Karlsruhe Institute of Technology

A specialized model for optimized memory consumption of time-dependent Contraction Hierarchies

Master Thesis of

Sebastian Schmidt

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers: Prof. Dr. Dorothea Wagner
Prof. Dr. Peter Sanders

Advisors: Tim Zeitz M.Sc.
Dr. Torsten Ueckerdt

Time Period: 1st September 2018 – 28th February 2019

KIT – The Research University in the Helmholtz Association

www.kit.edu

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 2019/02/28

Abstract

Time-dependent routing is becoming practical, as more detailed GPS traces of car rides become available. But using more detailed data for time-dependent routing means using more memory for applying speedup techniques. Customizable Contraction Hierarchies are a robust variant of Contraction Hierarchies that can cope with difficult metrics. They have the disadvantage of higher memory consumption, but allow for a very fast metric-dependent customization. In this work, we expand the CCH with time-dependent edge weights. We propose an optimization scheme for the input weight functions to reduce the space consumption of the TDCCH, while keeping the index exact on the optimized metric. We argue, that our weight optimization introduces only very small significant errors, while mostly staying below the noise of the input data.

Deutsche Zusammenfassung

Zeitabhängige Routenplanung wird momentan praktikabel, da mehr und mehr GPS-Daten über Autofahrten verfügbar werden. Aber die Nutzung von detaillierteren Daten für Zeitabhängige Routenplanung führt auch zu höherem Speicherverbrauch für Beschleunigungstechniken. Customizable Contraction Hierarchies sind eine robustere Variante von Contraction Hierarchies, die auch mit schwierigen Metriken umgehen können. Sie haben den Nachteil des höheren Speicherverbrauch, erlauben aber eine sehr schnelle metrikabhängige Customization. In dieser Arbeit erweitern wir die CCH mit zeitabhängigen Kantengewichten. Wir stellen ein Optimierungsschema für die Eingabefunktionen vor, um den Speicherverbrauch einer exakten TDCCH auf den optimierten Eingabedaten zu verringern. Wir argumentieren, dass unser Gewichtsoptimierungsschema nur sehr kleine signifikante Fehler einfügt, und größtenteils unter dem Rauschen in den Eingabedaten bleibt.

Contents

1. Introduction **1**
1.1. Related Work . . . 1
1.2. Our Contribution . . . 3
1.3. Outline . . . 3

2. Exact Approach **5**
2.1. Input Data . . . 5
2.1.1. Representation of Road Networks . . . 5
2.1.2. Interpretation of Input Data . . . 12
2.2. The Exact TDCCH . . . 13
2.2.1. Building an Exact TDCCH . . . 13
2.2.2. Exact Evaluation . . . 16
2.2.3. Exact Linking . . . 16
2.2.4. Exact Merging . . . 22
2.2.5. Problems with the Exact Approach . . . 23

3. Approximate Approach **25**
3.1. Phase Representation . . . 25
3.1.1. Evaluation . . . 27
3.1.2. Conversion . . . 27
3.1.3. Space Requirements . . . 28
3.2. The Approximate Approach . . . 29
3.2.1. Cover Potentials . . . 29
3.2.2. Maximizing the Cover Potentials by Approximation . . . 34
3.3. Linking of Phase Functions with Optimal Result Size . . . 43
3.3.1. Phase Function Views . . . 43
3.3.2. Covering . . . 44
3.3.3. LMC Algorithm . . . 45
3.3.4. Computing an Optimal Link . . . 52
3.4. Merging of Phase Functions . . . 54
3.4.1. Phase Merge Algorithm . . . 55
3.5. Weight Optimization . . . 55
3.5.1. Scheme . . . 55
3.5.2. Discussion . . . 56

4. Experiments **57**
4.1. Implementation . . . 57
4.1.1. Time-Expansion . . . 57
4.1.2. TDCCH . . . 58
4.2. Results . . . 59
4.2.1. Customization . . . 60
4.2.2. Query . . . 62

4.3. Comparison & Evaluation 66
4.3.1. Other techniques 66
4.3.2. TCH 67
4.3.3. Potentially Systematic Errors 68

5. Conclusion & Outlook **71**

Bibliography **73**

Appendix **75**
A. Experiment Plots 75

1. Introduction

The basic problem in routing is finding a shortest path with respect to some metric between two locations in a network. This metric can be distance, fuel usage, power usage, travel time or anything else one can imagine. When travel time is used as metric, and the network is a road network, then a problem arises. Travel times in road networks are not constant, but they vary from hour to hour, depending on the current traffic situation. This prohibits calculating correct optimal routes in a graph with a scalar travel time metric, because after a short time of ride, the travel times might have changed already.

In practice, traffic follows certain basic patterns that rarely change. For example, many people commute every morning and evening to and from work, and this is the most dominant source of traffic. These basic patterns allow the prediction of the behavior of traffic throughout the week with measurements of the past. This opens up the gates for time-dependent routing to improve the accuracy of shortest path queries in real-world road networks.

In static routing, the algorithmic goal is to answer shortest path queries fast. Since the network and the metric are static, expensive preprocessing is possible that allows for queries in the order of microseconds or even less. An increment to this is dynamic routing, that expands static routing with the requirement to change the metric fast to dynamically react to changing traffic situations. In this scenario, expensive metric-dependent preprocessing is prohibited, but the preprocessing from static routing techniques can be split up in metric-independent, and cheaper metric-dependent preprocessing. Solutions to the dynamic routing problems are often called customizable, where the metric-dependent preprocessing is called the customization.

In time-dependent routing, the basic problem is finding a shortest path in a static road network, but with time-dependent edge weights. A time-dependent shortest path query asks for the shortest path from node s to node t, when departing from s at time τ. The algorithmic challenge of time-dependent routing is to answer shortest path queries fast, but with time-dependence.

1.1. Related Work

There exist some speedup techniques for time-dependent, static and dynamic routing that we discuss in this section. In general, there are two problems related to time-dependent shortest paths. The first is to compute the length of a time-dependent shortest path given

a source, a target, and a departure time. This is called an earliest arrival time query. The second is to compute the length of all time-dependent shortest paths throughout a certain time frame, given a source and a target. This is called profile search.

The problem of finding a shortest path, or a shortest path profile in a time-dependent road network can be solved using a slightly modified version [CH66, Dre69] of Dijkstra's algorithm [Dij59], if the road network fulfills certain properties [OR90].

There are many existing speedup techniques for static routing. Goal-directed search techniques include ALT [GH05] and Arc-Flags [BD08]. Hierarchy-based techniques include single- and multi-level overlays, Reach [Gut04], and continuous overlays like Contraction Hierarchies [GSSD08] and Customizable Contraction Hierarchies [DSW16]. There are also other techniques like Hub Labeling [ADGW11] and combinations of these [BDG$^+$16].

For dynamic routing, Customizable Route Planning (CRP) [DGPW11], which is a special form of multi-level overlays, and Customizable Contraction Hierarchies are important techniques.

Some speedup techniques for time-independent routing have been adapted to time-dependent routing. They include TD-ALT [NDLS12], TD-Arc-Flags [Del11], different approaches to TDCHs [BGSV13, KLSV10] and TDCRP [BDPW16]. From the best of our knowledge, there is no publication expanding CCHs to time-dependent routing yet.

TD-ALT utilizes the A*-algorithm and the triangle inequality to direct the Dijsktra search towards landmarks that are behind the destination node. Combining this with contraction (TD-CALT) yields reasonable query times for country-sized instances and also allows fast dynamic traffic updates [NDLS12]. The basic idea of Arc-Flags is to divide the road network into regions and store flags for each road segment that state if the road segment is part of a shortest path into a region or not. Combining time-dependent Arc-Flags with landmarks and hierarchical regions (TD-L-SHARC) yields reasonable query times as well, while also allowing for profile searches [Del11]. Contraction Hierarchies build a continuous overlay over the graph for a certain node order, which is greedily determined during the building process. The node order is optimized to reduce the complexity of the overlay, while keeping the maximum length of paths that only move up in the node order short. In [BGSV13], the authors adapt this approach to the time-dependent scenario (TCH), allowing for significantly faster exact queries than other approaches on country-scale road networks with reasonable preprocessing time, but also with significantly higher space consumption. This prohibits using the index on continent-scale road networks with modern travel time data. Building only approximate overlays (ATCH) and using the approximation to determine a small corridor for exact querying reduces the memory usage of TDCHs significantly, down to the level of TD-L-SHARC, while increasing query time by less than a factor of two. The query times of the ATCH are still significantly better than those of TD-L-SHARC. A recent alternative is TDCRP [BDPW16]. The basic idea is to separate the road network into regions with a small amount of boundary nodes. For each region, a complete graph between the boundary nodes is introduced as overlay. This process is repeated for multiple layers of overlays. TDCRP allows for fast updates of the time-dependent metric and fast query times on continental scale road networks. But to keep space consumption reasonable, it uses approximation and introduces small query errors.

In a theoretical work, the authors the complexity of profiles of time-dependent shortest paths in graphs with piecewise linear arrival time functions as edge weights [FHS11]. They prove that the worst-case boundary for the complexity of a profile is superpolynomial in the size of the graph. There are graphs for which this is still true, if the edge weights are just linear arrival time functions. They also prove that the worst case complexity over all graphs can be reduced to a boundary, that is only linearly dependent on the total amount

of breakpoints for all weight functions in the graph, if the slopes of the weight functions are restricted to 0, 1 and ∞.

1.2. Our Contribution

In this work, we evaluate time-dependent customizable contraction hierarchies (TDCCHs) with restricted slopes on country-scale road networks, and give an optimization scheme to reduce their memory consumption. The optimization scheme takes the form of a separate customization step, in which the input weights are slightly modified within a user-defined threshold. On the modified weights, we build an exact TDCCH, and thus our accelerated queries are exact on the modified input weights. We show that the errors, with respect to the original metric with unrestricted slopes, introduced by the optimization scheme, are small.

Our optimization scheme is based on the observation that in practice, link operations introduce most of the points that make up the memory consumption of a TDCCH. We theoretically examine the behavior of piecewise linear functions in link operations, focusing on the size of the resulting link. With the results from this examination, we propose the *phase model* for piecewise linear functions that allows for optimization of the link size. We devise an algorithm that solves a related interval cover problem optimally in linear time to optimize the size of a link. We apply this algorithm in the weight optimizing customization, which modifies the input functions within a user-defined boundary when they are linked. This reduces the size of the customized exact TDCCH by 50% compared to a customized exact TDCCH on the same input data with restricted slopes, but without optimization.

1.3. Outline

In Chapter 2, we introduce the representation and interpretation of our data and describe how to build an exact TDCCH. We also give proof on the correctness of an algorithm for linking piecewise linear functions. In Chapter 3, we introduce the phase model and an algorithm to link near-optimally in this model. We also examine the link operation in detail and introduce the weight optimization scheme. In Chapter 4, we describe our implementation of the link and merge algorithms in the phase model, and the results of our experiments on different graphs. In Chapter 5, we recapitulate our results and give a prospect on what further research we want to do to expand on our contribution.

2. Exact Approach

In this chapter, we introduce the input data to our algorithms, and describe how to build an exact TDCCH. We particularly prove the correctness of an algorithm to link piecewise linear functions to allow arguing about the link in more detail in Chapter 3.

2.1. Input Data

In this work, we want to optimize the memory consumption and customization speed of a TDCCH. We base our research on data about real-world road networks of different countries and regions kindly provided by the *PTV Group* in 2017. The data has a certain format, that we describe in detail in Subsection 2.1.1. Also, since we work with real-world data, we discuss its accuracy and interpretation in Subsection 2.1.2.

2.1.1. Representation of Road Networks

Road networks are modeled as directed graphs. Each node with degree larger than two represents a crossing in the road network. Nodes with degree two split a road segment into two, and nodes with degree one are dead ends. Edges represent the road segments that connect the nodes. Since we want to do shortest path queries with a metric, the edges are weighted. We have three different metrics for our graphs.

First is the distance metric, it weighs each road segment according to its length in meters. Second, the time-independent travel time metric, which gives the time in seconds it takes to travel along a road segment, assuming no other traffic is present. Third and last, the time-dependent travel time metric. It is an estimation of the travel time along a road segment, depending on the time of day the road segment was entered. We have different metrics for each day of the week, except that there is only one for Tuesday to Thursday. The departure time is always given in seconds since midnight, and the travel time in seconds.

All of these metrics have in common that they are non-negative. While the first two metrics are scalar, the time-dependent travel time metric is functional. Its functions are piecewise linear, so they are made up of a set of linear functions.

Definition 2.1 (Linear Function)**.** *A* linear function $f : D \subseteq \mathbb{R} \to \mathbb{R}$ *is a function defined on an interval* D *with constant slope* $s(f)$ *and offset* $t(f)$*. For* $x \in D : f(x) := s(f) \cdot x + t(f)$*.*

A piecewise linear function is a set of linear functions defined on certain intervals. The linear pieces are interrupted by what we call breaks. Breaks can also be positive discontinuities, but we ignore negative discontinuities when defining breaks, since we do not need them in this work, as elaborated below.

Definition 2.2 (Break)**.** *Let f be a real function and f' its derivative. Then, an $x \in \mathbb{R}$, such that*

$$\lim_{\epsilon \to 0} f'(x-\epsilon) \neq \lim_{\epsilon \to 0} f'(x+\epsilon), \tag{2.1}$$

or

$$\lim_{\epsilon \to 0} f(x-\epsilon) < \lim_{\epsilon \to 0} f(x+\epsilon) \tag{2.2}$$

is called a break *of f.*

With the notion of breaks, we define piecewise linear functions.

Definition 2.3 (Piecewise Linear Function)**.** *A* piecewise linear function *$f : D \subseteq \mathbb{R} \to \mathbb{R}$ is a function defined on an interval D that is linear on $D \setminus B$ with $B \subset \mathbb{R}$ being a finite set of breaks.*

We also define the sequence of linear pieces of a piecewise linear function that we use further below.

Definition 2.4 (Sequence of Linear Pieces, Linear Piece)**.** *Let $f : D \subseteq \mathbb{R} \to \mathbb{R}$ be a piecewise linear function that is linear on $D \setminus B$ with $B := (x_1, \dots, x_n)$ being a finite sequence of breaks. Then $L := (f|_{(-\infty,x_1)}) \cdot (f|_{(x_i,x_{i+1})})_i \cdot (f|_{(x_n,\infty)})$ is its* sequence of linear pieces*, with $\cdot$ being the sequence concatenation. Each element $l \in L$ is called a* linear piece*.*

Note that joining two linear pieces never yields another linear piece.

Throughout this work we use a notation for subsequences that we define here.

Definition 2.5 (Subsequence Notation)**.** *Let $S := (s_1, \dots, s_n)$ be a sequence. Let $P : S \to \mathbb{F}_2$ be a predicate. Then $T := (s \mid s \in S \land P(s) = 1)$ is a subsequence of S. With this notation, T is defined to be sorted according to S, meaning that if $a, b \in T$, such that a is before b in T, it holds that a is before b in S.*

In this work, we represent piecewise linear functions by sorted sequences of support points that we call support sequence. The sequence is first sorted by increasing x. We only need piecewise linear functions that have positive discontinuities, because we work with piecewise linear functions that are FIFO in this work, as defined further below. A positive discontinuity, or jump, at some x_0 in some function f is a discontinuity with positive infinite slope, such that $\lim_{\epsilon \to 0} f(x_0 - \epsilon) < \lim_{\epsilon \to 0} f(x_0 + \epsilon)$. So in the support sequence, support points with the same x value are sorted by their y value. Note that the following definition also allows the support sequence to contain support points that can be removed without altering the derived piecewise linear function of the support sequence as defined below.

Definition 2.6 (Support Sequence, Support Point)**.** *Let $S := ((x_1, y_1), \dots, (x_n, y_n)) \in (\mathbb{R}^2, \dots, \mathbb{R}^2)$ be a sequence of $n \in \mathbb{N}_{\geq 1}$ points such that $\forall i < j : x_i \leq x_j$ and $\forall i < j : x_i = x_j \Rightarrow y_i \leq y_j$. Then S is called a* support sequence*. The points in S are called* support points*.*

The points in a support sequence are ordered, and we often refer to this ordering in this work by calling a point left or right to another.

Definition 2.7 (Leftness and Rightness)**.** *Let S be a support sequence. Let $p_i, p_j \in f$ be the i-th and j-th point in S.*

- *If $i < j$, then p_i is* left of/before *p_j and $p_i < p_j$.*
- *If $i > j$, then p_i is* right of/after *p_j and $p_i > p_j$.*

For deriving a piecewise linear function from a support sequence, we need a notion of interpolation between two points and from a point with a given slope. We use the interpolation between two points for defining the function between points, and the interpolation with a point and a slope for defining the function before the first and after the last point.

Definition 2.8 (Interpolation Based on Points)**.** *Let $p := (p_x, p_y)$ and $q := (q_x, q_y)$ be points such that $p_x \leq q_x$. Let $x \in [p_x, q_x]$. Then the interpolated value of p and q at x is $y := (q_x - x) \cdot p_y + (x - p_x) \cdot q_y$.*

Definition 2.9 (Interpolation Based on a Point and a Slope)**.** *Let $p := (p_x, p_y)$ be a point and $m \in \mathbb{R}$ a slope. Let $x \in \mathbb{R}$. Then the interpolated value of p and m at x is $y := p_y + (x - p_x) \cdot m$.*

Since we define piecewise linear functions on $\mathbb{R}$, but allow the support sequence to be finite, we need to define how the function behaves before the first and after the last point. We introduce the *implicit slopes* for that, which are the slopes of the first and last linear segment of a piecewise linear function.

Definition 2.10 (Implicit Slopes)**.** *Let f be a piecewise linear function with a finite amount of breaks. Then the* implicit slope *$m_l(f)$ is the slope of the first linear segment of f. And the* implicit slope *$m_r(f)$ is the slope of the last linear segment of f.*

From a support sequence S and slopes m_l and m_r, we derive a piecewise linear function f_S. In most parts of this work m_l and m_r are implicitly 1, so we do not include them in the subscript of f.

Definition 2.11 (Derived Piecewise Linear Function)**.** *Let S be a support sequence and $m_l, m_r \in \mathbb{R}$. Then f_S is the* derived piecewise linear function *of S and m_l and m_r, such that $f_S(x)$ is defined for each $x \in \mathbb{R}$ as follows.*

Let $p := (p_x, p_y)$ be the rightmost point in S with $p_x < x$. Let $q := (q_x, q_y)$ be the leftmost point in S with $x \leq q_x$.

- *the interpolated value y of p and q at x if both exist, or*
- *the interpolated value y of p and m_l at x if q does not exist, or*
- *the interpolated value y of q and m_r at x if p does not exist.*

This is well-defined, because support sequences contain a minimum of one point, so either p or q exists. Note that the definition of p and q causes a derived piecewise linear function f_S to attain the left-sided limit $\lim_{\epsilon \to 0} f(x_0 - \epsilon)$ of a discontinuity at some x_0. So derived piecewise linear functions are always left-continuous. If S contains only one point and $m_l = m_r$, then the derived piecewise linear function f_S is linear.

We sometimes refer to the function trace of a piecewise linear function. Since we only work with piecewise linear functions with positive discontinuities in this work, we define the function trace only for piecewise linear functions without negative discontinuities.

Definition 2.12 (Function Trace)**.** *Let $f : D \subseteq \mathbb{R} \to \mathbb{R}$ be a piecewise linear function. Then the sequence $F(f) := \textsc{sort}(\{(x, y) \mid x \in D \wedge y \in [\lim_{\epsilon\to 0} f(x-\epsilon), \lim_{\epsilon\to 0} f(x+\epsilon)]\})$ that is sorted lexicographic first by x and then by y is its* function trace*.*

We note, that all support points in a support sequence are in the function trace of its derived piecewise linear function.

Lemma 2.13 (Support Points are on the Function Trace)**.** *Let S be a support sequence. Let f_S be its derived piecewise linear function. Let $F(f_S)$ be its function trace. Then $S \subseteq F(f_S)$.*

Proof. Let $p := (x, y) \in S$. Then f_S is defined on x and $y \in [\lim_{\epsilon\to 0} f(x-\epsilon), \lim_{\epsilon\to 0} f(x+\epsilon)]$. So $p \in F(f_S)$. □

Since we want to reduce the memory usage of the contraction hierarchy, it is important that we can refer to piecewise linear functions derived from support sequences that do not contain redundant points. Redundant points are all points that can be left out of a support sequence, without altering the derived piecewise linear function. Those are duplicate points and points that are not breakpoints. Duplicate points are points $p := (p_x, p_y)$ and $q := (q_x, q_y)$ with $p_x = q_x$ and $p_y = q_y$. Breakpoints are and expansion of breaks and defined as follows.

