed to engineer our algorithm in such a way that we can provide fast queries in a server scenario, where a client performs a sequence of queries. Even though our query times are slower then the best available single-criteria methods, we provide very fast queries that are suitable for any server scenario as other parts of a client server communication takes more time than our query. Most algorithms presented so far use the Pareto-approach to provide flexibility. We have managed to avoid the problems of Pareto-optimality completely and still provide a set of very diverse routes. The amount of routes we provide is our main advantage in comparison to other flexible routing approaches. Also, we manage to provide exact shortest path queries, even for continental-sized road networks.

6.2 Future Work

There is still a lot of work to be done, though. Our algorithm still needs a lot of tuning. We might avoid most of the I/O-time by utilizing in-memory compression. This would result in even faster preprocessing times. Also, our algorithm can be parallelized very easily to perform faster preprocessing if we perform the contraction process for each split in parallel. This is a straight forward parallelization, since the different parts of the split are completely independent. Our parameter interval splitting could also possibly be tuned to perform better. We could factor in the length of the partial shortcuts to decide on better splitting points. Also, we could possibly combine some top level interval later if they seem to be very similar. Since most of the preprocessing time is spent in the top level interval of the hierarchy, this could save a lot of preprocessing time. We believe it to be possible to compute all shortest paths between a source and target pair efficiently with our algorithm. We believe this to be possible in proportional time to the number of different routes between the source and target pair. Also, it would be interesting to adapt our algorithm to a continuous range of parameter values. This should be possible in principal, but we believe it to be more of an academic question due to the added level of complication it requires. It should be possible to add even more flexibility if we allow a linear combination of more than two weight terms. The principal ideas our algorithm bases on are even valid in a multi-criteria scenario with more than two weight terms. Still, there remains the question of how to efficiently compute necessary shortcuts in such a scenario. We also believe that time-dependent routing should become flexible. This problem presents an even larger challenge, since it is even more difficult to optimize for a single criterion that is not the travel time but is also time-dependent. In analogy to our algorithm one might choose the term of time-dependent energy costs. As we calculate the cost term for our algorithm in dependence of the traveling speed one may factor in the different average traveling speeds during different times of the day, e.g., slow moving traffic during rush-hour. Also, a lot of other possible criteria we can factor in our cost term may depend on the time. For example special toll rates during different times of the day. The main problem is the time-dependency of the edge weights, and therefore most likely all Pareto-optimal pairs of travel time and the other weight term need to be computed. But hopefully, road networks are good-natured enough that this causes no hard problems in practice.

Appendices

Appendix A

Datastructures

This appendix illustrates some of the data structures used in the algorithm. The data structures presented are the static and the dynamic graph as well as the priority queue. We use the priority queue in the contraction process as well as during the queries. In contrast we only use the dynamic graphduring the contraction process, since, due to the shortcuts that we have to add in the process, the graph is subject to a lot of changes. During the query we use the static graph, since we do not have to change the graph. Finally we describe the file format we use for the multi-criteria Contraction Hierarchies.

A.1 Static Graph

The static graph is a common representation of the two graphs $G_{\uparrow}$ and $G_{\downarrow}$ as described in the preliminaries. It bases on the *forward star representation* [BBK03]. Due to our bucketing structure we use two adjacency arrays as a representation of our graph. This kind of representation uses a separate node and edge array. Since we need access to the incoming edges during the backward search of the Contraction Hierarchies query, we also store backward edges at each nodes. We sort the edge array using two criterion's. The first criterion is the source node. Therefore, the outgoing edges of a node are stored in order. Those resulting edge groups are then sorted into three groups. First the forward, then the bidirectional and finally the backward edges. We store, in addition to the normal edge, a sentinel edge at the front and a sentinel node at the end of the respective arrays. Note that the first edge id of the sentinel node actually point to invalid memory regions. We could prevent this with an additional sentinel edge at the edge of the edge arrays. But since the first edge of the sentinel edge will not be accessed in a correct algorithm, we do not insert the additional edge. The sentinel edge in the edge arrays prevent two things. Firstly we prevent edge underflows during a backward search edge array scan, secondly we prevent the ambiguous edge id zero.