Definition 2.14 (Breakpoint)**.** *Let f be a piecewise linear function and f' its derivative. Let $p := (x, y) \in F(f)$ be a point on the function trace of f. Then p is called a* breakpoint*, if*

$$\lim_{\epsilon\to 0} f(x-\epsilon) = \lim_{\epsilon\to 0} f(x+\epsilon) \tag{2.3}$$

$$\wedge \lim_{\epsilon\to 0} f'(x-\epsilon) \neq \lim_{\epsilon\to 0} f'(x+\epsilon) \tag{2.4}$$

or

$$\lim_{\epsilon\to 0} f(x-\epsilon) < \lim_{\epsilon\to 0} f(x+\epsilon) \tag{2.5}$$

$$\wedge \left(y = \lim_{\epsilon\to 0} f(x-\epsilon) \vee y = \lim_{\epsilon\to 0} f(x+\epsilon)\right) \tag{2.6}$$

Equations (2.3) and (2.4) demand that a breakpoint $p \in F(f)$, if f is continuous at p, marks a change in the slope of f. Equations (2.5) and (2.6) demand that a breakpoint $p \in F(f)$ that is at a jump in f, marks the beginning or end of that jump. We split the definition into two cases to avoid the special case of a jump that does not change the slope. A piecewise linear function with breakpoints is depicted in Figure 2.1

We prove the relationship between breakpoints and breaks.

Lemma 2.15 (Breakpoints are an Expansion of Breaks)**.** *Let f be a piecewise linear function without negative discontinuities.*

$$f \text{ has a break at } x \iff f \text{ has a breakpoint } (x, y) \text{ at some } y$$

Proof. The equivalence follows directly from the definitions of breaks and breakpoints, Definition 2.2 and 2.14. □

Piecewise linear functions that have breakpoints are also called nonlinear piecewise linear functions, and piecewise linear functions that do not have breakpoints are linear functions. We prove a lemma about the origin of breakpoints in derived piecewise linear functions.

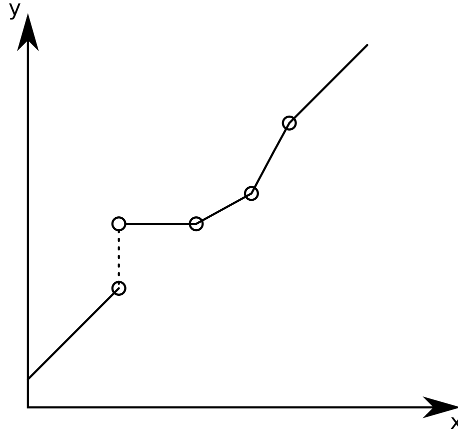


Figure 2.1.: An exemplary piecewise linear function with implicit slopes of one. Its function trace is depicted as solid and dashed lines. The breakpoints of the function are circled. The dashed line represents a jump. Values on the dashed line are never attained, since the function is left-continuous. The function trace contains the points around and including the second breakpoint as well, but we do not draw it here to emphasize the left-continuity.

Lemma 2.16 (Origin of Breakpoints in Derived Piecewise Linear Functions)**.** *Let S be a support sequence and m_l and m_r be implicit slopes. Let f_S be their derived piecewise linear function. If f_S has a breakpoint $p := (x, y)$, then $p \in S$.*

Proof. Let $S := ((x_1, y_1), \ldots, (x_n, y_n))$ be a support sequence and m_l and m_r be implicit slopes. Let f_S be their derived piecewise linear function.

Let $q := (q_x, q_y)$ be a breakpoint of f_S such that $q \notin S$. Let $p := (p_x, p_y)$ be the rightmost point in S with $p \leq q$. Let $r := (r_x, r_y)$ be the leftmost point in S with $r \geq q$.

The points p and r exist because by Definition 2.11 f_S is linear for $x < x_1$ and $x > x_n$.

If $p = q$ or $q = r$ then $q \in S$. Otherwise, $p < q < r$.

- If $p_x < q_x < r_x$. Then $f_S(q_x)$ and all its neighbors are determined by interpolation between p_x and r_x. So f_S is linear around q_x. So q is not a breakpoint in f_S.
- If $p_x = q_x = r_x$, then $p_y < q_y < r_y$. So Equations (2.3) and (2.6) do not hold for q. So q cannot be a breakpoint.
- If $p_x < q_x = r_x$. Then $q_y < r_y$. Then f_S has a jump at $x = r_x$ that starts at $y = q_y$. But then, for $x_0 \in (p_x, r_x)$, the interpolated value y_0 of p and r at x_0 would fulfill $f_S(x_0) < y_0$. That contradicts the definition of f_S. So q cannot be a breakpoint of f_S.
- If $p_x = q_x < r_x$. Then $p_y < q_y$. Then f_S has a jump at $x = p_x$ that ends at $y = q_y$. But then, for $x_0 \in (p_x, r_x)$, the interpolated value y_0 of p and r at x_0 would fulfill $f_S(x_0) > y_0$. That contradicts the definition of f_S. So q cannot be a breakpoint of f_S.

So, in all these cases, q is not a breakpoint of f_S. □

We define a symbol for the sequence of characteristic support points of a piecewise linear function.

Definition 2.17 (Characteristic Support Sequence)**.** *Let f be a piecewise linear function. Then the support sequence $B(f)$, that contains all pairwise distinct breakpoints of f, or, if f has no breakpoints, one point $(0, f(0))$, is its* characteristic support sequence*.*

The characteristic sequence of a linear function together with the implicit slopes m_l and m_r are necessary and enough to uniquely define it, as we show in the following.

Lemma 2.18 (Equality of Piecewise Linear Functions)**.** *Let f and g be piecewise linear functions. Then $B(f) = B(g) \wedge m_l(f) = m_l(g) \wedge m_r(f) = m_r(g) \iff \forall \tau \in \mathbb{R} : f(\tau) = g(\tau)$.*

Proof. Assume, $B(f) = B(g) \wedge m_l(f) = m_l(g) \wedge m_r(f) = m_r(g)$.

- If f and g are nonlinear, then the breakpoints and the implicit slopes uniquely define the linear segments of f and g. So $\forall \tau \in \mathbb{R} : f(\tau) = g(\tau)$.
- If f and g are linear, then they are equal if they are equal in one point and have the same slope. This is fulfilled, so $\forall \tau \in \mathbb{R} : f(\tau) = g(\tau)$.
- If f is linear but g is nonlinear (or the other way around), then $B(f) \neq B(g)$, $m_l(f) \neq m_l(g)$ or $m_r(f) \neq m_r(g)$ — a contradiction.

So, in all cases, it holds that $B(f) = B(g) \wedge m_l(f) = m_l(g) \wedge m_r(f) = m_r(g) \Rightarrow \forall \tau \in \mathbb{R} : f(\tau) = g(\tau)$

Let $\forall \tau \in \mathbb{R} : f(\tau) = g(\tau)$. Then $m_l(f) = m_l(g) \wedge m_r(f) = m_r(g)$.

Assume, $B(f) \neq B(g)$. Because equality is symmetric, we can assume that $\exists p := (x_p, y_p) \in B(f) \setminus B(g)$. Then f has a break at $x = x_0$. Since $B(f)$ and $B(g)$ are of countable cardinality, $\exists \epsilon > 0 : \forall x_0 \in [x_p - \epsilon, x_p + \epsilon] : g does not have a break at x = x_0$.

But then, either $f(x_p - \epsilon/2) \neq g(x_p - \epsilon/2)$ or $f(x_p + \epsilon/2) \neq g(x_p + \epsilon/2)$ — a contradiction. So, $B(f) = B(g)$.

So, it holds that $\forall \tau \in \mathbb{R} : f(\tau) = g(\tau) \Rightarrow B(f) = B(g) \wedge m_l(f) = m_l(g) \wedge m_r(f) = m_r(g)$. ☐

We can define a support sequence without redundant points as follows.

Definition 2.19 (Minimal Support Sequence)**.** *Let S be a support sequence and m_l and m_r implicit slopes. Then S is minimal, if for all support sequences T such that $|T| < |S|$: $f_S \neq f_T$.*

And we observe, that the characteristic support sequence of a piecewise linear function is a minimal support sequence.

Lemma 2.20 (The Characteristic Support Sequence is Minimal)**.** *Let f be a piecewise linear function, $B(f)$ its characteristic support sequence and $m_l := m_l(f)$ and $m_r := m_r(f)$. Then $B(f)$ is a minimal support sequence.*

Proof. Assume there is a support sequence T such that $|T| < |B(f)|$ and $f = f_T$. But then, by Lemma 2.16, $|B(f_T)| < |B(f)|$. And therefore, by Lemma 2.18, $f \neq f_T$. ☐

Above, we talk about storing weight functions in travel time representation without defining it, so we do that here. We keep in mind that travel times are non-negative. Travel time functions are then defined as follows.

Definition 2.21 (Travel Time Function)**.** *Let $G = (V, E)$ be a graph. Let $e \in E$ be an edge. The* travel time function *$TT(e)$ is a function $TT(e) : \mathbb{R} \to \mathbb{R}_{\geq 0}$ that maps from the departure time τ on e to the travel time $TT(e)(\tau)$ along e.*

While our input comes in travel time representation, there is also the arrival time representation. It is more convenient to define and execute the link operation that we introduce in Section 2.2. So in most parts of this work, we use the arrival time representation.

Definition 2.22 (Arrival Time Function)**.** *Let $G = (V, E)$ be a graph. Let $e \in E$ be an edge. The* arrival time function *$AT(e)$ is a function $AT(e) : \mathbb{R} \to \mathbb{R}$ that maps from the departure time τ on e to the arrival time $AT(e)(\tau)$.*

Our arrival time functions are piecewise linear with a finite set of breakpoints, and implicit slopes $m_l(f) = m_r(f) = 1$.

Travel time functions can be translated to arrival time functions and back by adding and subtracting the identity function. Since travel times are non-negative, we get the following lemma for arrival time functions.

Lemma 2.23 (Arrival Time Functions do not Decrease the Input)**.** *Let f be an arrival time function. For all $\tau \in \mathbb{R} : \tau \leq f(\tau)$.*

Proof. Let $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ be an arrival time function and $g := f - id$ the corresponding arrival time function. Let $\tau \in \mathbb{R}$. Then $f(\tau) = g(\tau) + id(\tau) \geq id(\tau) = \tau$. □

We only work with arrival time functions that respect the first-in first-out (FIFO) property in this work.

Definition 2.24 (FIFO)**.** *An arrival time function f is called FIFO, if it is weakly monotonous increasing. A travel time function f is called FIFO, if the corresponding arrival time function $f + id$ is FIFO.*

The FIFO property means that waiting at a node never leads to an earlier arrival time at an adjacent node. If our edge weights were not FIFO, the problem of finding a shortest path in a graph would be at least NP-Hard. This can be shown by reducing from the Subset-Sum problem [Dea04].

Each weight function f in our input maps from the time of day in seconds to the arrival time in seconds and fulfills certain properties. Its support sequence $S := \{(x_1, y_1), \dots, (x_n, y_n)\}$ consists of an uneven amount of support points of which the first one is always at $x = 0$ and all others have x-values in $[0, 86400)$. The linear pieces of f alternate between having an ascend of 1 and being a traffic change.

Definition 2.25 (Traffic Change)**.** *Let f be a piecewise linear arrival time function and L its set of linear pieces. A linear piece $l \in L$ is a traffic change, if its constant ascend l' is not equal to one.*

In our input graphs, weight functions only have traffic changes that are shorter than 15 minutes and that end on a quarter of an hour. We call functions in our input graphs *input functions.*

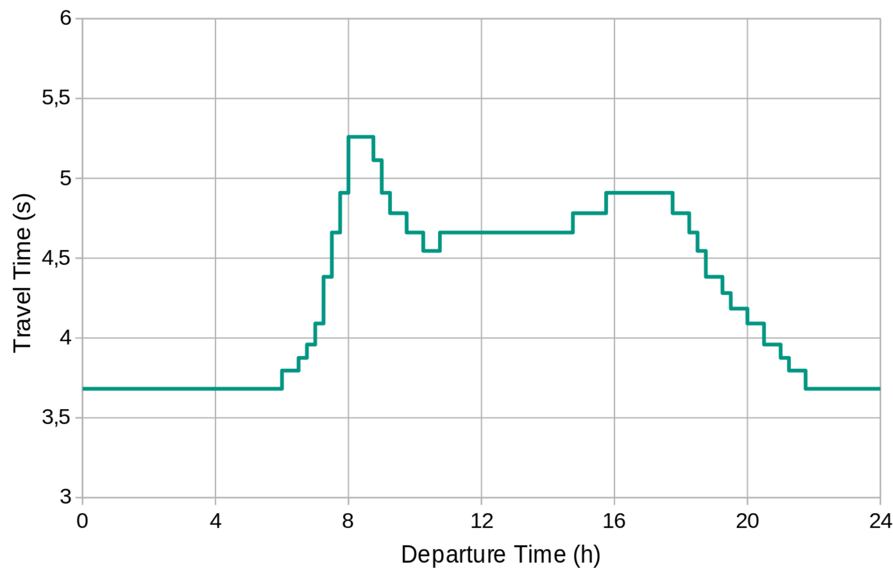


Figure 2.2.: An exemplary travel time function.

2.1.2. Interpretation of Input Data

The source of our data is the company *PTV Group*. They provided travel time prediction data for Western Europe in 2006 and 2017. The data from 2017 is the data we actually work with, but we include the data from 2006 for comparability with older publications. We know by word of mouth that they aggregated this data from different sources, but we do not know what these sources are and how the aggregation works. But we assume that the data matches current industry standards. An exemplary travel time function for a street segment in Luxembourg is depicted in Figure 2.2. The morning and evening rush hour is well visible.

The data has certain inaccuracies. It holds that every traffic change ends on a time point that is divided by 15 minutes. And every traffic change is at most 15 minutes long. This gives us the hint that the data's resolution is 15-minutes for departure times.

Furthermore, this kind of data about street networks is only a very rough estimate of reality. When looking at real GPS traces of cars, the travel times usually scatter a lot around the average for a certain point in time. One can imagine that, especially in cities, there are many events that can occur during a ride that alter travel times. For example, a driver might need to wait for pedestrians to cross, or wait at a crossing for other cars, or might be stuck behind two trucks overtaking each other. This all is noise that is introduced in the real world, and our data contains the average of this noise depending on time. But this means, that if we introduce noise into the input data, as long as it is not dominating all other sources of noise, it does not make our routes more inaccurate in the real world than they already are. We assume, that input data is shipped with random independent noise, that gives us a certain leeway for its interpretation.

We alter the interpretation of the data by restricting slopes of the input arrival time functions to zero, one and infinity. Since traffic changes are rounded to 15 minutes, this should not introduce relevant errors. Then, we allow the traffic changes to change position in terms of departure time within their 15 minute buckets. That means, it has to start at most 15 minutes before its original end, and not end after its original end, and it must still be consecutive with a slope of zero or infinity. This might introduce small notable errors for short queries. But, as long as the errors introduced on different street segments are small and independent, they should be hidden by the noise in the input data.

2.2. The Exact TDCCH

A TDCCH is a time-dependent version of the CCH [DSW16], where the scalar edge weights are replaced by weight functions that depend on time. In Section 2.1, we introduced travel time and arrival time representations for those weight functions. For usage in a TDCCH, we need the evaluation, and the link and merge operations for the weight functions, as explained below. For evaluation and merging, none of the two representations yields special advantages. But linking is easier to express with arrival time functions, so we choose this representation for the remainder of this section. We represent the arrival time functions as support sequences, and we set their implicit slopes to 1.

In the remainder of this section, we describe how to build an exact TDCCH in general in Subsection 2.2.1. We explain how to evaluate, link and merge exact arrival time functions based on their support sequence in Sections 2.2.2 to 2.2.4. We discuss the problems that arise from the exact approach in Subsection 2.2.5.

2.2.1. Building an Exact TDCCH

Building a TDCCH for an input graph consists of two steps. The first is building the topology and the second is customizing the topology with a given input metric. The advantage of a TDCCH is that the search space for querying can be restricted a lot. Below, we describe the preprocessing, customization and query.

Preprocessing

Given an undirected input graph $G = (V, E)$, the topology is built by contracting nodes out of the graph in a nested dissection order. A nested dissection order is built by recursively separating G in two node induced subgraphs $G_A := (V_A, E_A)$, $G_B := (V_B, E_B)$ and a separator $S \subset V$.

Definition 2.26 (Node Induced Subgraph)**.** *Let $G = (V, E)$ be a graph. Let $V' \subseteq V$. Then $G|'_V := (V', E' := \{\{u, v\} \in E | u, v \in V'\})$ is the* node induced subgraph *of V' of G.*

Definition 2.27 (Separator)**.** *Let $G = (V, E)$ be a graph. Let $S \subseteq V$. Let $V_A \subseteq V$ and $V_B \subseteq V$ such that S, V_A and V_B are pairwise disjoint and $S \cup V_A \cup V_B = V$. If there is no edge between V_A and V_B in G, then S is a* separator *in G.*

The nodes in the top level separator S are the highest in the nested dissection order. The nodes in the separators of G_A and G_B are the second highest, and so on.

Nodes are contracted out of the graph starting with the lowest until the graph is empty. Contracting a node n out of a graph G means removing the node and adding shortcuts for all length-2-paths that have n as middle node.

Definition 2.28 (Node Contraction)**.** *Let $G = (V, E)$ be a graph. Let $n \in V$ be a node. Then $G - n := (V \setminus \{n\}, \{\{u, v\} \in E | u \neq n \neq v\} \cup \{\{u, v\} | \{u, n\}, \{n, v\} \in E\})$ is the result of* contracting *n out of G.*

All shortcuts are added to G. The result $T := (V, E_T) := G + \textsc{shortcuts}$ is the CCH-topology of G.

Customization

Given a metric $m : \overrightarrow{E} \to F$ for the directed expansion of the undirected input graph G, with F being a set of weights, G's CCH-topology T can be customized.

Definition 2.29 (Directed Expansion)**.** *Let $G := (V, E)$ be an undirected graph. We define $\overrightarrow{G} := (V, \overrightarrow{E} := \{(u, v), (v, u) \mid \{u, v\} \in E\})$ as its* directed expansion*.*

For that, the metric is expanded to $\overrightarrow{E}_T$ by defining $m(e) := \perp$ for $e \in \overrightarrow{E}_T \setminus \overrightarrow{E}$. In the case of a TDCCH, the weights are piecewise linear arrival time functions.

The metric m is modified by traversing the nodes of T in the same nested dissection order as in the preprocessing. For a node n, all outgoing edges e with a higher end node are iterated. For each e, all lower triangles (e_0, e_1, e) are enumerated.

Definition 2.30 (Lower Triangle)**.** *Let $G = (V, \overrightarrow{E})$ be a graph. Let $e := (u, w) \in \overrightarrow{E}$. Let $v \in V$. We define a* lower triangle *$(e_0 := (u, v) \in \overrightarrow{E}, e_1 := (v, w) \in \overrightarrow{E}, e)$ of e as a tuple of edges such that $v < u \wedge v < w$ in the nested dissection order.*

For each lower triangle (e_0, e_1, e), the weights $m(e_0)$ and $m(e_1)$ are linked. The weights are never $\perp$ because of the order the nodes, edges and lower triangles are processed. The link describes first driving over edge e_0 and then driving over edge e_1. For piecewise linear arrival time functions, it is defined as follows.

Definition 2.31 (Linking)**.** *Let f and g be weakly increasing piecewise linear functions. Then the* link *of f and g is the function $f \oplus g$ defined as:*

$$\forall \tau \in \mathbb{R} : (f \oplus g)(\tau) = g(f(\tau))$$

We try to reduce the memory usage of a TDCCH, so we introduce another term that refers to a link with minimal support sequence.

Definition 2.32 (Minimal Link)**.** *Let S and T be support sequences such that their derived piecewise linear functions f_S and f_T are weakly increasing. Let $S \oplus_m T$ be a minimal support sequence such that $\forall \tau \in \mathbb{R} : f_{S \oplus_m T}(\tau) = (f_S \oplus f_T)(\tau)$. Then $S \oplus_m T$ is called the* minimal link *of S and T.*

The minimal link $l := m(e_0) \oplus m(e_1)$ is stored in $m(e)$. But $m(e)$ might already store a weight. In this case, there are at least two alternatives of paths that e can represent. Since we are searching for shortest paths, we take the minimum of those alternatives and store it in $m(e)$.

Definition 2.33 (Merging)**.** *Let f and g be piecewise linear functions. Then the* merge *of f and g is the function $\min_m\{f, g\}$ defined as:*

$$\forall \tau \in \mathbb{R} : \min_m\{f, g\}(\tau) = \min\{f(\tau), g(\tau)\}$$

For the merge operation we introduce a term for a merge with minimal support sequence as well.

Definition 2.34 (Minimal Merge)**.** *Let S and T be support sequences and f_S and f_T their derived piecewise linear functions. Let $\min_m\{S, T\}$ be a minimal support sequence such that $\forall \tau \in \mathbb{R} : f_{\min_m\{S,T\}}(\tau) = \min\{f_S, f_T\}(\tau)$. Then $\min_m\{S, T\}$ is called the* minimal merge *of S and T.*

Linking and merging of piecewise linear functions are expensive operations compared to the addition and minimization of ordinals. So before we customize the TDCCH with weight functions, we customize the CCH one time with the lower bounds of all functions and one time with the upper bounds. The bounds are taken in travel time representation. Then, we only link $m(e_0)$ and $m(e_1)$ if the sum of their lower bounds is lower or equal to the precomputed upper bound of $m(e)$. And we only take the minimum, if the bound intervals of the weight functions intersect. We update the precomputed upper bound of $m(e)$ if the minimum was taken.

Query

Building the TDCCH as described above yields a graph G that fulfills the CCH-invariant.

Definition 2.35 (CCH-Invariant)**.** *Let $G = (V := (v_1, \ldots, v_n), E)$ be an edge-weighted graph with ordered nodes. Let $v_a, v_b \in V$ be a pair of nodes. Let l be the length of the shortest path from v_a to v_b in G. Let $V' := (v_k, \ldots, v_n) \subseteq V$ be a subsequence of V such that $k \leq \min\{a, b\}$. Let G' be the node induced subgraph of V' of G. Let l' be the length of the shortest path from v_a to v_b in G'.*

If $l = l'$ for all choices of v_a, v_b and V', then G is defined as fulfilling the CCH-invariant*.*

A TDCCH is queried with a bidirectional Dijkstra [Dij59]. The CCH-invariant allows us to prune all edges incident to a node that lead to lower nodes in the nested dissection order. This pruned search is called *upward search*. The upward search is done with forward edges from the departure node, and with backward edges from the destination node.

When having time-dependent weights, a problem arises. It is not possible to do the backward search from the destination node, because the arrival time is unknown. This makes it impossible to evaluate the weights of the backward edges correctly.

A solution to that is enumerating all edges than can be reached by a backward upward search from the destination node. Then Dijkstra is run on the forward and the backward search space combined.

In a CCH, the upward search can be run without queue. This is called an elimination tree query.

Definition 2.36 (Elimination Tree)**.** *Let $G = (V, E)$ be an undirected graph with ordered nodes. Let $T = (V, E_T)$ be the CCH-topology of G. For each $v \in V$, let* $\text{parent}(v) := \min\{w | \{v, w\} \in E_T, w > v\}$*. Then, the* elimination tree *of T is defined as the tree $L := (V, \{\{u, v\} | v = \text{parent}(u)\})$.*

Walking the elimination tree upwards from a node $n \in V$ visits exactly the nodes in the upward search space of n [DSW16]. For a node $n \in V$, the parent in the elimination tree is the lowest neighbor that is higher than n. So when doing the upward query, instead of fetching the next node from the queue, the next node is the parent in the elimination tree. This way, the whole upward search space of n is traversed.

For the downward search to the destination node, first, the upward facing outgoing edges for nodes in the upward search space are enumerated. Then, after doing the upward forward search from the departure node, the reverses of the enumerated edges are relaxed in reverse order.

When relaxing an edge e from node u to node v, we evaluate its weight function $m(e)$ conditionally based on its lower bound. If traversing e from u cannot decrease the length of the shortest known path to v, then we skip evaluating $m(e)$. If we evaluate $m(e)$, we use binary search to find the correct pair of support points for interpolation.

Algorithm 2.1: EVALUATE

Input: Support sequence $S := \{(x_1, y_1), \ldots, (x_n, y_n)\}$, implicit slopes m_l and m_r and some $\tau \in \mathbb{R}$

Output: $f_S(\tau)$ with f_S having implicit slopes m_l and m_r

```
1 if τ ≤ x_1 then
2     return y_1 − m_l · (x_1 − τ)
3 if τ > x_n then
4     return y_n + m_r · (τ − x_n)
5 foreach Pair of points ((x_i, y_i), (x_{i+1}, y_{i+1})) ∈ S do
6     if x_i ≤ τ ≤ x_{i+1} then
7         return (x_{i+1} − τ) · y_i + (τ − x_i) · y_{i+1}
```

Algorithm 2.2: EVALUATEINV

Input: Support sequence $S := \{(x_1, y_1), \ldots, (x_n, y_n)\}$, implicit slopes m_l and m_r and some $\tau \in \mathbb{R}$

Output: $f_S^{-1}(\tau)$ with f_S having implicit slopes m_l and m_r

```
1 if τ < y_1 then
2     return x_1 − (y_1 − τ)/m_l
3 if τ ≥ y_n then
4     return x_n + (τ − y_n)/m_r
5 foreach Pair of points ((x_i, y_i), (x_{i+1}, y_{i+1})) ∈ S in reverse do
6     if y_i ≤ τ ≤ y_{i+1} then
7         return (y_{i+1} − τ) · x_i + (τ − y_i) · x_{i+1}
```

2.2.2. Exact Evaluation

The evaluation of piecewise linear functions represented as support sequences is formalized in Algorithm 2.1. Note that it evaluates the derived function left-continuous, conforming to Definition 2.11.

Algorithm 2.2 evaluates the inverse of a piecewise linear function f_S with support sequence S at a point τ. A piecewise linear function f and its inverse f^{-1} have the can be represented by the same support points, but with swapped coordinates. If f_S is left-continuous, then f^{-1} is right-continuous. Since we never need the inverse of a function itself, we never calculate it.

The correctness of the algorithms is easy to see by comparing them to the definitions of interpolation and the derived piecewise linear function. The relevant definitions are Definition 2.8, 2.9 and 2.11.

Algorithm 2.1 and 2.2 run in linear time. It is also possible to use binary search for finding these consecutive supports to get the result in logarithmic time.

With the functions represented exactly with supports, the link and merge operations can also be implemented exactly as explained in the two subsections below.

2.2.3. Exact Linking

Algorithm 2.3 takes two support sequences S and T and produces their link $S \oplus T$. The algorithm is applied on two exemplary support sequences in Figure 2.3. It treats the

Algorithm 2.3: LINK

Input: Support sequences S and T such that f_S and f_T are arrival time functions
Output: Support sequence L such that $f_L = f_S \oplus f_T$

```
1  Let S and T be iterators over their points
2  while (a, b) ← S.CURRENT() ∧ (c, d) ← T.CURRENT() do
3      if b ≤ c then
4          L.INSERT((a, f_T(b)))
5          ADVANCE(S)
6      else
7          L.INSERT((f_S^{-1}(c), d))
8          ADVANCE(T)
9  while (a, b) ← S.CURRENT() do
10     L.INSERT((a, f_T(b)))
11     ADVANCE(S)
12 while (c, d) ← T.CURRENT() do
13     L.INSERT((f_S^{-1}(c), d))
14     ADVANCE(T)
```

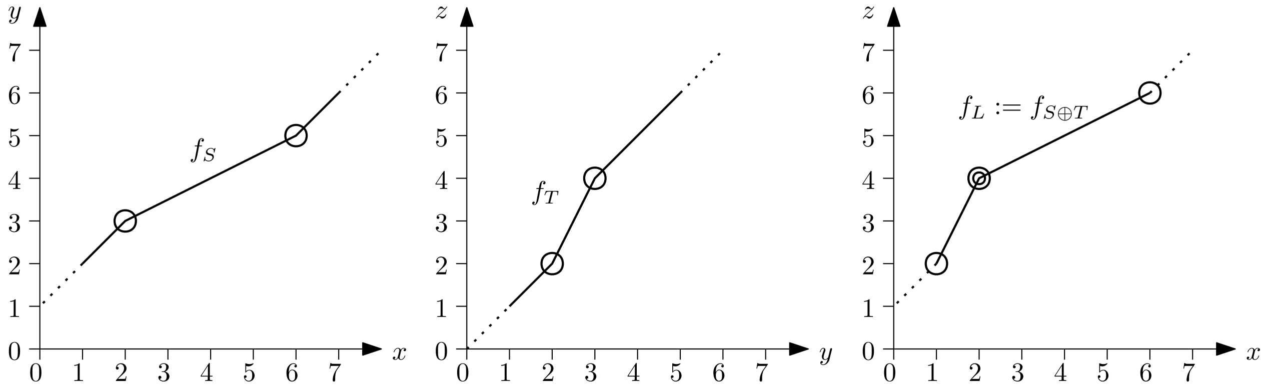


Figure 2.3.: Link of two exemplary support sequences. The support points are marked by circles. Support point $(2, 3) \in S$ produces the support point $(2, f_T(3)) = (2, 4) \in L$. Support point $(2, 2) \in T$ produces the support point $(f_S^{-1}(2), 2) = (1, 2) \in L$. The other support points are handled accordingly.

support sequences as iterators for easier notation. The CURRENT method copies the point that is at the current position of the iterator. The assignment from a call to CURRENT returns true if the iterator points into the support sequence, and false if it ran over the end of the sequence. The lists are sorted lexicographic first by x-value, then by y-value.

We say a point p in a link support sequence LINK(S, T) *originates* from $p_S \in S$, if it is added in Line 4 or 10. We say a point p in a link support sequence LINK(S, T) *originates* from $p_T \in T$, if it is added in Line 7 or 13. If a point p in a link $f_S \oplus f_T$ originates from a point q in S or T, then we say that q *produced* p.

Note that this algorithm does not ensure that the output is a minimal link. Algorithm 2.4 takes a support sequence and minimizes it.

Both the link and the clean algorithm run in linear time. The function evaluations in link run in constant time, since only the current and the last point of the corresponding iterator

Algorithm 2.4: CLEAN

```
   Input: Support sequence S that represents an arrival time function
   Data: Support sequence T, a slope s
   Output: The characteristic support sequence U of f_S
   // Remove redundant supports
 1 foreach Point p := (x, y) ∈ S do
 2     if |T| > 0 then
 3         in a link f ⊕ g if T.LAST() ≠ p then
 4             T.INSERT(p)
 5     else
 6         T.INSERT(p)
   // Remove consecutive collinear triples
 7 foreach Pair of points (p_1, p_2) ∈ T do
 8     s ← SLOPE(p_1, p_2)
 9     if |U| > 0 then
10         if ¬ISCOLLINEAR(U.LAST(), p_1, p_2) then
11             U.INSERT(p_1)
12     else if s ≠ 1 then
13         U.INSERT(p_1)
14 if |U| = 0 ∨ (|T| ≥ 2 ∧ s ≠ 1) then
15     U.INSERT(T.LAST())
16 if |U| = 1 then
17     U[0] = (0, f_U(0))
```

are required for evaluation. If there is no last point, then the current point is interpolated with ascend 1, as defined in Definition 2.9.

In the following we prove the correctness of the LINK and CLEAN algorithm. We argue for the correctness of LINK by arguing that it produces a function with the correct breakpoints. We start with proving that LINK produces a weakly increasing support sequence.

Lemma 2.37 (LINK Produces a Weakly Increasing Support Sequence)**.** *The output of* LINK *is a weakly increasing support sequence.*

Proof. Let S and T be support sequences as required by the algorithm.

The output $L := \text{LINK}(S, T)$ is a sequence of points. By Definition 2.6, L is a support sequence, if its support points are sorted lexicographic by first x and then y. And it is weakly increasing, if the support points are sorted by x and y independently.

Let $<$ be the partial order on support points ordering them by x and y independently. The input support sequences S and T are sorted by $<$. The result L is sorted by $<$, if for each pair of distinct points $p < q \in L$ it holds that p is added before q.

In the loop, the algorithm is processing the points $(a, b) \in S$ and $(c, d) \in T$. It chooses between adding $s := (a, f_T(b))$ or $t := (f_S^{-1}(c), d)$ to U. In later iterations, since S and T are sorted, and f_S and f_T are weakly increasing, only points that are not smaller than s and t will be added. So it is enough to prove that the s or t added in an iteration is a locally smallest.

- If $b \leq c$, s is added first. Then $a \leq f_S^{-1}(b) \leq f_S^{-1}(c)$. The first inequality holds, because $(a, b) \in S$ and f_S^{-1} is right-continuous. The second inequality holds, because f_S is weakly increasing.

  Also, $f_T(b) \leq f_T(c) \leq d$. The first inequality holds, because f_T is weakly increasing. The second inequality holds, because $(c, d) \in T$ and f_T is left-continuous. So s is a locally smallest point.

- If $c < b$, t is added first. Since $(a, b) \in S$ with $c < b$ and f_S weakly increasing, $f_S^{-1}(c) \leq a$. Since $(c, d) \in T$ with $c < b$ and f_T weakly increasing, $f_T(b) \geq d$. So t is a locally smallest point.

Since the support points from S and T are sorted correctly, and the algorithm merges the points in the correct order, the output L is a weakly increasing support sequence. □

Below, we prove a lemma similar to Lemma 2.1 from [FHS11]. This strongly restricts the space the breakpoints of h can originate from.

Lemma 2.38 (Breakpoints Originate from Support Points)**.** *Let f, g be weakly increasing piecewise linear functions. Let $h := f \oplus g$ be their link. If h has a breakpoint (a, d), then either f has a breakpoint at a or g^{-1} has a breakpoint at d.*

Proof. Assume, h has a breakpoint (a, d), such that f is linear around a, and g^{-1} is linear around d.

- If g^{-1} is not constant around d, then $g(f(a)) = h(a)$ is linear around a, so (a, d) is not a breakpoint in h.
- If f is not constant around a, then $f^{-1}(g^{-1}(d)) = h^{-1}(d)$ is linear around d, so (a, d) is not a breakpoint in h.
- Assume, g^{-1} is constant around d and f is constant around a. Then g has a jump at $f(a)$ such that $g(f(a)) < d$. And since f is constant around a, $g(f(a)) < d$ around a. But then, $h(a) < d$ around a, so (a, d) is not on the function trace of h - a contradiction.

□

With this we can prove that each breakpoint in h is generated by the LINK-algorithm.

Note that for a breakpoint (a, d) to exist in h it is not necessary that $h(a) = d$ or $h^{-1}(d) = a$. When we say that a point (a, d) is *not on* h, we mean that it is not on the function trace of h.

We define two special subsequences of a support sequence to easier express the results of the following lemma and later results.

Definition 2.39 (Sub Support Sequence)**.** *Let S be a weakly increasing support sequence. Then for $x_0 \in \mathbb{R}$, $S_{x=x_0} := ((x, y) \in S | x = x_0)$. And for $y_0 \in \mathbb{R}$, $S_{y=y_0} := ((x, y) \in S | y = y_0)$. Both have the same order as S. We call $S_{x=x_0}$ and $S_{y=y_0}$* sub support sequences*.*

Lemma 2.40 (Each Breakpoint in h is Generated by the LINK-Algorithm)**.** *Let S and T be support sequences such that f_S and f_T are arrival time functions. Let $h := f_S \oplus f_T$ be their link. Let $L :=$ LINK(S, T) be the result of Algorithm 2.3 applied to S and T. If h has a breakpoint (a, d), then $(a, d) \in L$.*

Proof. Let h have a breakpoint (a, d). Then by Lemma 2.38, f_S has a breakpoint at a or f_T^{-1} has a breakpoint at d.

Let $((a, b_1), \ldots, (a, b_r)) := S_{x=a}$. Let $((c_1, d), \ldots, (c_s, d)) := T_{y=d}$.

- If only S has support points at $x = a$. Then f_T^{-1} is linear around d. Let $c := f_T^{-1}(d)$.
  - If $c \notin [b_1, b_r]$, then (a, d) is not on h.
  - If $c \in (b_1, b_r)$, then f_S^{-1} is linear around c. So $f_T^{-1} \oplus f_S^{-1}$ is linear around d. But then, h cannot have a breakpoint at $y = d$.
  - If $c = b_1 \vee c = b_r$.
    - If $f_T(c) = d$, then (a, d) originates from (a, b_1) or (a, b_r).
    - If $f_T(c) < d$. Then f_T^{-1} is constant around d. Then, $f_T^{-1} \oplus f_S^{-1}$ is constant around d, so h cannot have a breakpoint at $y = d$.
- If only T has support points at $y = d$. Then f_S is linear around a. Let $b := f_S(a)$.
  - If $b \notin [c_1, c_s]$, then (a, d) is not on h.
  - If $b \in (c_1, c_s)$, then f_T is linear around b. So $f_S \oplus f_T$ is linear around a. But then, h cannot have a breakpoint at $x = a$.
  - If $b = c_1 \vee b = c_s$.
    - If $f_S^{-1}(b) = a$, then (a, d) originates from (c_1, d) or (c_s, d).
    - If $f_S^{-1}(b) > a$. Then f_S is constant around a. Then, $f_S \oplus f_T$ is constant around a, so h cannot have a breakpoint at $x = a$.
- If S has at least one support point at $x = a$ and T has at least one support point at $y = d$.
  - If $[b_1, b_r] \cap [c_1, c_s] = \emptyset$, then (a, d) is not on h.
  - If $b_1 \in (c_1, c_s]$, then $f_T(b_1) = d$, so (a, d) originates from (a, b_1).
  - If $b_r \in (c_1, c_s]$, then $f_T(b_r) = d$, so (a, d) originates from (a, b_r).
  - If $c_1 \in [b_1, b_r)$, then $f_S^{-1}(c_1) = a$, so (a, d) originates from (c_1, d).
  - If $c_s \in [b_1, b_r)$, then $f_S^{-1}(c_s) = a$, so (a, d) originates from (c_s, d).
  - If $b_r = c_1$.
    - If $f_T(b_r) = d$, then (a, d) originates from (a, b_r).
    - If $f_S^{-1}(c_1) = a$, then (a, d) originates from (c_1, d).
    - Else, $f_T(b_r) =: d_0 < d \wedge f_S^{-1}(c_1) =: a_1 > a$. But then f_S is constant on $(a, a_1]$ with value b_r. So

      $$\lim_{\epsilon \to 0} h(a + \epsilon) = \lim_{\epsilon \to 0} f_T(f_S(a + \epsilon)) = f_T(b_r) = d_0 < d.$$

      And

      $$h(a) = f_T(f_S(a)) = f_T(b_r) = d_0 < d.$$

      So (a, d) is not on the function trace of h.

So in each case, if (a, d) is on the function trace of h, then $(a, d) \in L$. □

By Lemma 2.40, Algorithm 2.3 outputs a function that has all necessary breakpoints. So for its correctness, we need to make sure, that it does not add any breakpoints besides these.

Lemma 2.41 (LINK does not Add Wrong Breakpoints)**.** *Let S and T be support sequences such that f_S and f_T are arrival time functions. Let $h := f_S \oplus f_T$ be their link. Let $L := \text{LINK}(S, T)$ be the result of Algorithm 2.3 applied to S and T. Let $p \in L$ such that it is not a breakpoint in h. Then p is on a linear piece of h, including jumps.*

Proof. Let $p := (a, d) \in L$ such that it is not a breakpoint of h. Let $q_0 := (a_0, d_0)$ and $q_1 := (a_1, d_1)$ be the first breakpoint before p and the first breakpoint after p of h.

If q_0 or q_1 does not exist, then add a ghost point left or right of p that obeys the implicit slope of h and call it a breakpoint for the sake of this proof, even though it is not. Every time we use the breakpoint property on a ghost point, the result is the same.

We show that p is on the line between the breakpoints of h before and after it.

- If $a_0 = a_1$, since by Lemma 2.37, L is sorted, $a = a_0$. And since q_0 and q_1 are breakpoints in h but p is not, $d \in (d_0, d_1)$. So p is on the line between q_0 and q_1.
- If $d_0 = d_1$, since by Lemma 2.37, L is sorted, $d = d_0$. And since q_0 and q_1 are breakpoints in h^{-1} but p is not, $a \in (a_0, a_1)$. So p is on the line between q_0 and q_1.
- If $a_0 < a_1$ and $d_0 < d_1$, then h is linear on (a_0, a_1) and $a \in (a_0, a_1)$. And h^{-1} is linear on (d_0, d_1) and $d \in (d_0, d_1)$.
  - If p originates from (a, b) with $f_T(b) = d$, then $f_S(a) \leq b$.
    - If $f_S(a) = b$, then $d = f_T(f_S(a)) = h(a)$. So p is on the line between q_0 and q_1.
    - If $f_S(a) < b$, then f_S has a jump at a from b_0 to b_1 such that $b \in (b_0, b_1]$. But h does not have a jump at a. So f_T needs to have a constant at b_0 to at least b_1 with value $f_T(b) = d$. Also, f_T does not have a jump at $f_S(a) = b_0$. So, $f_T(b_0) = f_T(b) = d$.

      But then, $h(a) = f_T(f_S(a)) = f_T(b_0) = d$. So p is on the line between q_0 and q_1.
  - If p originates from (c, d) with $f_S^{-1}(c) = a$, then $f_T^{-1}(d) \geq c$.
    - If $f_T^{-1}(d) = c$, then $a = f_S^{-1}(f_T^{-1}(d)) = h^{-1}(d)$. So p is on the line between q_0 and q_1.
    - If $f_T^{-1}(d) > c$, then f_T^{-1} has a jump at d from c_0 to c_1 such that $c \in [c_0, c_1)$. But h^{-1} does not have a jump at d. So f_S^{-1} needs to have a constant at c_0 to at least c_1 with value $f_S^{-1}(c) = a$. Also, f_S^{-1} does not have a jump at $f_T^{-1}(d) = c_1$. So, $f_S^{-1}(c_1) = f_S^{-1}(c) = a$.