Figure A.1 and Figure reffig:bucketing-detail depict a possible search graph and its representation in the search graph as we already described in Section 4.5.3. For simplicity reasons we limit the example to only two possible parameter values. Note that if we allow more parameters we can get edges that are in a subset of the buckets. It is even possible for one edge to belong to a nonconsecutive set of buckets, e.g., one edge may belong to the buckets one, four, five and nine. We also do not show the landmark distances. Assuming we can map all nodes correctly without needing extra mappings, we can store the landmark distances in a number of $|Top - Level|$ arrays of length $|core|$. If we combine it with the id of the top level interval, the ambiguous mappings from graph nodes to core ids becomes unique. This allows for an O(1) access to the landmark distances.

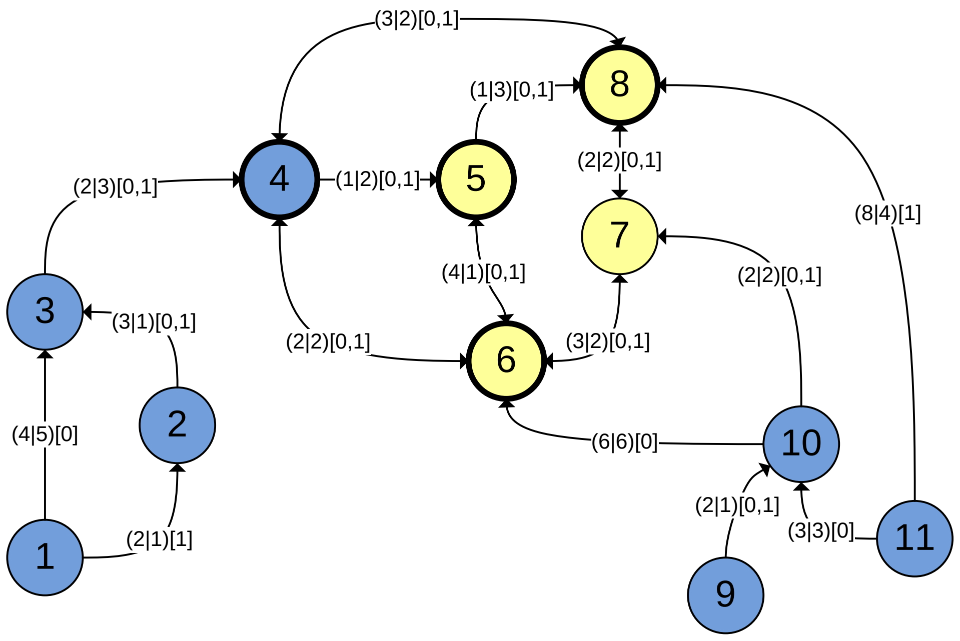
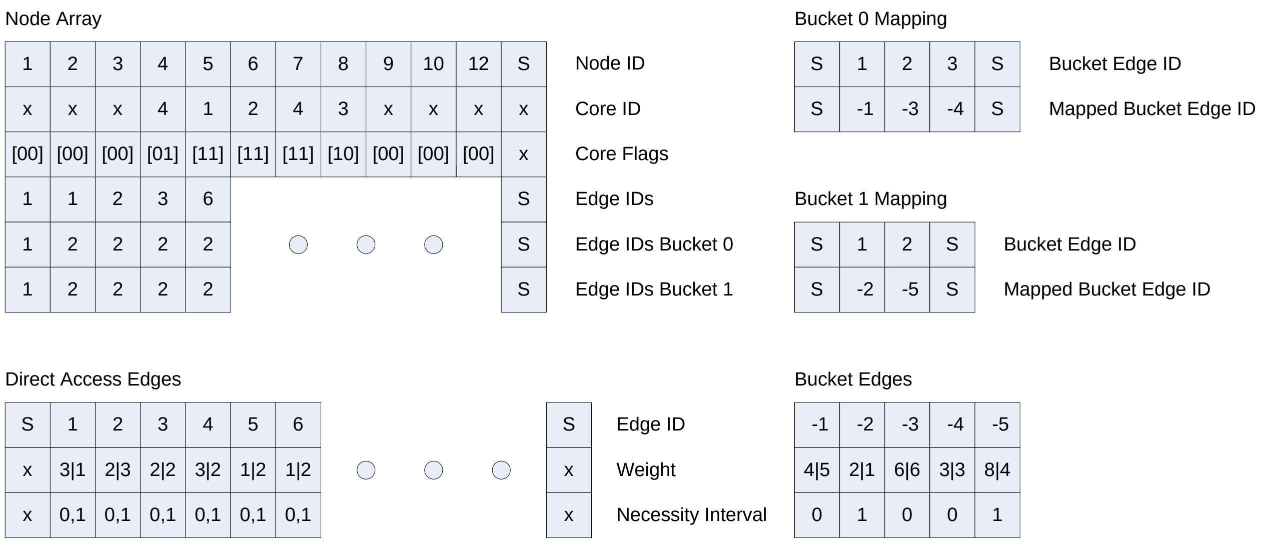