      But then, $h^{-1}(d) = f_S^{-1}(f_T^{-1}(d)) = f_S^{-1}(c_1) = a$. So p is on the line between q_0 and q_1.

So since p is on the line between the breakpoints in h before and after it, it is on a linear piece of h, including jumps. ☐

Using the lemmas above, we prove the correctness of Algorithm 2.3.

Theorem 2.42 (Correctness of LINK)**.** *Let S and T be support sequences such that f_S and f_T are arrival time functions. Let $h := f_S \oplus f_T$ be their link. Let $L :=$ LINK(S, T) the result of Algorithm 2.3 applied to S and T. Then $\forall \tau : h(\tau) = f_L(\tau)$.*

Proof. Since f_S and f_T are piecewise linear functions, their link h is also piecewise linear. By Lemma 2.37, L is a support sequence.

By Lemma 2.40, each breakpoint in h is also a support point in L. And by Lemma 2.41, each support point $p \in L$, that is not a breakpoint in h, is on a linear piece of h, including jumps. So $B(h) = B(f_L)$. Also, since the implicit slopes of arrival time functions are always 1, they are invariant under the LINK algorithm. So, by Lemma 2.18, $\forall \tau \in \mathbb{R} : h(\tau) = f_L(\tau)$. ☐

Below we prove the correctness of Algorithm 2.4.

Theorem 2.43 (Correctness of CLEAN)**.** *Let S be a support sequence such that f_S is an arrival time function. Then $U :=$ CLEAN(S) the characteristic support sequence of f_S.*

Proof. Let T be the intermediate result calculated by CLEAN. Then f_T is equivalent to f_S, since $\forall p_S \in S : \exists p_T \in T : p_S = p_T$ and $\forall p_T \in T : \exists p_S \in S : p_T = p_S$.

The result U contains no collinear triples and either just one point or the endpoints are breakpoints. Also, U contains no duplicate points. So U is a characteristic support sequence. Also, U contains all breakpoints of f_S and no points were added, and their implicit slopes are the same, so $f_S = f_U$. ☐

With the correctness of both LINK and CLEAN, the algorithm CLEAN ∘ LINK, which is the consecutive execution of first LINK and then CLEAN, produces the minimal link support sequences of its input support sequences.

2.2.4. Exact Merging

The exact merging is done with Algorithm 2.5. The general idea is to scan the line segments and support points of both functions from left to right, such that relevant pairs are compared. The minimum of two piecewise linear functions is a piecewise linear function, that can contains support points from both functions, and support points that are created by crossing line segments. So relevant pairs that need comparison are each support point with the line segment it is above or below of or on, and each pair of line segments from different input functions that might intersect.

In the algorithm the STARTPOINT and ENDPOINT functions are used, but the first line segment of a piecewise linear function has no start point, and the last has no endpoint. If a line segment l has no startpoint, then STARTPOINT(l) behaves like $(-\infty, -\infty)$ in comparisons, in that it is smaller than everything else. If a line segment l has no endpoint, then ENDPOINT(l) behaves like (∞, ∞) in comparisons, in that it is larger than everything else. The function ISBELOWORON checks if the calling point is below or on the given line segment. A point $p := (x, y)$ is below or on a line segment $l := (p_0 := (x_0, y_0), p_1 := (x_1, y_1))$, if $x_0 \leq x \leq x_1$ and $y \leq y_x$, with y_x being the interpolated value of p_0 and p_1 at x. The function INTERSECTION returns the unique intersection point between two line segments, or $\perp$, if it does not exist.

Only one line segment is advanced per iteration, and as soon as both reach the end, the loop terminates. So in Line 5, always one of the endpoints exists, and the function ADVANCE is never called on last line segments. Also, in Line 7 and Line 13, the startpoint always exists. So the algorithm is well-defined.

Algorithm 2.5: MERGE

Input: Nonempty support sequences S and T such that f_S and f_T are arrival time functions.
Data: Line segments s and t.
Output: Support sequence M such that $f_M = \min\{f_S, f_T\}$.

```
1  s ← First line segment of f_S
2  t ← First line segment of f_T
3  e ← NIL
4  while ¬(s and t are the last line segments of f_S and f_T) do
5      if ENDPOINT(s) ≤ ENDPOINT(t) then
6          ADVANCE(s)
7          if STARTPOINT(s).ISBELOWORON(t) then
8              M.INSERT(STARTPOINT(s))
9          if INTERSECTION(s, t) ≠ ⊥ then
10             M.INSERT(INTERSECTION(s, t))
11     else
12         ADVANCE(t)
13         if STARTPOINT(t).ISBELOWORON(s) then
14             M.INSERT(STARTPOINT(t))
15         if INTERSECTION(s, t) ≠ ⊥ then
16             M.INSERT(INTERSECTION(s, t))
```

We argue shortly for the correctness of the algorithm. Since the line segments are traversed in order of their endpoints, all possible intersections are checked, except for the first and last one. But the first is between the first line segments with ascend one, and the last between the last line segments with ascend one, so they cannot intersect in a single support point. The intersections are added at the correct position. Also, each support point is compared against the correct line segment with ISBELOWORON in the correct order. So all correct support points are added to M in the right order. So $f_M = \min_m\{f_S, f_T\}$.

2.2.5. Problems with the Exact Approach

While the exact approach yields exact results for the given metric, it has a problem with memory consumption. As discussed in Chapter 4, customizing a TDCCH for the German road network with realistic data from 2017 with limited slopes takes around 202GiB of memory, while the input graph takes only 2.3GiB. Customizing a time-dependent road graph of Western Europe exactly is not possible on a machine with 252GiB of RAM. Furthermore, an exact TDCCH not only requires expensive machines with lots of RAM, but also customization time suffers from linking and merging the large support sequences higher in the hierarchy. Our exact customization takes five hours for the Germany graph, which is too slow to enable real-time updates.

3. Approximate Approach

In Chapter 2, we describe how to build an exact TDCCH, and that it leads to high memory consumption. In this chapter, we introduce a different approach to build a smaller TDCCH. The index is still exact on the given weights, but we alter the input weights a bit to reduce memory consumption. For that, we introduce the *phase model* in Section 3.1. In Section 3.2, we introduce cover potentials as a way to describe the size of a linked function. With this we propose an algorithm to calculate an optimized link based on user-defined degrees of freedom for the input functions in Section 3.3. We describe an algorithm to merge in the phase model in Section 3.4. Using the optimized link and the merge algorithm, we describe a scheme to optimize the input weights for less space consumption in an exact customization in Section 3.5.

3.1. Phase Representation

Our model does not operate on the support sequences as they are, but converts them to a different format that we call phase function. The phase function representation focuses on the traffic changes of the input function. Converting a support sequence to a phase function, we convert each traffic change to a phase. Where a support sequence is a set of support points, a phase function is a sequence of phases. Traffic changes and phases contain nearly the same information, but a phase only knows its height difference, not its absolute position on the y-axis. When interpreted as travel time function, a phase function is implicitly constant between the phases, like the input function is between traffic changes. When interpreted as arrival time function, a phase function has ascend one between the phases. We first define phases and related terms.

Definition 3.1 (Phase, Phase Interval/Start/End/Length/Height, Height Bound)**.** *A* phase *p is a tuple $([a, b], h)$, with $[a, b]$ being the (closed)* phase interval*, and h the* height *of the phase.*

- *The height of the phase is the y-difference between the start and the end of the phase in travel time representation.*
- *The opening bound a of the phase interval is called the* start *of the phase, and the closing bound b is called the* end *of the phase.*
- *The* length *of a phase p is defined as the length of the phase interval* $\text{len}(p) := b - a$.
- *A phase must fulfill the* height bound $-h \leq b - a$.

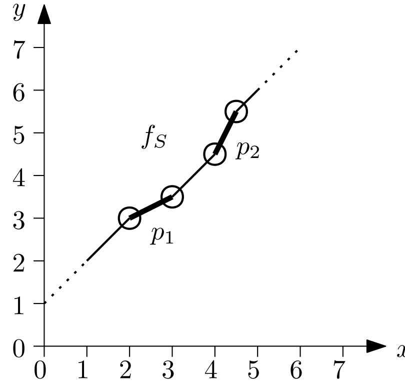


Figure 3.1.: Exemplary support sequence with highlighted phases. It contains the phases $p_1 = ([2,3], -0.5)$ and $p_2 = ([4, 4.5], 0.5)$ in arrival time representation. The support points are marked by circles.

The height bound originates from the FIFO property of our input functions. If a phase would not fulfill the height bound, it would represent a section of a travel time function that contains a descend smaller than -1, violating the FIFO property for travel time functions. Phases might have height 0, but then they would not need to be stored. We allow height 0 nevertheless, because it might appear as a result of some computation and it is well-defined. Figure 3.1 displays a piecewise linear function and its phases.

Before we define phase functions, we need terms to describe the valid relative positioning of phases towards each other. Since phases describe slopes, they cannot intersect in more than one point, because otherwise the height of the phases is not well-defined anymore.

Definition 3.2 (Intersecting/Touching/Disjoint Phases). *Let* $p_1 := ([a_1, b_1], h_1)$ *and* $p_2 := ([a_2, b_2], h_2)$ *be phases.*

- *Phases* p_1 *and* p_2 *are called* intersecting*, if* $[a_1, b_1] \cap [a_2, b_2] \neq \emptyset$.
- *Phases* p_1 *and* p_2 *are called* disjoint*, if* $[a_1, b_1] \cap [a_2, b_2] = \emptyset$.
- *Phases* p_1 *and* p_2 *are called* touching*, if* $a_1 = b_2 \vee b_1 = a_2$.

Phases have a natural ordering, that is defined as follows.

Definition 3.3 (Natural Ordering of Phases). *Let* $p_1 := ([a_1, b_1], h_1)$ *and* $p_2 := ([a_2, b_2], h_2)$ *be phases that are disjoint or touching. Then* p_1 *is* left *of* p_2*, if* $b_1 \leq a_2$ *and* p_1 *is* right *of* p_2*, if* $a_1 \geq b_2$.

Whenever we refer to sorted sequences of phases, we mean sorted by the phases' natural ordering. We define phase functions as ordered sequence of phases.

Definition 3.4 (Phase Function, Zero). *A* phase function $f : \mathbb{R} \to \mathbb{R}$ *is a tuple* (P, z) *where* P *is a sorted sequence of pairwise disjoint or touching phases and* z *the value of* $\lim_{x \to -\infty} f(x)$ *in travel time representation.* z *is also called zero.*

Definition 3.5 (Phase Function Size). *Let* $f := (P, z)$ *be a phase function. Then* $|f| := |P|$ *is its* size*.*

Phase functions are not bound to be travel time or arrival time functions, since they do not contain explicit y-values. The interpretation can be chosen on evaluation or when the phase function is converted to a different format.

To understand what a phase within a phase function is actually doing to the input space when applied, we introduce the notion of the output interval.

Definition 3.6 (Phase Output Interval)**.** *Given a phase function $f = (P, z)$, a phase $p = ([a, b], h) \in P$ and $\hat{h}$, which is the sum of z and the heights of all phases before p. Then its* phase output interval *under the phase function f is $f_{out}(p) := [a + \hat{h}, b + \hat{h} + h]$.*

The output interval can be understood as the image of the phase interval under the phase function when interpreted as arrival time function.

Having defined this representation, two questions arise. First, how do we evaluate a phase function for a certain time point τ? And second, how do we convert between the input and the phase representation?

3.1.1. Evaluation

Algorithm 3.1: Evaluate Phase Function as Arrival Time Function

Input: Phase function $f = (P, z)$, time point τ
Data: Current y-value
Output: $f(\tau)$

```
1 y ← z + τ
2 foreach Phase ([a, b], h) ∈ P do
3     if τ ≤ a then
4         return y
5     else if τ ≤ b then
          // Interpolate within phase
6         return (y · (b − τ) + (y + h) · (τ − a))/(b − a)
7     else
8         y ← y + h
9 return y
```

A phase function can be evaluated as arrival time function by sweeping over the phases, as shown in Algorithm 3.1. This is linear in the amount of phases, and can be slow for large amounts of phases.

If we do not store the height of a phase, but the y-value of the start of the phase, then we can calculate the height of a phase using its y-value and the y-value of the next phase. This enables us to use binary search for evaluation, which should result in speedups for large amounts of phases. But since we never evaluate phase functions, we do not do this.

3.1.2. Conversion

To work with phase functions, we need to be able to convert into and out of this format.

Algorithm 3.2 converts a support sequence representing an arrival time function to a phase function. Note that we loose the information about the implicit slopes in this process. But we only construct phase functions from arrival time functions, so they are always 1.

The conversion from a phase function back to a support sequence is done with Algorithm 3.3. It uses the end of the last phase e to ensure that it does not create duplicate points if phases touch. Note that the resulting support sequence represents an arrival time function.

When the algorithm processes phase p, it inserts two support points if they do not exist. We refer to those as the support points corresponding to p.

With these conversion rules, we can define all operations, that we define on support sequences, also on phase functions. The phase functions are converted to support sequences

Algorithm 3.2: phaseFunction($\cdot$)

Input: Support sequence $S := ((x_1, y_1), \dots, (x_n, y_n))$.
Data: Current y-value.
End of last phase e.
Output: Phase function $f := (P, z)$.

```
1 z ← y_1 − x_1
2 foreach ((x_i, y_i), (x_{i+1}, y_{i+1})) ⊆ S do
3     if x_{i+1} − x_i ≠ y_{i+1} − y_i then
4         P.append(([x_i, x_{i+1}], (y_{i+1} − y_i) − (x_{i+1} − x_i)))
```

Algorithm 3.3: supportSequence($\cdot$)

Input: Phase function $f := (P, z)$.
Data: Current y-value.
End of last phase e.
Output: Support Sequence S.

```
1  y ← z
2  if |P| = 0 then
3      S.append((0, y))
4  else
5      e ← −∞
6      foreach Phase ([a, b], h) ∈ P do
7          if e ≠ a then
8              S.append((a, y))
9          y ← y + h
10         if a ≠ b ∨ h ≠ 0 then
11             S.append((b, y))
12         e ← b
```

first, then the operation is applied, and then the results are converted back to phase functions, if applicable.

3.1.3. Space Requirements

Depending on the function, the phase function representation can consume more or less memory than the support sequence representation. A support sequence with n support points takes $2n$ units of memory. If phases are disjoint, then pairs of support points that take four numbers only take three in phase representation. Then, the same function in phase representation takes only $1 + 3(n-1)/2$. For n towards infinity, we get a factor of

$$\frac{1 + 3(n-1)/2}{2n} = \frac{3n-1}{4n} \xrightarrow{n \to \infty} \frac{3}{4}$$

for memory consumption.

If a phase is touching another phase on both ends, then these three phases take nine numbers, while the four points required to describe the same section in support sequence representation take only eight numbers. A section of n touching phases takes $n+1$ support points in support sequence representation, so $2n+2$ units of memory. In phase

representation, it takes n phases, so $3n+1$ units of memory. For n towards infinity we then get a factor of

$$\frac{3n+1}{2n+2} \xrightarrow{n\to\infty} \frac{3}{2}$$

for memory consumption.

Since phases never touch in our input data, we save space with phase representation for our input data.

3.2. The Approximate Approach

The idea for the actual approximation originates from the link operation. When linking two functions, support points can cover each other, meaning that only one of them needs to be stored for the linked function. But we can go even further. In this subsection we will introduce a different interpretation of phase functions that allow them to cover whole intervals of input or output space. For that, in Section 3.2.1, we first introduce the term *cover potential* for piecewise linear arrival time functions that are represented as support sequences. It is a subset of the input or output space of a function such that breakpoints that are mapped into these subsets on composition become redundant in the link support sequence of the composed function. In Section 3.2.2, we use the insights from cover potentials to devise an approximation for phase functions that maximize their cover potentials. With that, we prove a theorem about the size of a link of squashed phase functions.

3.2.1. Cover Potentials

Cover potentials exist for both travel time and arrival time functions, but they are easier to describe with arrival time functions. Since the translation from travel time to arrival time functions and back is simple, it is enough to work with arrival time functions in this section.

We define cover potentials together with cover potential conditions that help us define and argue about cover potentials.

Definition 3.7 (Weak/Strong Input/Output Cover Potential (Condition) of Weakly Increasing Support Sequences)**.** *Let S be a weakly increasing support sequence. Let $f_S : D \to R$ be its derived piecewise linear function with domain D and range R and implicit slopes of* 1*. Let $\mathcal{T}$ be the set of all weakly increasing minimal support sequences.*

We define D' as the weak input cover potential *of f_S, and R' as the* weak output cover potential *of f_S. And we define D'' as the* strong input cover potential *of f_S, and R'' as the* strong output cover potential *of f_S.*

- *Let $x_0 \in D$. Let $T \in \mathcal{T}$. If the link support sequence* LINK(T, S) *contains exactly one redundant support point that originates from a non-redundant support point $p \in T_{y=x_0}$, or all $p \in T_{y=x_0}$ are redundant, then T fulfills the* weak input cover potential condition *for S and x_0. If all $T \in \mathcal{T}$ fulfill the weak input cover potential condition for f_S and x_0, then $x_0 \in D'$.*

  If each $p \in T_{y=x_0}$ produces a redundant support point in the link support sequence LINK(T, S)*, then T fulfills the* strong input cover potential condition *for S and x_0. If all $T \in \mathcal{T}$ fulfill the strong input cover potential condition for f_S and x_0, then $x_0 \in D''$.*

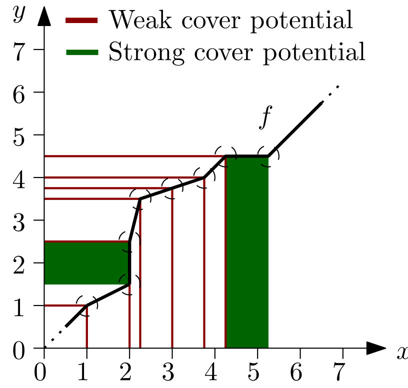


Figure 3.2.: The cover potentials of a support sequence. Cover potentials drawn towards the x-axis are input cover potentials, and cover potentials drawn towards the y-axis are output cover potentials. The support points are marked by dashed circles.

- *Let $y_0 \in R$. Let $T \in \mathcal{T}$. If the link support sequence* LINK(S,T) *contains exactly one redundant support point that originates from a non-redundant support point $p \in T_{x=y_0}$, or all $p \in T_{x=y_0}$ are redundant, then T fulfills the* weak output cover potential condition *for S and y_0. If all $T \in \mathcal{T}$ fulfill the weak output cover potential condition for f_S and y_0, then $y_0 \in R'$.*

  If each $p \in T_{x=y_0}$ produces a redundant support point in the link support sequence LINK(S,T)*, then T fulfills the* strong output cover potential condition *for S and y_0. If all $T \in \mathcal{T}$ fulfill the strong output cover potential condition for f_S and y_0, then $y_0 \in R''$.*

The following lemmas describe the structure of the cover potentials of weakly increasing support sequence in detail. The cover potentials of an exemplary support sequence are displayed in Figure 3.2. We first define specific function behaviors that we need in the lemmas below.

Definition 3.8 (Function is Constant on Interval)**.** *Let $f : D \subseteq \mathbb{R} \to R \subseteq \mathbb{R}$ be a function. Then f is* constant on *an interval $I \subseteq D$, if $\exists y \in R : \forall x \in I : f(x) = y$.*

Definition 3.9 (Function Jumps over Interval)**.** *Let $f : D \subseteq \mathbb{R} \to R \subseteq \mathbb{R}$ be a function. Then f* jumps over *an interval $I \subseteq R$, if $\exists x \in D : \forall y \in I : f(x) = y$.*

Definition 3.10 (Function is Constant around Ordinal)**.** *Let $f : D \subseteq \mathbb{R} \to R \subseteq \mathbb{R}$ be a function. Then f is* constant around *an ordinal $x_0 \in D$, if $\exists \epsilon > 0 : \exists y \in R : \forall x \in [x_0 - \epsilon, x_0 + \epsilon] : f(x) = y$.*

Definition 3.11 (Function Jumps over Ordinal)**.** *Let $f : D \subseteq \mathbb{R} \to R \subseteq \mathbb{R}$ be a function. Then f* jumps over *an ordinal $y_0 \in R$, if $\exists \epsilon > 0 : \exists x \in D : \forall y \in [y_0 - \epsilon, y_0 + \epsilon] : f(x) = y$.*

Lemma 3.12 (Support Points are Weak Cover Potential)**.** *Let S be a weakly increasing support sequence. Let f_S be its derived piecewise linear function and D' its weak input cover potential and R' its weak output cover potential. Then for all support points $(x_0, y_0) \in S$:*

(a) f_S is not constant on $[x_0 - \epsilon_S, x_0]$ for any $\epsilon_S > 0 \Rightarrow x_0 \in D'$,

(b) f_S does not jump over $[y_0, y_0 + \epsilon_S]$ for any $\epsilon_S > 0 \Rightarrow y_0 \in R'$.

Proof. Let $(x_0, y_0) \in S$.

(a) Let f_S not be constant on $[x_0 - \epsilon_S, x_0]$ for any $\epsilon_S > 0$. Let $\mathcal{T}$ be the set of all weakly increasing minimal support sequences. Let $T \in \mathcal{T}$. Let $L := \text{LINK}(T, S)$.

- Assume, $|T_{y=x_0}| = 0$. Then T fulfills the weak input cover potential condition for f_S and x_0.
- Assume, $|T_{y=x_0}| = 1$. Let $(p_1 := (a_1, x_0)) := T_{y=x_0}$. Since T is minimal, p_1 is not redundant.

  Then, p_1 produces $(a_1, f_S(x_0)) \in L$.

  Since $(x_0, y_0) \in S$, it follows that $(x_0, f_S(x_0)) \in S$. Then, $(x_0, f_S(x_0))$ produces $(f_T^{-1}(x_0), f_S(x_0)) = (a_1, f_S(x_0)) \in L$.

  So, p_1 produces a redundant support point in L. So, T fulfills the weak input cover potential condition for f_S and x_0.
- Assume, $|T_{y=x_0}| = 2$. Let $(p_1 := (a_1, x_0), p_2 := (a_2, x_0)) := T_{y=x_0}$. Since T is minimal, p_1 and p_2 are not redundant.

  Since $f_T^{-1}(x_0) = a_2$, p_2 produces a redundant support point in L by the same argument as for $|T_{y=x_0}| = 1$ above.

  We give proof for p_1 producing the non-redundant point $p'_1 := (a_1, f_S(x_0)) \in L$ by proving that p'_1 is not duplicate and a breakpoint.

  The support point p'_1 is not duplicate in L because:

  * If T contains a support point $(a_1, \check{x})$ with $\check{x} < x_0$, then, since f_S is not constant on $[x_0 - \epsilon_S, x_0]$ for any $\epsilon_S > 0$, $f_S(\check{x}) < f_S(x_0)$. So, the product of $(a_1, \check{x})$ in L is not equal to p'_1.
  * T cannot contain a support point $(a_1, \hat{x})$ with $\hat{x} > x_0$, since $p_1 \neq p_2$.
  * If T contains a support point p with $x \neq a_1$, then the product of p in L is not equal to p'_1.
  * If S contains a support point $(\check{x}, f_S(x_0))$ with $\check{x} < x_0$, then f_S would be constant on $[x_0 - \epsilon_S, x_0]$ for some $\epsilon_S > 0$, but by definition it is not.
  * If S contains a support point $(\hat{x}, f_S(x_0))$ with $\hat{x} > x_0$, then $f_T^{-1}(\hat{x}) \geq a_2 > a_1$. So, the product of $(\hat{x}, f_S(x_0))$ in L is not equal to p'_1.
  * If S contains a support point p with $y \neq f_S(y_0)$, then the product of p in L is not equal to p'_1.

  So, only p_1 can produce a point that is equal to p'_1.

  We prove that p'_1 is a breakpoint.

  Since p_1 is the first point with $y = x_0$ in T, f_T is not constant on $[a_1 - \epsilon_T, a_1]$ for any $\epsilon_T > 0$. So, $f_T(\check{a}) < x_0$ for $\check{a} < a_1$. Also, since f_S is not constant on $[x_0 - \epsilon_S, x_0]$ for any $\epsilon_S > 0$, $f_S(\check{x}) < y_0$ for $\check{x} < x_0$. So, $f_L(\check{a}) < y_0$ for $\check{a} < a_1$.

  But since f_T is constant on $(a_1, a_2]$, f_L is constant on $(a_1, a_2]$ with value $f_S(x_0)$. So, p'_1 is the starting point of this constant. So, p'_1 is a breakpoint in f_L.

  So T fulfills the weak input cover potential condition for f_S and x_0.
- Assume $|T_{y=x_0}| > 2$. This is a contradiction to the minimality of T.

So all $T \in \mathcal{T}$ fulfill the weak input cover potential condition for f_S and x_0. So $x_0 \in D'$.

(b) Let f_S not jump over $[y_0, y_0 + \epsilon_S]$ for any $\epsilon_S > 0$. Let $\mathcal{T}$ be the set of all weakly increasing support sequences that have at least one support point $x = y_0$. Let $T \in \mathcal{T}$. Let $L := \text{LINK}(S, T)$.

- Assume, $|T_{x=y_0}| = 0$. Then T fulfills the weak output cover potential condition for f_S and y_0.
- Assume, $|T_{x=y_0}| = 1$. Let $(p_1 := (y_0, b_1)) := T_{x=y_0}$. Since T is minimal, p_1 is not redundant.

  Then, p_1 produces $(f_S^{-1}(y_0), b_1) \in L$.

  Since $(x_0, y_0) \in S$, it follows that $(f_S^{-1}(y_0), y_0) \in S$. Then, $(f_S^{-1}(y_0), y_0)$ produces $(f_S^{-1}(y_0), f_T(y_0)) = (f_S^{-1}(y_0), b_1) \in L$.

  So, p_1 produces a redundant support point in L. So, T fulfills the weak output cover potential condition for f_S and y_0.
- Assume, $|T_{x=y_0}| = 2$. Let $(p_1 := (y_0, b_1), p_2 := (y_0, b_2)) := T_{x=y_0}$. Since T is minimal, p_1 and p_2 are not redundant.

  Since $f_T(y_0) = b_1$, p_1 produces a redundant support point in L by the same argument as for $|T_{x=y_0}| = 1$ above.

  We give proof for p_2 producing the non-redundant point $p_2' := (f_S^{-1}(y_0), b_2) \in L$ by proving that p_2' is not duplicate and a breakpoint.

  The support point p_2' is not duplicate in L because:

  * If T contains a support point $(\hat{y}, b_2)$ with $\hat{y} > y_0$, then, since f_S does not jump over $[y_0, y_0 + \epsilon_S]$ for any $\epsilon_S > 0$, $f_S^{-1}(\hat{y}) > f_S^{-1}(y_0)$. So, the product of $(\hat{y}, b_2)$ in L is not equal to p_2'.
  * T cannot contain a support point $(\check{y}, b_2)$ with $\check{y} < y_0$, since $p_1 \neq p_2$.
  * If T contains a support point p with $y \neq b_2$, then the product of p in L is not equal to p_2'.
  * If S contains a support point $(f_S^{-1}(y_0), \hat{y})$ with $\hat{y} > y_0$, then f_S would jump over $[y_0, y_0 + \epsilon_S]$ for some $\epsilon_S > 0$, but by definition it does not.
  * If S contains a support point $(f_S^{-1}(y_0), \check{y})$ with $\check{y} < y_0$, then $f_T(\check{y}) \leq b_1 < b_2$. So the product of $(f_S^{-1}(y_0), \check{y})$ in L is not equal to p_2'.
  * If S contains a support point p with $x \neq f_S^{-1}(y_0)$, then the product of p in L is not equal to p_2'.

  So, only p_2 can produce a point that is equal to p_2'.

  We prove that p_2' is a breakpoint.

  Since p_2 is the last point with $x = y_0$ in T, f_T does not have a jump at y_0 such that $\lim_{\epsilon \to 0} f_T(y_0 + \epsilon) > b_2$. So, $f_T^{-1}(\hat{b}) > y_0$ for $\hat{b} > b_2$. Also, since f_S does not jump over $[y_0, y_0 + \epsilon_S]$ for any $\epsilon_S > 0$, $f_S^{-1}(\hat{y}) > x_0$ for $\hat{y} > y_0$. So, $f_L^{-1}(\hat{b}) > x_0$ for $\hat{b} > b_2$.

  But since f_T^{-1} is constant on $[b_1, b_2)$, f_L^{-1} is constant on $[b_1, b_2)$ with value $f_S^{-1}(y_0)$. So, p_2' is the ending point of this constant. So, p_2' is a breakpoint in f_L.

  So, T fulfills the weak output cover potential condition for f_S and y_0.
- Assume $|T_{x=y_0}| > 2$. This is a contradiction to the minimality of T.

So all $T \in \mathcal{T}$ fulfill the weak output cover potential condition for f_S and y_0. So $y_0 \in R'$.

□

Lemma 3.13 (Constants and Jumps are Strong Cover Potential). *Let S be a weakly increasing support sequence. Let f_S be its derived piecewise linear function and D'' its strong input cover potential and R'' its strong output cover potential. Then*

(a) $\forall x_0 : f_S$ *is constant on* $[x_0 - \epsilon_S, x_0]$ *for some* $\epsilon_S > 0 \Rightarrow x_0 \in D''$ *and*

(b) $\forall y_0 : f_S$ *jumps over* $[y_0, y_0 + \epsilon_S]$ *for some* $\epsilon_S > 0 \Rightarrow y_0 \in R''$*.*

Proof. Let $\mathcal{T}$ be the set of all weakly increasing minimal support sequences.

(a) Let $x_0 \in \mathbb{R}$ such that f_S is constant on $[x_0 - \epsilon_S, x_0]$ for some $\epsilon_S > 0$. Let $T \in \mathcal{T}$ with support point (a, x_0) for some a. Let $L := \text{LINK}(T, S)$.

Since f_S is constant on $[x_0 - \epsilon_S, x_0]$ for some $\epsilon_S > 0$, $(\check{x}, f_S(x_0)), (\hat{x}, f_S(x_0)) \in S$ for $\check{x} < x_0$ and $\hat{x} \geq x_0$.

So, $(a, f_S(x_0)), (f_T^{-1}(\check{x}), f_S(x_0)), (f_T^{-1}(\hat{x}), f_S(x_0)) \in L$. It holds that $f_T^{-1}(\check{x}) \leq a$ and $f_T^{-1}(\hat{x}) \geq a$.

But then, either L contains duplicate points or $(a, f_S(x_0))$ is not a breakpoint in f_L, so L contains a redundant point that originates from $(a, x_0) \in T$. So each $T \in \mathcal{T}$ fulfills the strong input cover potential condition for f_S and x_0. So $x_0 \in D''$.

(b) Let $y_0 \in \mathbb{R}$ such that f_S jumps over $[y_0, y_0 + \epsilon_S]$ for some $\epsilon_S > 0$. Let T be a weakly increasing support sequence with support point (y_0, b). Let $L := \text{LINK}(S, T)$.

Since f_S jumps over $[y_0, y_0 + \epsilon_S]$ for some $\epsilon_S > 0$, $(f_S^{-1}(y_0), \check{y}), (f_S^{-1}(y_0), \hat{y}) \in S$ for $\check{y} \leq y_0$ and $\hat{y} > y_0$.

So, $(f_S^{-1}(y_0), b), (f_S^{-1}(y_0), f_T(\check{y})), (f_S^{-1}(y_0), f_T(\hat{y})) \in L$. It holds that $f_T(\check{y}) \leq b$ and $f_T(\hat{y}) \geq b$.

But then, either L contains duplicate points or $(f_S^{-1}(y_0), b)$ is not a breakpoint in f_L, so L contains a redundant point that originates from $(y_0, b) \in T$. So each $T \in \mathcal{T}$ fulfills the strong output cover potential condition for f_S and y_0. So $y_0 \in R''$.

□

Lemma 3.14 (Everything Else Never Covers). *Let S be a weakly increasing support sequence. Let f_S be its derived piecewise linear function, D' and D'' its weak and strong input cover potential and R' and R'' its weak and strong output cover potential.*

(a) Let $x_0 \in \mathbb{R}$ such that f_S not constant around x_0 and $\forall (a, b) \in S : x_0 \neq a$. Then $x_0 \notin D' \cup D''$. Moreover, for all weakly increasing support sequences T with unique breakpoint $p := (c, x_0)$ for some c, the product of p in $\text{LINK}(T, S)$ *is not redundant.*

(b) Let $y_0 \in \mathbb{R}$ such that f_S does not jump over y_0 and $\forall (a, b) \in S : y_0 \neq b$. Then $y_0 \notin R' \cup R''$. Moreover, for all weakly increasing support sequences T with unique breakpoint $p := (y_0, d)$ for some d, the product of p in $\text{LINK}(S, T)$ *is not redundant.*

Proof.

(a) Let $x_0 \in \mathbb{R}$ such that f_S not constant around x_0 and $\forall (a,b) \in S : x_0 \neq a$. Then f_S has an ascend in $(0, \infty)$ around x_0, so it is bijective around x_0.

Let T be a weakly increasing support sequence with unique breakpoint $p := (c, x_0)$. Then $L := \text{LINK}(T, S)$ has a support point $p' := (c, f_S(x_0))$ which is the product of p in L.

Since S is bijective around x_0 and S contains no support point at $x = x_0$, S contains no support point with $y = f_S(x_0)$. Since p is a unique breakpoint in T and S is bijective around x_0, T does not contain a support point (c, x_0') with $f_S(x_0) = f_S(x_0')$. So p' is a unique support point in L.

Also, p' is a breakpoint, since p is a breakpoint in T and the ascend before and after p is changed by the same linear factor when linking. So p' is not redundant in L.

(b) Let $y_0 \in \mathbb{R}$ such that f_S does not jump over y_0 and $\forall (a,b) \in S : y_0 \neq b$. Then f_S has an ascend in $(0, \infty)$ around $x_0 := f_S^{-1}(y_0)$, so it is bijective around x_0.

Let T be a weakly increasing support sequence with unique breakpoint $p := (y_0, d)$. Then $L := \text{LINK}(S, T)$ has a support point $p' := (f_S^{-1}(y_0), d)$ which is the product of p in L.

Since S is bijective around x_0, and S contains no support point at $y = y_0$, S contains no support point with $x = x_0$. Since p is a unique breakpoint in T and S is bijective around x_0, T does not contain a support point (y_0', d) with $f_S^{-1}(y_0) = f_S^{-1}(y_0')$. So p' is a unique support point in L.

Also, p' is a breakpoint, since p is a breakpoint in T and the ascend before and after p is changed by the same linear factor when linking. So p' is not redundant in L.

□

The cover potentials describe where points are saved on link. Note that redundant points never produce non-redundant points on link. So with cover potentials, an upper bound for the size of the link can be given. But we use cover potentials with phases, because there we can give a more accurate bound, and we use the phase model for our optimization scheme.

3.2.2. Maximizing the Cover Potentials by Approximation

We have seen that for creating strong cover potentials, an arrival time function is required to have constants and jumps. But our input functions do not have these features, their traffic changes are not as steep. We approximate the input functions by defining phases to not have a uniform slope, but to contain a jump or constant somewhere within the phase.

Then, our weight functions have slopes restricted to 0, 1, and ∞. As discussed in [FHS11], this reduces the worst case complexity of our shortcut weights from $K \cdot n^{\mathcal{O}(\log n)}$ to $\mathcal{O}(K)$, where n is the amount of nodes in the graph and K the total amount of breakpoints in all weight functions. While this is just a worst case bound, it is a hint that we reduce the size of the TDCCH with restricted slopes compared to unrestricted slopes.

Since we try to reduce memory consumption, we restrict each phase to have exactly one jump or constant. Otherwise, we would need more space to store the lengths and the locations of multiple jumps or constants. A phase $p := ([a, b], h)$ gets expanded to $([a, b], h, c)$, where c describes the position of the jump or constant. We do not store c explicitly, as we only need it temporarily in our optimization algorithm for the phase function link. The condition for maximum absolute slopes means, that phases with positive height are approximated with jumps, and phases with negative height are approximated with constants. If a phase

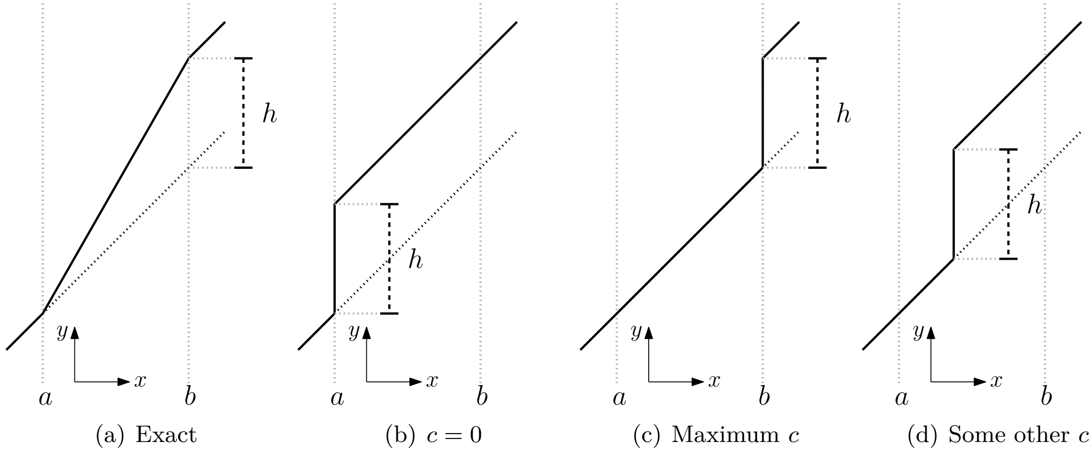


Figure 3.3.: From exact phases to approximate phases. In Subfigure (a), the original increasing phase is displayed. In Subfigure (b), (c) and (d), possible approximations of the same phase are displayed.

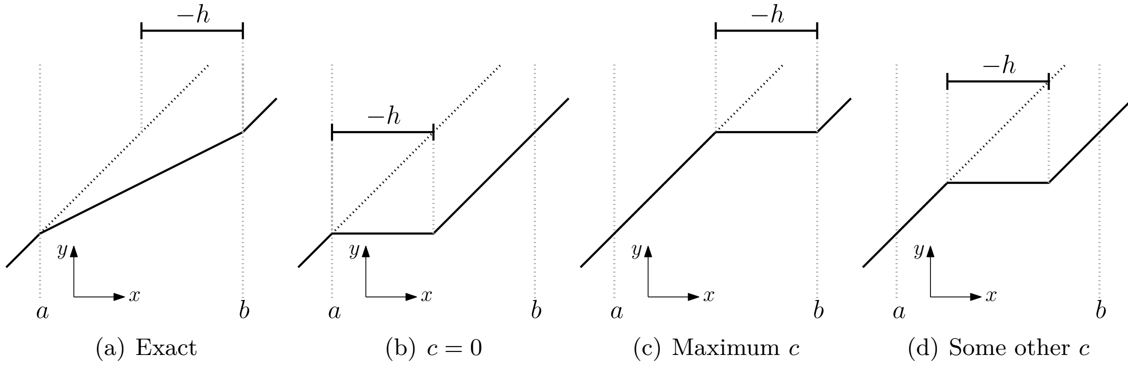


Figure 3.4.: From exact phases to approximate phases. In Subfigure (a), the original increasing phase is displayed. In Subfigure (b), (c) and (d), possible approximations of the same phase are displayed.

with negative height would be approximated by anything with a negative slope, then the function would not be a FIFO arrival time function anymore. In Figures 3.3 and 3.4, we give an example for the approximation. Note that all approximations displayed are valid for the displayed exact phase.

With this approximation, we gain degrees of freedom for optimizing the result size of a link. In the remainder of this subsection, we examine how phases behave on link, and we conclude the section by giving a formula for the size of the link depending on its input functions.

We define the terms for the approximation we describe above.

Definition 3.15 (Approximated Phase)**.** *Let* $p := ([a, b], h)$ *be a phase. Let* $c \in [0, b - \max\{0, -h\}]$*. We define* $\bar{p} := ([a, b], h, c)$ *as its* approximated phase *with* offset *c.*

Using approximated phases, we define approximated phase functions.

Definition 3.16 (Approximated Phase Function)**.** *Let* $f := (P := (p_1, \ldots, p_n), z)$ *be a phase function. Let* $c := (c_1, \ldots, c_n)$ *be a real sequence. We define* $\bar{f} := ((\bar{p}_i \text{ with offset } c \mid p_i \in P), z)$*.*

The offset determines the positioning of the jump or the constant that approximates the phase. We call this jump or constant a *phase core*.

Definition 3.17 (Input Phase Core, Output Phase Core)**.** *Let* $f = (P, z)$ *be a phase function,* $p := ([a, b], h) \in P$ *be a phase and* $[a', b'] := f_{out}(p)$ *its phase output interval. Let* $\bar{p} := ([a, b], h, c)$ *be an approximation of* p*. We define* $\bar{p}_{in} := [a + c, a + \max\{0, -h\} + c]$ *as its* input phase core *and* $\bar{p}_{out} = [a' + c, a' + \max\{0, h\} + c]$ *as its* output phase core.