Figure A.1: Sample graph for two parameter values $(0, 1)$. The graph consists of a hierarchy part and two top level cores. The nodes of the first core are marked in bold; the nodes of the second core are colored differently than the other nodes.



Node Array

1	2	3	4	5	6	7	8	9	10	12	S	Node ID
x	x	x	4	1	2	4	3	x	x	x	x	Core ID
[00]	[00]	[00]	[01]	[11]	[11]	[11]	[10]	[00]	[00]	[00]	x	Core Flags
1	1	2	3	6							S	Edge IDs
1	2	2	2	2		○		○	○		S	Edge IDs Bucket 0
1	2	2	2	2							S	Edge IDs Bucket 1

Bucket 0 Mapping

S	1	2	3	S	Bucket Edge ID
S	-1	-3	-4	S	Mapped Bucket Edge ID

Bucket 1 Mapping

S	1	2	S	Bucket Edge ID
S	-2	-5	S	Mapped Bucket Edge ID

Direct Access Edges

S	1	2	3	4	5	6				S	Edge ID
x	3|1	2|3	2|2	3|2	1|2	1|2	○	○	○	x	Weight
x	0,1	0,1	0,1	0,1	0,1	0,1				x	Necessity Interval

Bucket Edges

-1	-2	-3	-4	-5
4|5	2|1	6|6	3|3	8|4
0	1	0	0	1

Figure A.2: Simplified representation of the sample graph in Figure A.1. We do not show the edge direction flags and do not show details for every edge.

A.2 Dynamic Graph

We extend the basic structure of the static graph to form a dynamic graph. We augment the edge array for each node to hold possible future edges. By default we choose the edge array for each node to be able to hold twice the original edges. The actual size may be controlled by a tuning parameter to control the overhead. Whenever we need to insert an edge into the graph, the dynamic graph will replace one of the empty slots until the edge array is completely filled. If we have to insert a further edge, we will double the size of the current edge array again, and we will reallocate the edges of the node at the end of the edge array. This may lead to a long running operation. Also, this method introduces a a lot of wasted space, since many holes may be generated in the edge array. To overcome this shortcoming, we may choose to compact the graph from time to time. Since compaction may have to move a huge amount of edges this operation should be used very carefully.

A.3 Priority Queue

The current implementation uses an adressable binary heap priority queue implementation provided by Dominik Schultes. He devided the heap into a data array and an index array. In the data array the user may store additional user data. In the index array he realizes the actual heap functionality. An item of the heap itself is just represented by a key, thus increasing the performance of a single heap operation, since only small elements have to be moved. The heap will not move the user data. Therefore, the index of an element in the heap has to be known in advance. We store this number within the node array, thus interweaving the priority queue and the graph. We also have to update this index into the heap every time we perform a heap operation.

A.4 Multi Criteria Contraciton Hierarchy Fileformat

The multi-criteria contraction hierarchy input and search graphs are stored using so called protocol buffers. The protocol buffers are a data-structure developed by Google and made available to the public. Its original purpose is the use as a data interchange format in servers but are also quite useful in data storage. The goal is to serialize data in and xml like manor but at the same time avoid the overhead of ascii-files. The file format for a protocol buffer file is specified in plain text format. The text format then is compiled into source and header files providing all necessary routines for file access.

The graph format of a mcch graph is divided into multiple parts. First there is a single protocol header message containing the number of nodes n, the number of top level interval t and the number of edges e in the graph. This header is then followed by n node messages and e edge messages. Finally a number of t top level bounds follows.