Since approximated phases are either constants or jumps, either their input phase core or their output phase core is an interval with one element.

We can restrict a phase to its core, such that it is equal to its approximation.

Definition 3.18 (Squashed Phase (Function))**.** *Let* $f = (P, z)$ *be a phase function and* $\bar{f}$ *its approximated phase function. For a phase* $p := ([a, b], h)$ *we define* $\rangle p \langle := (\bar{p}_{in}, h)$ *as its* squashed phase*, where* $\bar{p}_{in}$ *is the input phase core of* p*. We define the* squashed phase function *of* f *as* $\rangle f \langle := ((\rangle p \langle \mid p \in P), z)$*.*

The squashed phase function $\rangle f \langle$ is a phase function. It can be translated back into a support sequence S such that its derived piecewise linear function f_S contains only ascends of zero, one and infinity. We describe the translation in Algorithm 3.3 further above.

A squashed phase function $\rangle f \langle := (P, z)$ has a unique approximated phase function, such that for each $p := ([a, b], h) \in P$, it holds that $\bar{p} = ([a, b], h, 0)$. So for squashed phase functions we can use the terms *input phase core* and *output phase core* without giving an approximation as context.

With squashed phases, we can define the product of a phase under the link operation.

Definition 3.19 (Phase Product)**.** *Let* $f := (P_f, z_f)$ *and* $g := (P_g, z_g)$ *be squashed phase functions. Let* $S := \textsc{supportSequence}(f)$ *and* $T := \textsc{supportSequence}(g)$*. Let* $p \in P_f$*. Let* p_1 *and* p_2 *be its corresponding support points in* S*. Let* L *be the link support sequence of linking* S *and* T *in any order.*

Then, the phase p' *corresponding to the products of* p_1 *and* p_2 *in* L *is called the* product *of* p*.*

The phase product is only well-defined, if the products of p_1 and p_2 describe a phase. We prove this in the following lemma.

Lemma 3.20 (Phase Product is sound)**.** *Let* $f := (P_f, z_f)$ *and* $g := (P_g, z_g)$ *be squashed phase functions. Let* $S := \textsc{supportSequence}(f)$ *and* $T := \textsc{supportSequence}(g)$*. Let* $p \in P_f$*. Let* $p_1 := (x_1, y_1)$ *and* $p_2 := (x_2, y_2)$ *be its corresponding points in* S*. Let* L *be the link support sequence of linking* S *and* T *in any order. Let* $p'_1 := (x'_1, y'_1)$ *and* $p'_2 := (x'_2, y'_2)$ *be the products of* p_1 *and* p_2 *in* L*.*

Then, p'_1 *and* p'_2 *describe a jump or a constant.*

Proof.

- Assume $x_1 = x_2$.
  - If $L = \textsc{link}(S, T)$, then $x'_1 = x_1 = x_2 = x'_2$. So, p'_1 and p'_2 describe a jump phase.
  - If $L = \textsc{link}(T, S)$, then $x'_1 = g^{-1}(x_1) = g^{-1}(x_2) = x'_2$. So, p'_1 and p'_2 describe a constant phase.

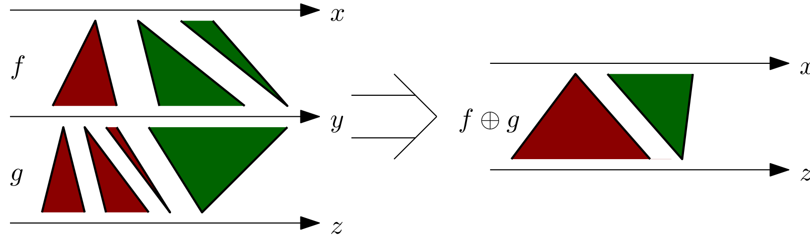


Figure 3.5.: Cover groups for exemplary squashed phase functions and their products in the link. Phases of the same cover group are highlighted by the same color.

- Assume $y_1 = y_2$.
  - If $L = \text{LINK}(S, T)$, then $y'_1 = g(y_1) = g(y_2) = y'_2$. So, p'_1 and p'_2 describe a jump phase.
  - If $L = \text{LINK}(T, S)$, then $y'_1 = y_1 = y_2 = y'_2$. So, p'_1 and p'_2 describe a constant phase.
- If neither $x_1 = x_2$ nor $y_1 = y_2$, then p would not be a squashed phase.

So, in all cases, p'_1 and p'_2 describe a jump phase or a constant phase. □

From a specific approximation $\bar{f}$ for f, we also get specific unique cover potentials for the derived piecewise linear function $\rangle f \langle_S$ of its squashed counterpart. Moreover, the set of input and output cores of $\rangle f \langle$ corresponds to the strong input and output cover potential of f_S. So, we can investigate how phases that fall into the cover potential of another phase in a link behave. We introduce terms for this kind of phase group.

Definition 3.21 (Covering, Coverer, Coveree, Input/Output Cover Group, Cover Group Size)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions. Let $p \in P_f$. Let $p_{in} := [x_1, x_2]$ be its input phase core and $p_{out} := [y_1, y_2]$ be its output phase core.*

(a) If $x_1 \neq x_2$, let $C_{in}(P_g, p) := \{q \in P_g \mid q_{out} \subseteq p_{in}\}$. Then we say that p covers *each $q \in C_{in}(P_g, p)$ when linking $g \oplus f$. We call p the* coverer *and each $q \in C_{in}(P_g, p)$ a* coveree*. We call $(C_{in}(P_g, p), p)$ the* input cover group *of p. We call $|(C_{in}(P_g, p), p)| := |C_{in}(P_g, p)|$ its* size*.*

(b) If $y_1 \neq y_2$, let $C_{out}(p, P_g) := \{q \in P_g \mid q_{in} \subseteq p_{out}\}$. Then we say that p covers *each $q \in C_{out}(p, P_g)$ when linking $f \oplus g$. We call p the* coverer *and each $q \in C_{in}(p, P_g)$ a* coveree*. We call $(p, C_{out}(p, P_g))$ the* output cover group *of p. We call $|(p, C_{out}(p, P_g))| := |C_{out}(p, P_g)|$ its* size*.*

In Figure 3.5, an exemplary pair of squashed phase functions and their cover groups are displayed.

Definition 3.22 (Cover Group Set)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions. We define $\text{COVERGROUPS}(f, g) := \{(p, C_{out}(p, P_g)) \mid p \in P_f \wedge p \text{ is jump} \wedge C_{out}(p, P_g) \neq \emptyset\} \cup \{(C_{in}(P_f, q), q) \mid q \in P_g \wedge q \text{ is constant} \wedge C_{in}(P_f, q) \neq \emptyset\}$ as the* cover group set *of f and g.*

An intuition of covering is that the covering phase swallows all coverees in the link, and thus each covered phase leads to one less phase in the link.

Note that this intuition of covering only works, if the phases of each function are pairwise disjoint within the function. By Lemma 3.13, the startpoint of a constant is not part of the

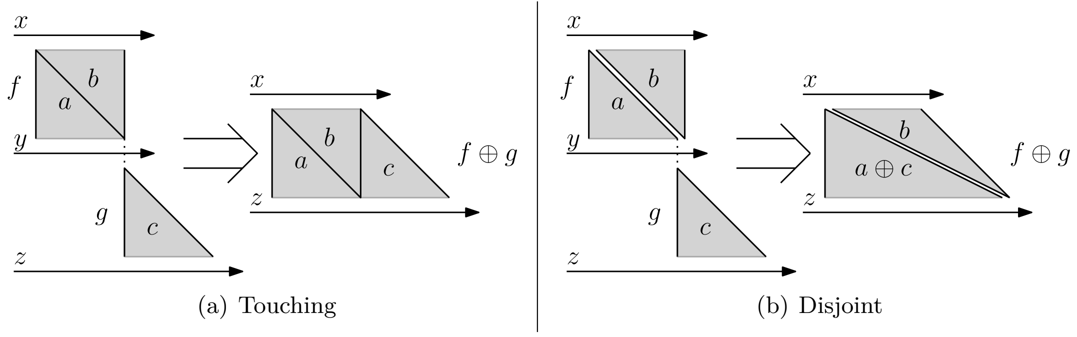


Figure 3.6.: Example for touching phases prohibiting a covering between unrelated phases. The functions are displayed with parallel axes, where f maps from x to y and g from y to z. The phases are displayed as triangles, where a triangle with its tip at the top is a jump, and a triangle with its tip at the bottom is a constant. In Subfigure (a), even though the output phase core of a intersects the input phase core of c, they produce separate phases in the link. In Subfigure (b), the cores intersect and thus a and c form a cover relationship and produce one common phase.

strong input cover potential and the endpoint of a jump is not part of the strong output cover potential. In effect, if a constant is preceded by a jump, then the constant does not cover constants that fall into its startpoint from input side. And if a jump is followed by a constant, then the jump does not cover jumps that fall into its endpoint from output side. But if jumps and constants are isolated, then their end- or startpoints only being weak cover potential does not prevent them from covering other phases in that point. The case of a jump following a constant is displayed in Figure 3.6.

We make the intuition of covering more concrete in the following lemma.

Lemma 3.23 (Phase Cover Group)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions such that the phases in P_f are pairwise disjoint and the phases in P_g are pairwise disjoint. Let $S :=$ SUPPORTSEQUENCE(f) and $T :=$ SUPPORTSEQUENCE(g). Let $p \in P_f$.*

(a) If p is a constant with value y_1, let $(C_{in}(P_g, p), p)$ be the input cover group of p. Let $L :=$ LINK(T, S). Then the products p'_1 and p'_2 of the support points $p_1, p_2 \in S$, $p_1 < p_2$, corresponding to p and the products q'_1 and q'_2 of the support points in $q_1, q_2 \in T$, $q_1 < q_2$, corresponding to each $q \in Q$ are collinear in L with $y = y_1$.

(b) If p is a jump at x_1, let $(p, C_{out}(p, P_g))$ be the output cover group of p. Let $L :=$ LINK(S, T). Then the products p'_1 and p'_2 of the support points $p_1, p_2 \in S$, $p_1 < p_2$, corresponding to p and the products q'_1 and q'_2 of the support points in $q_1, q_2 \in T$, $q_1 < q_2$, corresponding to each $q \in Q$ are collinear in L with $x = x_1$.

Proof.

(a) Let p be a constant with value y_1. Let $(C_{in}(P_g, p), p)$ be the input cover group of p. Let $L :=$ LINK(T, S).

Then it holds that $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ with $x_1 < x_2$ and $y_1 = y_2$. And since the phases in P_f are pairwise disjoint, f_S does not have a jump at x_1. So f_S is constant on p_{in} with value y_1.

For $q \in C_{in}(P_g, p)$, let $(a_1, b_1) := q_1$ and $(a_2, b_2) := q_2$. Then it holds that $b_1, b_2 \in p_{\text{in}}$, since $q_{\text{out}} \subseteq p_{\text{in}}$. So, q'_1 and q'_2 are at $y = y_1$. Also, p'_1 and p'_2 are at $y = y_1$. So, q'_1, q'_2, p'_1 and p'_2 are collinear.

(b) Let p be a jump at x_1. Let $(p, C_{\text{out}}(p, P_g))$ be the output cover group of p. Let $L := \text{LINK}(S, T)$.

Then it holds that $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ with $y_1 < y_2$ and $x_1 = x_2$. And since the phases in P_f are pairwise disjoint, f_S does not have a constant with value y_2. So f_S^{-1} is constant on p_{out} with value x_1.

For $q \in C_{\text{out}}(p, P_g)$, let $(a_1, b_2) := q'_1$ and $(a_2, b_2) := q'_2$. Then it holds that $a_1, a_2 \in p_{\text{out}}$, since $q_{\text{in}} \subseteq p_{\text{out}}$. So, q'_1 and q'_2 are at $x = x_1$. Also, p'_1 and p'_2 are at $x = x_1$. So, q'_1, q'_2, p'_1 and p'_2 are collinear.

□

So, the products of the support points of all phases in a cover group are collinear. The resulting phase of an input cover group is almost always a constant phase, and the resulting phase of an output cover group is almost always a jump phase in the link. The exception to that are exact cover groups. Their product has height zero.

Lemma 3.24 (Exact Cover Group)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions such that the phases in P_f are pairwise disjoint and the phases in P_g are pairwise disjoint. Let $p \in P_f$. Let $(C_{in}(P_g, p), p)$ be the input cover group of p, if it exists, and $(p, C_{out}(p, P_g))$ be the output cover group of p, if it exists.*

(a) If $\bigcup_{q \in C_{in}(P_g,p)} q_{out} = p_{in}$, then the products of all support points corresponding to $(C_{in}(P_g, p), p)$ are equal in the link.

(b) If $\bigcup_{q \in C_{out}(p,P_g)} q_{in} = p_{out}$, then the products of all support points corresponding to $(p, C_{out}(p, P_g))$ are equal in the link.

Proof.

(a) Assume, $\bigcup_{q \in C_{\text{in}}(P_g,p)} q_{\text{out}} = p_{\text{in}}$. Then, since phases do not touch, $|C_{\text{in}}(P_g, p)| = 1$. Let $\{q\} := C_{\text{in}}(P_g, p)$.

The support points corresponding to p and q are of the form $q_1 := (x, y_1)$, $q_2 := (x, y_2)$, $p_1 := (y_1, z)$ and $p_2 := (y_2, z)$ such that $y_1 < y_2$. But then, since phases do not touch, their products are all (x, z).

(b) Assume, $\bigcup_{q \in C_{\text{out}}(p,P_g)} q_{\text{in}} = p_{\text{out}}$. Then, since phases do not touch, $|C_{\text{out}}(p, P_g)| = 1$. Let $\{q\} := C_{\text{out}}(p, P_g)$.

The support points corresponding to p and q are of the form $p_1 := (x, y_1)$, $p_2 := (x, y_2)$, $q_1 := (y_1, z)$ and $q_2 := (y_2, z)$ such that $y_1 < y_2$. But then, since phases do not touch, their products are all (x, z).

□

We look at all other possible positionings of phase cores and their behaviors on link.

Lemma 3.25 (Free Phases)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions such that the phases in P_f are pairwise disjoint and the phases in P_g are pairwise disjoint. Let $S :=$ SUPPORTSEQUENCE(f) and $T :=$ SUPPORTSEQUENCE(g). Let $p \in P_f$. Let $[x_1, x_2] := p_{in}$ and $[y_1, y_2] := p_{out}$.*

(a) If $\forall q \in P_g : q_{out} \cap p_{in} = \emptyset \vee (q_{out} = p_{in} \wedge x_1 = x_2)$, then the support points $p_1 := (x_1, y_1), p_2 := (x_2, y_2) \in S$ that correspond to p produce the consecutive breakpoints $p'_1 := (g^{-1}(x_1), y_1), p'_2 := (g^{-1}(x_2), y_2) \in L :=$ LINK(T, S).

(b) If $\forall q \in P_g : q_{in} \cap p_{out} = \emptyset \vee (q_{in} = p_{out} \wedge y_1 = y_2)$, then the support points $p_1 := (x_1, y_1), p_2 := (x_2, y_2) \in S$ that correspond to p produce the consecutive breakpoints $p_1' := (x_1, g(y_1)), p_2' := (x_2, g(y_2)) \in L := \text{LINK}(S, T)$.

Proof.

(a) Let $L := \text{LINK}(T, S)$. Let $p_1 := (x_1, y_1), p_2 := (x_2, y_2) \in S$ be the support points that correspond to p. Let $p_1' := (g^{-1}(x_1), y_1), p_2' := (g^{-1}(x_2), y_2)$ be their products in L. Let $\forall q \in P_g : q_{\text{out}} \cap p_{\text{in}} = \emptyset \vee (q_{\text{out}} = p_{\text{in}} \wedge x_1 = x_2)$. The condition $(q_{\text{out}} = p_{\text{in}} \wedge x_1 = x_2)$ holds for at most one $q \in P_g$ and contradicts $q_{\text{out}} \cap p_{\text{in}} = \emptyset$.

- If $\nexists q \in P_g : q_{\text{out}} = p_{\text{in}} \wedge x_1 = x_2$, then g^{-1} has an ascend of 1 on $[x_1, x_2]$. Then, p_1' and p_2' are consecutive breakpoints in L.
- Otherwise, g contains a constant phase q with value $x_1 = x_2$. Then, the support points corresponding to q are of the form $q_1 := (a_1, x_1)$ and $q_2 := (a_2, x_1)$ with $a_1 < a_2$. And because p is a jump at x_1, it holds that $y_1 < y_2$.

  The products of q_1 and q_2 are $q_1' := (a_1, f(x_1)), q_2' := (a_2, f(x_1)) \in L$. There are no other support points with $x = x_1$ in S and no other support points with $y = x_1$ in T. So $f(x_1) = y_1$ and $g^{-1}(x_1) = a_2$. So LINK produces the consecutive support points $q_1' = (a_1, y_1), q_2' = (a_2, y_1), p_1' = (a_2, y_1), p_2' = (a_2, y_2) \in L$.

  We prove, that they are all breakpoints in L.

  * Since the listed products are consecutive, $q_2' = p_1'$ are breakpoints in L.
  * Since phases are disjoint, $\exists \epsilon > 0 : \forall \delta \in (0, \epsilon) : f(x_1 + \delta) = f(x_1) + \delta$. And, since phases are disjoint, $\exists \epsilon > 0 : \forall \delta \in (0, \epsilon) : g(a_2 + \delta) = g(a_2) + \delta = x_2 + \delta$. So, $\exists \epsilon > 0 : \forall \delta \in (0, \epsilon) : (g \oplus f)(a_2 + \delta) = (g \oplus f)(a_2) + \delta$. So, the right-sided derivative of $(g \oplus f)$ is 1 at a_2. So, p_2' is breakpoint in L.
  * By an analogous argument, q_1' is breakpoint in L.

(b) Let $L := \text{LINK}(S, T)$. Let $p_1 := (x_1, y_1), p_2 := (x_2, y_2) \in S$ be the support points that correspond to p. Let $p_1' := (x_1, g(y_1)), p_2' := (x_2, g(y_2))$ be their products in L. Let $\forall q \in P_g : q_{\text{in}} \cap p_{\text{out}} = \emptyset \vee (q_{\text{in}} = p_{\text{out}} \wedge y_1 = y_2)$. The condition $q_{\text{in}} = p_{\text{out}} \wedge y_1 = y_2$ holds for at most one $q \in P_g$ and contradicts $q_{\text{in}} \cap p_{\text{out}} = \emptyset$.

- If $\nexists q \in P_g : q_{\text{in}} = p_{\text{out}} \wedge y_1 = y_2$, then g^{-1} has an ascend of 1 on $[y_1, y_2]$. Then, p_1' and p_2' are consecutive breakpoints in L.
- Otherwise, g contains a jump phase q at $y_1 = y_2$. Then, the support points corresponding to q are of the form $q_1 := (y_1, b_1)$ and $q_2 := (y_1, b_2)$ with $b_1 < b_2$. And because p is a constant with value y_1, it holds that $x_1 < x_2$.

  The products of q_1 and q_2 are $q_1' := (f^{-1}(y_1), b_1), q_2' := (f^{-1}(y_1), b_2) \in L$. There are no other support points with $y = y_1$ in S and no other support points with $x = y_1$ in T. So $f^{-1}(y_1) = x_2$ and $g(y_1) = b_1$. So LINK produces the consecutive support points $p_1' = (x_1, b_1), p_2' = (x_2, b_1), q_1' = (x_2, b_1), q_2' = (x_2, b_2) \in L$.

  We prove, that they are all breakpoints in L.

  * Since the listed products are consecutive, $p_2' = q_1'$ are breakpoints in L.
  * Since phases are disjoint, $\exists \epsilon > 0 : \forall \delta \in (0, \epsilon) : g(y_1 + \delta) = g(y_1) + \delta$. And, since phases are disjoint, $\exists \epsilon > 0 : \forall \delta \in (0, \epsilon) : f(x_2 + \delta) = f(x_2) + \delta = y_1 + \delta$. So, $\exists \epsilon > 0 : \forall \delta \in (0, \epsilon) : (f \oplus g)(x_2 + \delta) = (f \oplus g)(x_2) + \delta$. So, the right-sided derivative of $(f \oplus g)$ is 1 at x_2. So, q_2' is breakpoint in L.

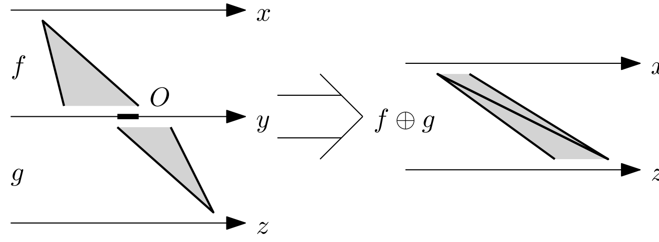


Figure 3.7.: An exemplary link of squashed phase functions with overlapping phases. The overlap O is marked by a heavy line on the y-axis.

* By an analogous argument, p'_1 is breakpoint in L.

□

By Lemma 3.25, free phases have one product in the link.

The basic situation of the lemma below is depicted in Figure 3.7.

Lemma 3.26 (Partially Overlapping Phases)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions such that the phases in P_f are pairwise disjoint and the phases in P_g are pairwise disjoint. Let $S :=$ SUPPORTSEQUENCE(f) and $T :=$ SUPPORTSEQUENCE(g). Let $p \in P_f$. Let $[x_1, x_2] := p_{in}$ and $[y_1, y_2] := p_{out}$.*

(a) For $q \in P_g$ such that $O_q := q_{out} \cap p_{in} \neq \emptyset$, $q_{out} \neq O_q$ and $p_{in} \neq O_q$, let $(x_p, y_p) \in S$ be the support point in S that corresponds to p with $x_p \in O_q$ and $(x_q, y_q) \in T$ be the support point in T that corresponds to q with $y_q \in O_q$. Then their products $l_p, l_q \in L :=$ LINK(T, S) are equal. The product of p is constant and touches the product of q that is a jump.

(b) For $q \in P_g$ such that $O_q := q_{in} \cap p_{out} \neq \emptyset$, $q_{in} \neq O_q$ and $p_{out} \neq O_q$, let $(x_p, y_p) \in S$ be the support point in S that corresponds to p with $y_p \in O_q$ and $(x_q, y_q) \in T$ be the support point in T that corresponds to q with $x_q \in O_q$. Then their products $l_p, l_q \in L :=$ LINK(S, T) are equal. The product of p is a jump and touches the product of q that is constant.

Proof.

(a) Let $q \in P_g$ such that $O_: = q_{\text{out}} \cap p_{\text{in}} \neq \emptyset \wedge q_{\text{out}} \neq O_q \wedge p_{\text{in}} \neq O_q$. Then, p is a constant and q is a jump.

Let $L :=$ LINK(T, S). Let $(x_p, y_p) \in S$ be the support point in S that corresponds to p with $x_p \in O_q$ and $(x_q, y_q) \in T$ be the support point in T that corresponds to q with $y_q \in O_q$. Then, f is constant on O_q with value y_p. And, g^{-1} is constant on O_q with value x_q.

Then, the product of $(x_p, y_p) \in S$ in L is $l_p := (g^{-1}(x_p), y_p) = (x_q, y_p)$. And, the product of $(x_q, y_q) \in T$ in L is $l_q := (x_q, f(y_q)) = (x_q, y_p)$. So, it holds that $l_p = l_q$. And, the product of p and the product of q in L touch.

Let $(x'_p, y_p) \in S$ be the support point in S that corresponds to p with $x'_p \notin O_q$ and $(x_q, y'_q) \in T$ be the support point in T that corresponds to q with $y'_q \notin O_q$. Then, the product of $(x'_p, y_p) \in S$ in L is $l'_p := (g^{-1}(x'_p), y_p)$. And, the product of $(x_q, y'_q) \in T$ in L is $l'_q := (x_q, f(y'_q))$.

Since $x'_p \notin O_q$ and phases in P_g do not touch, it holds that $g^{-1}(x'_p) \neq g^{-1}(x_p)$. So, the product of p in L is a constant. Since $y'_q \notin O_q$ and phases in P_f do not touch, it holds that $f(y'_q) \neq f(y_q)$. So, the product of q in L is a jump.

(b) Let $q \in P_g$ such that $O_q := q_{\text{in}} \cap p_{\text{out}} \neq \emptyset \wedge q_{\text{in}} \neq O_q \wedge p_{\text{out}} \neq O_q$. Then, p is a jump and q is a constant.

Let $L := \textsc{link}(S, T)$. Let $(x_p, y_p) \in S$ be the support point in S that corresponds to p with $y_p \in O_q$ and $(x_q, y_q) \in T$ be the support point in T that corresponds to q with $x_q \in O_q$. Then, f^{-1} is constant on O_q with value x_p. And, g is constant on O_q with value y_q.

Then, the product of $(x_p, y_p) \in S$ in L is $l_p := (x_p, g(y_p)) = (x_p, y_q)$. And, the product of $(x_q, y_q) \in T$ in L is $l_q := (f^{-1}(x_q), y_q) = (x_p, y_q)$. So, it holds that $l_p = l_q$. And, the product of p and the product of q in L touch.

Let $(x_p, y'_p) \in S$ be the support point in S that corresponds to p with $y'_p \notin O_q$ and $(x'_q, y_q) \in T$ be the support point in T that corresponds to q with $x'_q \notin O_q$. Then, the product of $(x_p, y'_p) \in S$ in L is $l'_p := (x_p, g(y'_p))$. And, the product of $(x'_q, y_q) \in T$ in L is $l'_q := (f^{-1}(x'_q), y_q)$.

Since $y'_p \notin O_q$ and phases in P_g do not touch, it holds that $(y'_p) \neq g(y_p)$. So, the product of p in L is a jump. Since $x'_q \notin O_q$ and phases in P_f do not touch, it holds that $f^{-1}(x'_q) \neq f^{-1}(x_q)$. So, the product of q in L is a constant.

□

By Lemma 3.26, a pair of partially overlapping phases produces two touching phases in the link.

With these results about the behavior of squashed phases under the link operation, we can give a rule to calculate the amount of phases in a link, depending on the phases in the input functions.

Theorem 3.27 (Size of a Phase Link)**.** *Let $f := (P_f, z_f)$ and $g := (P_g, z_g)$ be squashed phase functions such that the phases in P_f are pairwise disjoint and the phases in P_g are pairwise disjoint. Then $|f \oplus g| = |f| + |g| - \sum_{c \in \textsc{coverGroups}(f,g)} |c| - |\textsc{exactCoverGroups}(f, g)|$.*

Proof. We prove the equality by proving that phases that are coverees or that are exact coverers do not appear in the link, but all others do.

Let $c \in \textsc{coverGroups}(f, g)$. Then, by Lemma 3.23, the products of all points corresponding to G are collinear in the link. So they can be described by one phase. So each cover group c saves $|c|$ phases. If c is exact, then, by Lemma 3.24, its product is a phase of height 0. Then, c saves $|c| + 1$ phases.

If a phase is free in the link, then, by Lemma 3.25, it has a product in the link. If a phase is overlapping another phase in the link, then, by Lemma 3.26, both of them have a product in the link. □

By Theorem 3.27, given two approximate phase functions, we can minimize the size of their link by moving their cores into positions that maximize the sum of the sizes of all cover groups and the amount of exact cover groups. Exact cover groups are rare, and they complicate the problem of minimization, since they introduce priorities between possible coverings. So, we give an optimal solution to the more uniform problem of maximizing the amount of coverees.

3.3. Linking of Phase Functions with Optimal Result Size

We want to link two phase functions f and g, such that the resulting phase function has minimal size, without removing phases with height zero. Phases with height zero are phases that are products of exact cover groups, or phases that had height zero in one of the inputs of the link. For this section, we interpret phase functions as arrival time functions. By Theorem 3.27, minimizing the size of the result $f \oplus g$ means maximizing the sum of the sizes of the cover groups $\textsc{coverGroups}(f, g)$, if height zero phases are kept. Since restricting phase functions to not contain touching phases gives us a simple notion of covering, this restriction applies for this section.

3.3.1. Phase Function Views

For the algorithm we introduce in this section, we introduce phase function views that strip all information from phase functions that is unnecessary for this section, and add information that makes our formulations shorter.

We note that covering has a symmetric nature. When linking phase functions $f \oplus g$, then phases from f can cover phases from g and the other way around. For phases from phase function f, only the output core is relevant for forming cover groups, and for phases from phase function g, only the input core is relevant for forming cover groups.

We define phase views by constructing them from phases.

Definition 3.28 (Phase Input/Output View, Phase View Interval, Phase View Length, Phase View Core, core(·), Phase View Core Length)**.** *Let $f = (P, z)$ be a phase function. Let $p := ([a, b], h) \in P$. Let $[a_{out}, b_{out}] := f_{out}(p)$ be its phase output interval.*

We define the phase input view of p as $([a, b], \max\{0, -h\})$, and the phase output view as $([a_{out}, b_{out}], \max\{0, h\})$.

Let $p' := ([a', b'], c)$ a phase input or output view. Then $[a', b']$ is its (closed) phase view interval denoted by $\text{interval}(p')$*, and c is the* length *of the core of the phase view. The length of p' is defined as* $\text{len}(p') := b' - a'$.

The core $\text{core}(p', o)$ *of p' is a closed interval $[a' + o, a' + c + o] \subseteq [a', b']$ for $o \in [0, b' - c]$.*

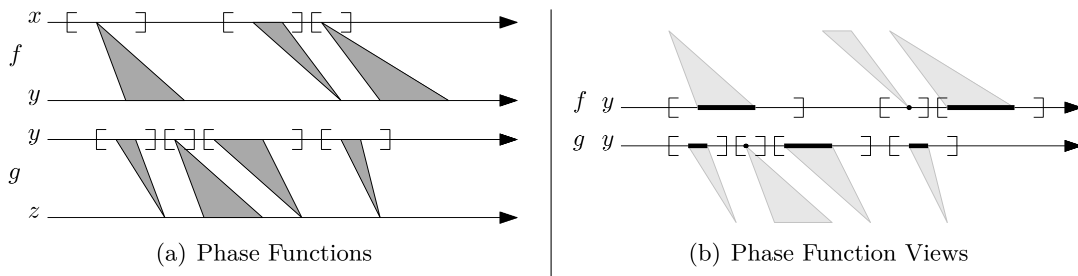
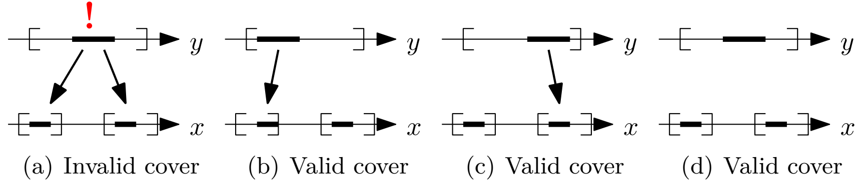


Figure 3.8.: Converting phase functions to phase function views. The brackets represent the phase intervals.

Then we can define phase function views by constructing them from phase functions. An exemplary pair of phase functions and their phase function views is displayed in Figure 3.8.

Definition 3.29 (Input/Output Phase Function View, fn(·))**.** *Let $f = (P, z)$ be a phase function.*

The input phase function view of f is defined as $f' := (p' \mid p'$ is phase input view of $p \in P)$. The output phase function view of f is defined as $f' := (p' \mid p'$ is phase output view of $p \in P)$. For a given phase view p', we denote by $\text{fn}(p')$ *the phase function view that contains p'.*



(a) Invalid cover (b) Valid cover (c) Valid cover (d) Valid cover

Figure 3.9.: Valid and invalid covers for a set of phase function views. In Figure 3.9(a), the phase view cores cannot be placed such that both cover relationships are fulfilled, so the cover is invalid. In Figures 3.9(b) to 3.9(d), the phase view cores can be placed such that both cover relationships are fulfilled, so the covers are valid.

With these views, and knowing that covering has a symmetric nature, for finding a maximum set of cover relationships, as defined in Subsection 3.3.2, the order of the phase function views is irrelevant.

3.3.2. Covering

We study covering in Subsection 3.2.2. For the algorithm we introduce in this section, we need more detailed terms for covering. The algorithm uses the geometric properties of phase views only when processing single phases. On larger scale, it constructs a set of cover relationships that is of maximum size for the given phase function views.

Definition 3.30 (Cover Relationship, Coverer, Coveree, Possible/Valid Cover Relationship, $\mathrm{rel}(\cdot)$)**.** *Let f' and g' be phase function views. Let $p' \in f'$ and $q' \in g'$. We call $p'q'$ a* cover relationship*, where p' is the* coverer*, and q' the* coveree*. For a phase r', we can write $r' \in p'q'$, if $r' = p'$ or $r' = q'$.*

Cover relationship $p'q'$ is possible*, if $\exists o_q, o_p : \mathrm{core}(q', o_q) \subseteq \mathrm{core}(p', o_p)$.*

The cover relationships that are possible in a set of phase views P from at most two phase function views can be written as $\mathrm{rel}(P)$*.*

Because of the subset relationship in cover relationships, a phase view can cover an arbitrary amount of other phase views, while it can only be covered by one other phase view. For a phase view p', that covers multiple phase views q'_i, it is not enough that each single cover relationship is possible on its own. The cover relationships introduce restrictions to the positionings of the phase cores, that might contradict for multiple cover relationships.

We define a set of cover relationships and when it is valid. An example of valid an invalid covers is given in Figure 3.9.

Definition 3.31 (Cover, Valid Cover, $\mathrm{phaseViews}(\cdot)$)**.** *Let f' and g' be phase function views. We define a* cover *$\mathcal{C}$ as a set of cover relationships $p'q'$ between f' and g', where the coverer can be in f' or in g'.*

A cover $\mathcal{C}$ is called valid*, if there exists an o_p for each phase $p \in f' \cup g'$, such that each cover relationship in $\mathcal{C}$ is valid.*

The set of phase views in the relationships in a cover $\mathcal{C}$ can be written as $\mathrm{phaseViews}(\mathcal{C}) := \{p' \mid \exists r \in \mathcal{C} : p' \in r\}$*.*

Our algorithm works on cover relationships, and searches for a leftmost maximum cover between two phase function views, as defined below. Since a cover is a set of cover relationships, a maximum cover can naturally be defined as a valid set of cover relationships of maximum cardinality.

Definition 3.32 (Maximum Cover, $\text{maxCover}(\cdot)$)**.** *Let $\mathcal{C}$ be a valid cover for two phase function views f' and g', such that there is no other valid cover $\mathcal{C}'$ with $|\mathcal{C}'| > |\mathcal{C}|$ where $|\cdot|$ is the set cardinality.*

We call such a $\mathcal{C}$ a maximum *cover of f' and g'.*

We denote the set of maximum covers of a set of phase views P' from two phase function views as $\text{maxCover}(P')$*.*

While finding a maximum cover is technically enough to solve our problem, our algorithm computes a leftmost maximum cover. This allows an inductive calculation of a maximum cover in linear time.

Cover relationships can be uniquely sorted from left to right, since they do not cross. From this partial ordering we can define a partial ordering on covers by looking at their rightmost cover relationship.

Definition 3.33 (Leftness of Phase Views)**.** *Let $r' := ([a_r, b_r], c_r)$ and $s' := ([a_s, b_s], c_s)$ be phase function views.*

We define $r' < s'$, if $b_r < a_s$. We call r' left *of s', if $r' < s'$.*

Definition 3.34 (Leftness of Cover Relationships)**.** *Let f' and g' be phase function views. Let $r', s' \in f'$ and $u', v' \in g'$. Let a be a possible cover relationship between r' and u' and b be a possible cover relationship between s' and v'.*

We define $a < b$, if $r' \leq u' \wedge s' \leq v' \wedge (r' < u' \vee s' < v')$. We call a left of b if $a < b$.

Definition 3.35 (Leftness of Covers)**.** *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be covers.*

We define $\mathcal{C}_1 < \mathcal{C}_2$, if $\max \mathcal{C}_1 < \max \mathcal{C}_2$. We call $\mathcal{C}_1$ left of $\mathcal{C}_2$, if $\mathcal{C}_1 < \mathcal{C}_2$.

With this notion of leftness, a leftmost maximum cover of a pair of phase views can naturally be defined.

Definition 3.36 (Leftmost Maximum Cover, $\text{lmc}(\cdot)$)**.** *Let $\mathcal{C}$ be a maximum cover of a set of phase views P' from two phase function views, such that there is no other maximum cover $\mathcal{C}' \in \text{maxCover}(P')$ with $\mathcal{C}' < \mathcal{C}$.*

We call such a $\mathcal{C}$ a leftmost maximum cover (LMC) *of P'.*

We denote an LMC of a set of phase views P' as $\text{lmc}(P')$*.*

3.3.3. LMC Algorithm

We define a few terms to describe the algorithm. We need a notion of intersection for phase views to restrict the list of candidate cover relationships for each phase.

Definition 3.37 (Intersecting Phase Views, Phase View Neighborhood)**.** *Let f' and g' be phase function views. Let $p' \in f'$. Then* $\text{intersecting}(p') := (q' \in g' | \text{interval}(p') \cap \text{interval}(q') \neq \emptyset)$ *are the* intersecting phase views *of p'.*

Let $P' \subseteq f' \cup g'$. Then $\text{intersecting}(P') := (q' \in f' \cup g' | \exists p' \in P' : q' \in \text{intersecting}(p'))$ *are the* intersecting phase views *of P'. And $\mathcal{P} := P' \cup \text{intersecting}(P')$ is the* neighborhood *of P'.*

The algorithm greedily processes phase views in a certain order.

Algorithm 3.4: LeftmostMaximumCover

Input: Phase function views f' and g'
Data: Cover $\mathcal{C}$, cover relationship rs
Output: LMC $\mathcal{C}$ of f' and g'

```
1  foreach p' ∈ reducedPhaseViewOrderSequence(f', g') do
2      if C.isEmpty() then
3          C.append(phaseLMC(p'))
4      else
5          r ← C.last()
6          if r.role(p') ≠ Independent then
7              C.popLast()
8              C.append(phaseLMC(p'))
9          else
10             C.append(phaseLMCWithoutPrevious(p'))
```

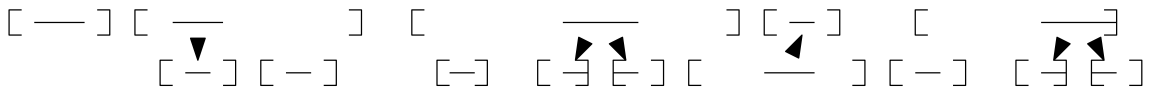


Figure 3.10.: An exemplary leftmost maximum cover for a pair of phase function views. The arrow heads indicate a cover relationship. The cores of the phase views are placed to give visual proof for the validity of the cover.

Definition 3.38 (Phase View Order)**.** *Let f' and g' be phase function views. Let $p', q' \in f' \cup g'$. Let $[a_p, b_p] := \text{interval}(p')$ and $[a_q, b_q] := \text{interval}(q')$. Then, we define $p' < q'$, if $(a_p \leq a_q \land \text{fn}(p') = f') \lor (a_p < a_q \land \text{fn}(p') = g')$, and call this partial order on $f' \cup g'$ the* phase view order *of f' and g'.*

Definition 3.39 (Phase View Order Sequence)**.** *Let f' and g' be phase function views. The* phase view order sequence *of f' and g' is* phaseViewOrderSequence(f', g')*, which is $f' \cup g'$ ordered by their phase view order.*

For the algorithm it is enough to look at a subsequence of the phase view order sequence.

Definition 3.40 (Reduced Phase View Order Sequence)**.** *Let f' and g' be phase function views. Let $P' :=$ phaseViewOrderSequence(f', g') be their* phase view order sequence*. Let P'_i be the consecutive subsequence of P', that starts from the first element in P' and ends with the i-th. Then,* reducedPhaseViewOrderSequence$(f', g') := (p'_i \in P' | \mathcal{P}_i \neq \mathcal{P}_{i-1})$ *is the* reduced phase view order sequence *of f' and g', where $\mathcal{P}_i$ is the neighborhood of P'_i.*

The reduced phase view order sequence skips all phase views that do not add any new phase views to the neighborhood. We do not need them, since our algorithm works by inductively finding LMCs for the current's and all previously processed phase views' neighborhood, and phase views that do not expand the neighborhood do not change the LMC.

We propose Algorithm 3.4 (LeftmostMaximumCover) for calculating an LMC of two phase function views. In Figure 3.10 we give an exemplary LMC for and exemplary pair of phase function views. Cores in the figure are placed to show that the cover is valid.

It uses the subroutine phaseLMC(p') which calculates the LMC of a single phase view with its intersecting phases in linear time. The subroutine phaseLMCWithoutPrevious(p') only considers phase views that are after p' in phase view order. These two subroutines

calculate the LMC by trying all candidates in linear time in the amount of intersecting phase views of p'. For this, the core of p' is shifted from left to right within its interval, and for each positioning, all valid cover relationships are enumerated. The leftmost position with a maximum cover is the returned LMC.

The algorithm runs in linear time $\mathcal{O}(|f'|+|g'|)$. It iterates over the reduced phase view order sequence, which skips phase views p' that intersect another phase view q', such that they are not first or last in intersecting(q'). So PHASELMC and PHASELMCWITHOUTPREVIOUS look at each phase view in reducedPhaseViewOrderSequence at most thrice, once when it is the first intersecting phase of another phase, once when it is the last intersecting phase of another phase and once when it is processed itself. And they look at each phase view not in reducedPhaseViewOrderSequence at most once, when its intersecting phase view is processed. All other operations are in $\mathcal{O}(1)$ and are executed at most $\mathcal{O}(|f'| + |g'|)$ times.

For proving the correctness of this algorithm, we need some definitions.