A node message contains the ID as a mandatory part of the message with $ID \in [0, n-1]$. Optional the id in the graph as specified by the PTV-AG may be stored as well as a node level for a contracted node and the id of the country the node belongs to. The node messages may be omitted for a contracted hierarchy, as node levels are not necessary if the edges are already filtered into upwards edges for the forward and backward graph.

The edges contain a source and a target that are guaranteed to lie in $[0, n-1]$. Optional following parameters may be specified:

- direction: either eCLOSED, eFORWARD, eBACKWARD or eBIDIRECTIONAL describing the directions the edge may be traveled in. Note that in a normal multi-criteria contraction hierarchy no closed edge should be stored.

- weight: the weight function of the edge split into its linear components starting with the constant part.
- middle node: if the edge describes a shortcut the middle node can be stored to enable shortcut extraction.
- necessity interval: the intervals for which parameters a given edge is necessary. This is especially important, since a lot of edges can be skipped during the search yielding faster speedups.

After the edges we can choose to write the bounds of the splits generated by our splitting approach. These can be useful, as they provide a good choice for possible buckets.

Bibliography

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[BBK03] Daniel K. Blandford, Guy E. Blelloch, and Ian A. Kash. Compact Representation of Separable Graphs. In *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'03)*, pages 679–688, 2003.

[BD08] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.

[BDS$^+$08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[BGS08] Gernot Veit Batz, Robert Geisberger, and Peter Sanders. Time Dependent Contraction Hierarchies - Basic Algorithmic Ideas. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2008.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[Del09] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, inf-ka, 2009. `http://i11www.ira.uka.de/extra/publications/d-earpa-09.pdf`.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[DW09] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 125–136. Springer, June 2009.

[Gei08] Robert Geisberger. Contraction Hierarchies. Master's thesis, inf-ka, 2008. http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[Han79] P. Hansen. Bricriteria Path Problems. In Günter Fandel and T. Gal, editors, *Multiple Criteria Decision Making – Theory and Application –*, pages 109–127. Springer, 1979.

[IHI+94] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by AI search techniques. In *Proceedings of the Vehicle Navigation and Information Systems Conference (VNSI'94)*, pages 291–296. ACM Press, 1994.

[KM08] Peter Sanders Kurt Mehlhorn. *Algorithms and Data Structures*. Springer, 2008.

[KO80] Richard M. Karp and James B. Orlin. Parametric Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Mathematics*, 3:37–45, 1980.

[KS93] David E. Kaufman and Robert L. Smith. Fastest Paths in Time-Dependent Networks for Intelligent Vehicle-Highway Systems Application. *Journal of Intelligent Transportation Systems*, 1(1):1–11, 1993.

[Lau97] Ulrich Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[Mar84] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[Möh99] Rolf H. Möhring. Verteilte Verbindungssuche im öffentlichen Personenverkehr – Graphentheoretische Modelle und Algorithmen. In Patrick Horster, editor, *Angewandte Mathematik insbesondere Informatik, Beispiele erfolgreicher Wege zwischen Mathematik und Informatik*, pages 192–220. Vieweg, 1999.

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.

[MW01] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[NY91] James Orlin Neal Young, Robert Tarjan. Faster Parametric Shortest Path and Minimum Balance Algorithms. *Networks*, 21(2):205–221, 1991.

[OR90] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.

[PM98] Anna Sciomachen Paola Modesti. A utility measure for finding multiobjective shortest paths in urban multimodal transportation networks. *European Journal of Operational Research.*, 111(3):495–508, 1998.

[Poh71] Ira Pohl. Bi-directional Search. In Bernard Meltzer and Donald Michie, editors, *Proceedings of the Sixth Annual Machine Intelligence Workshop*, volume 6, pages 124–140. Edinburgh University Press, 1971.

[RE09] Andrea Raitha and Matthias Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers and Operations Research*, 4(36):1299–1331, 2009.

[Sch08a] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, inf-ka, January 2008.

[Sch08b] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, inf-ka, February 2008. `http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf`.

[SS05] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, June 2007.

[Zie01] Mark Ziegelmann. *Constrained Shortest Paths and Related Problems*. PhD thesis, Universität des Saarlandes, 2001.