Definition 3.41 (Variables for this section)**.** *In this section, the following definitions are used:*

- *Let f' and g' be phase function views.*
- *Let $(p'_1, \dots, p'_n) :=$ reducedPhaseViewOrderSequence(f', g') be their reduced phase view order sequence.*
- *Let $P'_i := (p'_1, \dots, p'_i)$.*
- *Let $\mathcal{P}_i := P'_i \cup$ intersecting(P'_i).*
- *Let $S := S_i := \text{rel}(\mathcal{P}_i)$ be the set of possible relationships in $\mathcal{P}_i$ and $\hat{S}$ an LMC of S.*
- *Let $S_{i+1} := \text{rel}(\mathcal{P}_{i+1})$ be the set of possible relationships in $\mathcal{P}_{i+1}$.*
- *Let $T := S_{i+1} \setminus S_i$ be the set of new possible relationships in $\mathcal{P}_{i+1}$ and $\hat{T}$ an LMC of T.*

Note that with these definitions, it holds that $\forall i : \text{rel}(\mathcal{P}_i) \subseteq \text{rel}(\mathcal{P}_{i+1})$. This makes covers stay valid between iterations. The following lemma tells us, that the algorithm considers all relationships between f' and g'.

Lemma 3.42 (Completeness of $\mathcal{P}_n$)**.** *Let f', g' and $\mathcal{P}_n$ be defined as in Definition 3.41. It holds that $\mathcal{P}_n = f' \cup g'$.*

Proof. Clearly, $\mathcal{P}_n \subseteq f' \cup g'$, so let $p'_i \in f' \cup g'$, and we shall show that $p'_i \in \mathcal{P}_n$ by contradiction. Assume, $p'_i \notin \mathcal{P}_n$. Assume, p'_i is the leftmost such phase view. It can not be leftmost in phaseViewOrderSequence(f', g'), since the leftmost is always part of reducedPhaseViewOrderSequence$(f', g') \subseteq \mathcal{P}_n$.

Let p'_j be the leftmost phase in intersecting(p'_i). If p'_j does not exist, then $\mathcal{P}_i \neq \mathcal{P}_{i-1}$, so $p'_i \in$ reducedPhaseViewOrderSequence$(f', g') \subseteq \mathcal{P}_n$. If p'_j does exist, then $\mathcal{P}_j \neq \mathcal{P}_{j-1}$, so $p'_j \in$ reducedPhaseViewOrderSequence(f', g'), so $p'_i \in \mathcal{P}_n$.

So $\mathcal{P}_n = f' \cup g'$. □

The following lemma tells us, that in reduced phase view order sequence, we always process a phase that is at least as right as everything we considered before. We display the two cases of the proof in Figure 3.11.

Lemma 3.43 (Each p'_{i+1} is Rightmost in $\mathcal{P}_I \cup \{p'_{i+1}\}$)**.** *Let f', g', $\mathcal{P}_i$ and p'_{i+1} be defined as in Definition 3.41. Then p'_{i+1} is rightmost in $\mathcal{P}_i \cup p'_{i+1}$.*

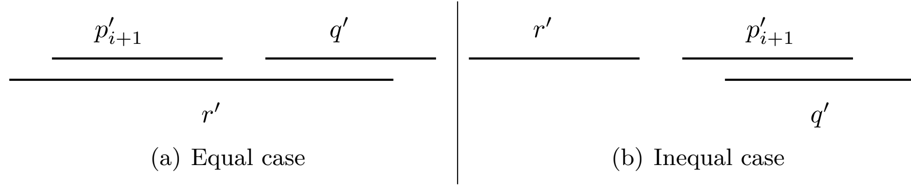


Figure 3.11.: The situation in the two cases of the proof for Lemma 3.43. The lines are the intervals of the respective phase views.

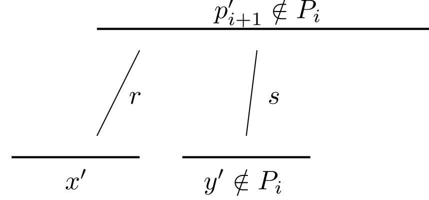


Figure 3.12.: The situation described in the proof of Lemma 3.45.

Proof. Assume for the sake of contradiction there is a phase view $q' \in \mathcal{P}_i$ such that $p'_{i+1} < q'$. Then by Definition 3.41, $q' \notin P_i$, so $\exists r' \in P_i : q' \in \text{intersecting}(r')$. Since $r' \in P_i$ and $p'_{i+1} \in P_{i+1} \setminus P_i$, r' is left of p'_{i+1}.

(a) Assume, $\text{fn}(p'_{i+1}) = \text{fn}(q')$. Then, since r' intersects q' and r' is left of p'_{i+1}, r' intersects p'_{i+1} as well.

    But then, the interval of p'_{i+1} is a subset of the interval of r'. So, $\mathcal{P}_i = \mathcal{P}_{i+1}$, which contradicts the construction the $\mathcal{P}_{i+1}$.

(b) Assume, $\text{fn}(p'_{i+1}) \neq \text{fn}(q')$. Then, $\text{fn}(p'_{i+1}) = \text{fn}(r')$.

    Let x be the start of the interval of p'_{i+1}. Then, since $r' < p'_{i+1} < q'$, the interval of r' ends left of x, and the interval of q' starts at or right of x. But this is a contradiction to $q' \in \text{intersecting}(r')$.

So, in all cases, we found a contradiction to the premise. □

Definition 3.44 (Phase View is Free in Cover)**.** *Let P' be a set of phase views of at most two different phase function views. Let $\mathcal{C}$ be a valid cover of P'. Let $p' \in P'$. Then, p' is* free *in $\mathcal{C}$, if $\forall r \in \mathcal{C} : p' \notin r$.*

The following lemma tells us, that there is only one relationship in contact between the newly considered phases and all old phases, so we have to backtrack at most one relationship in each iteration. Since our covers are always leftmost, backtracking one step keeps the cover maximum on everything before the backtracked relationship. The leftmost property is lost though, but since we add relationships on the right anyways, it is simple to make the result of the iteration leftmost again. The situation in the proof of the following lemma is depicted in Figure 3.12.

Lemma 3.45 (Only Rightmost Relationships Can Contain the Next Phase)**.** *Let $\mathcal{P}_i$, S and p'_{i+1} be defined as in Definition 3.41. Let $r \in S$. Then $p'_{i+1} \in r \Rightarrow r$ is rightmost in S.*

Proof. Assume, there is a relationship $r \in S$, such that $p'_{i+1} \in r$ but r is not rightmost in S. Let s be a rightmost relationship in S. Since $p'_{i+1} \in r$, and, by Lemma 3.43, p'_{i+1} is rightmost in $\mathcal{P}_i \cup p'_{i+1}$, $p'_{i+1} \in s$.

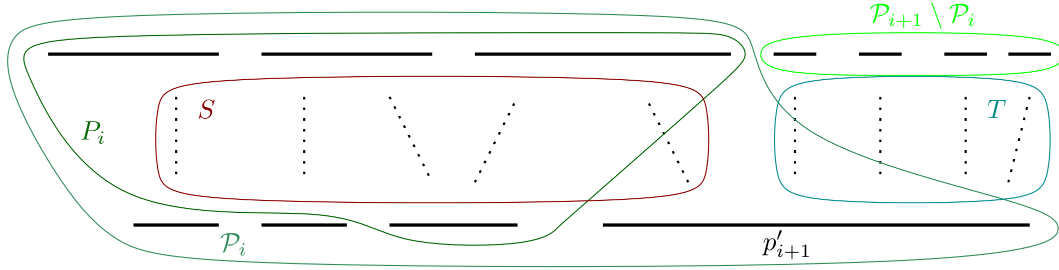


Figure 3.13.: The structure of the set T as proven in Lemma 3.46.

So, there are two phase views $x', y' \in \mathcal{P}_i, x' < y'$ that intersect with p'_{i+1}. Then, $\text{fn}(x') = \text{fn}(y') \neq \text{fn}(p'_{i+1})$, so $x' \cap y' = \emptyset$. But then, the interval of y' has to start inside the interval of p'_{i+1}. But then, y' can not be touched by any phase before p'_{i+1}, so $y' \notin \mathcal{P}_i$. So, there can not be two different phase views in $\mathcal{P}_i$ that intersect p'_{i+1}.

But then, since $p'_{i+1} \in s$, $p'_{i+1} \notin r$. □

Note that there can be up to two rightmost relationships in $\mathcal{P}_i$, since the relationships can have an inverse that is equal in context of the leftness ordering. So when backtracking one relationship we also have to drop its inverse from the domain of the maximum if it exists.

The following lemma tells us how the newly considered relationships in an iteration look like. In Figure 3.13, we depict the structure of T that is proven in the following lemma.

Lemma 3.46 (Structure of T)**.** *Let f', g', $\mathcal{P}_i$, p'_{i+1}, S and T be defined as in Definition 3.41. Let $u \in T$ be a possible relationship. Then*

(1) $p'_{i+1} \in u$,

(2) $\exists q' \in u$ with $q' \neq p'_{i+1}$ and $q' \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$ and

(3) $\forall r \in S : r < u$.

Proof.

(1) Assume there is $u \in T$ such that $p'_{i+1} \notin u$. Then $\exists q' \in u : q' \in \mathcal{P}_{i+1} \setminus (\mathcal{P}_i \cup \{p'_{i+1}\})$.

By definition, $P_{i+1} \setminus P_i = \{p'_{i+1}\}$. So, $\forall q' \in \mathcal{P}_{i+1} \setminus (\mathcal{P}_i \cup \{p'_{i+1}\}) : q' \in \text{intersecting}(p'_{i+1}) \wedge \text{fn}(q') \neq \text{fn}(p'_{i+1})$. So, since by Lemma 3.43, p'_{i+1} is rightmost in $\mathcal{P}_{i+1} \cup \{p'_{i+1}\}$, only the leftmost such q' can intersect a phase view other than p'_{i+1}.

So $\exists q' \in u$ such that q' is leftmost in $\mathcal{P}_{i+1} \setminus (\mathcal{P}_i \cup \{p'_{i+1}\})$ and $\exists r' \in \text{intersecting}(q') : r' \neq p'_{i+1}$. Then, $q' < p'_{i+1}$. But then, when constructing the reduced phase view order sequence of f' and g', q' would be chosen as successor of p'_i instead of p'_{i+1} — a contradiction.

So, $p'_{i+1} \in u$ and thus Statement (1) holds.

(2) Assume there is $u \in T$ such that $q' \in u$ with $q' \neq p'_{i+1}$ and $q' \notin \mathcal{P}_{i+1} \setminus \mathcal{P}_i$. Then, $q' \in \mathcal{P}_i$. Since, by Statement (1), $p'_{i+1} \in u$, it holds that $p'_{i+1} \in \text{intersecting}(q')$.

Since $q' \in \mathcal{P}_i$ and $q' \neq p'_{i+1}$, by Lemma 3.43, it holds that $q' < p'_{i+1}$. But then, when constructing $\text{reducedPhaseViewOrderSequence}(f', g')$, p'_{i+1} would not be chosen as successor of p'_i — a contradiction.

So, $q' \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$ and thus Statement (2) holds.

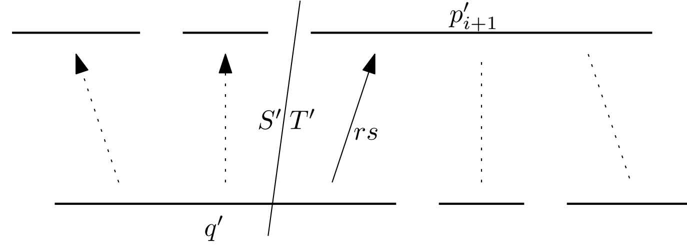


Figure 3.14.: The situation described in Theorem 3.48 in case p'_{i+1} is coverer or coveree in rs. For phase views we only show their intervals, ignoring what the cores might look like. The relationship rs is drawn solid, since it is definitely there in the situation. The other relationships might or might not be there. For the other relationships of q', we know that they have q' as coverer.

(3) Let $u \in T$. Then, by Statement (1) and (2), $q', p'_{i+1} \in u$ with $q' \neq p'_{i+1}$ and $q' \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$.

Since $q' \in \mathcal{P}_{i+1} \setminus \mathcal{P}_i$, if $q' < p'_{i+1}$, q' would be chosen as successor of p'_i instead of p'_{i+1}, when constructing reducedPhaseViewOrderSequence(f', g'). So, since $q' \neq p'_{i+1}$, $p'_{i+1} < q'$. So, since by Lemma 3.43, p'_{i+1} is rightmost in $\mathcal{P}_i \cup \{p'_{i+1}\}$, q' is right of all phase views in $\mathcal{P}_i$. So, u is right of all relationships in S.

So, Statement (3) holds.

□

Lemma 3.47 (Subsets of valid covers are valid covers)**.** *Let $\mathcal{C} \subseteq \text{rel}(\mathcal{P})$ be a valid cover of a set of phase views $\mathcal{P}$. Then, $\forall \mathcal{C}' \subseteq \mathcal{C} : \forall \mathcal{P}' \subseteq \mathcal{P} : \mathcal{C}' \subseteq \text{rel}(\mathcal{P}') \Rightarrow \mathcal{C}'$ is valid cover of $\mathcal{P}'$.*

Proof. Let $\mathcal{C}, \mathcal{C}', \mathcal{P}$ and $\mathcal{P}'$ be defined as in the statement. Assume that $\mathcal{C}'$ is not valid. Then, there is no positioning of the cores of the phase views in $\mathcal{P}'$, such that all relationships of $\mathcal{C}'$ are fulfilled. Take one of these phase views p' that is a coverer, but that cannot cover all its coverees assigned by $\mathcal{C}'$. Let $p'q'_i \in \mathcal{C}'$ be the relationships in which p' is coverer of a $q'_i \in \mathcal{P}'$. Then, adding more relationships to $\mathcal{C}'$ does not allow p' to cover all q'_i. Also, adding more phases to $\mathcal{P}'$ does not allow p' to cover all q'_i.

So, in all cases, $\mathcal{C}$ would be an invalid cover of $\mathcal{P}$, which contradicts the premise. □

The lemmas above are the building blocks to prove the correctness of LeftmostMaximumCover. The basic situation that is described in the dependent case of the theorem below is depicted in Figure 3.14.

Theorem 3.48 (Correctness of LeftmostMaximumCover)**.** LeftmostMaximumCover *calculates a leftmost maximum cover of the phase function views f' and g'.*

Proof. We induct over the reduced phase view order sequence $(p'_1, \dots, p'_n)$ of f' and g'.

We prove, that for each $i \in \{1, \dots, n\}$, LeftmostMaximumCover calculates a leftmost maximum cover of $\mathcal{P}_i$.

$i = 1$: ✓

$i \curvearrowright i+1$: Let $\mathcal{C}_i$ be an LMC of $\mathcal{P}_i$. We construct an LMC $\mathcal{C}_{i+1}$ of $\mathcal{P}_{i+1}$.

Let rs be the rightmost relationship in $\mathcal{C}_i$. If rs does not exist, then $\mathcal{C}_{i+1} := \mathcal{C}_i \cup \text{lmc}(p'_{i+1})$ is LMC of $\mathcal{P}_{i+1}$.

We check the role of p'_{i+1} in rs.

- Assume p'_{i+1} is independent of rs.
  - We prove the validity of $\mathcal{C}_{i+1}$.

    By Lemma 3.45, p'_{i+1} is not part of any other relationship in $\hat{S}$. So, p'_{i+1} is free in $\hat{S}$. Also, by Lemma 3.46, all relationships in T contain p'_{i+1} and one phase view that is part of $\mathcal{P}_{i+1} \setminus \mathcal{P}_i$. So, all phase views in relationships in T are free in $\hat{S}$. So $\mathcal{C}_{i+1} := \hat{S} \cup \hat{T}$ is a valid cover.
  - We prove the maximality of $\mathcal{C}_{i+1}$.

    Assume there exists a valid cover $\tilde{\mathcal{C}}_{i+1} \subseteq \text{rel}(\mathcal{P}_{i+1})$ with $|\tilde{\mathcal{C}}_{i+1}| > |\mathcal{C}_{i+1}|$. Then, because $\text{rel}(\mathcal{P}_{i+1}) = S \cup T$, it follows that $|\tilde{\mathcal{C}}_{i+1} \cap S| > |\hat{S}|$ or $|\tilde{\mathcal{C}}_{i+1} \cap T| > |\hat{T}|$.

    We assume $|\tilde{\mathcal{C}}_{i+1} \cap S| > |\hat{S}|$. But then, by Lemma 3.47, $\tilde{\mathcal{C}}_{i+1} \cap S$ would be a valid cover of S, which contradicts the maximality of $\hat{S}$.

    The case $|\tilde{\mathcal{C}}_{i+1} \cap T| > |\hat{T}|$ is analogous. So, $\mathcal{C}_{i+1}$ is a maximum cover of $\mathcal{P}_{i+1}$.
  - We prove the leftmostness of $\mathcal{C}_{i+1}$.

    Assume, there is a maximum cover $\tilde{\mathcal{C}}_{i+1} \in \text{maxCover}(\mathcal{P}_i)$, that is more left than $\mathcal{C}_{i+1}$. Then, since $\mathcal{C}_{i+1} \cap T = \hat{T}$ is LMC of T, $\mathcal{C}_{i+1} \cap S = \hat{S}$ is not maximum. But it was chosen maximum, so $\mathcal{C}_{i+1}$ is LMC of $\mathcal{P}_{i+1}$.

  The LMC $\hat{S} = \mathcal{C}_i$, and T is the set of relationships taken into account when calculating phaseLMCWithoutPrevious in LeftmostMaximumCover. So, phaseLMCWithoutPrevious calculates $\hat{T}$, so LeftmostMaximumCover calculates exactly $\mathcal{C}_{i+1}$, which is LMC of $\mathcal{P}_{i+1}$.
- Now assume p'_{i+1} is coverer or coveree in rs.

  We define $S' := S \setminus \{rs, sr\}$ and $T' := T \cup (S \setminus S')$. Then, $\mathcal{P}_{i+1} = S' \cup T'$. Also, we define $\hat{S}' := \hat{S} \cap S'$ and $\hat{T}'$ as an LMC of T'. Furthermore, let q' be the partner of p'_{i+1} in rs.
  - We prove the validity of $\mathcal{C}_{i+1}$.

    All cores of phases that are part of relationships of either only $\hat{S}'$ or only $\hat{T}'$ can be placed correctly for $\mathcal{C}_{i+1}$. So, we need to make sure, that cores of phases that have relationships in both $\hat{S}'$ and $\hat{T}'$ can be placed correctly.

    By Lemma 3.46, all relationships in S are lefter than T and all relationships in T contain p'_{i+1}. Since $p'_{i+1} \in rs$, by Lemma 3.45, it holds that rs is a rightmost relationship in S, and that rs is lefter than all relationships in T. From this we can deduct, that
    * all relationships in S' are lefter than T',
    * all relationships in T' contain p'_{i+1},
    * by Lemma 3.45, no relationship in S' contains p'_{i+1} and
    * rs is the unique leftmost relationship in T'.

    Since rs is leftmost in T', phases that have relationships in both S' and T' need to be part of rs. Those phases are q' and p'_{i+1}. But, no relationship in S' contains p'_{i+1}. So, we only need to prove that $\mathcal{C}_{i+1}$ is valid for relationships that include q'.
    * If q' is free in $\hat{S}'$, then $\mathcal{C}_{i+1}$ is valid, because $\hat{T}'$ is valid.

    * If q' is not free in $\hat{S}'$, then q' is in at least one relationship in $\hat{S}'$, and in $rs \notin \hat{S}'$. So, q' is coverer, and p'_{i+1} is coveree in rs.

      But then, since phases do not touch, the length of the core of q' is greater than the length of the core of p'_{i+1}.

      So, since by Lemma 3.46(1), $\forall uv \in T : p'_{i+1} \in uv$, p'_{i+1} is the only phase view that q' can relate with in T'. So, q' cannot be coveree in relationships in $\hat{T}'$.

      So, q' can either be free or a coverer in relationships in $\hat{T}'$.

      · If q' is coverer in a relationship in $\hat{T}'$, then $\hat{T}' = \{rs\}$, so $\mathcal{C}_{i+1} = \mathcal{C}_i$, and thus, $\mathcal{C}_{i+1}$ is valid by induction.

      · If q' is free in $\hat{T}'$, then all phases in relationships in $\hat{T}'$ are free in S', so $\mathcal{C}_{i+1}$ is valid.

  So, in all cases, $\mathcal{C}_{i+1}$ is valid.

- We prove the maximality of $\mathcal{C}_{i+1}$.

  The cover relationship set $\hat{S}'$ is a maximum cover of S', because if it was not, $\hat{S}$ would not be an LMC of S.

  Assume, there exists a valid cover $\tilde{\mathcal{C}}_{i+1} \subsetneq \text{rel}(\mathcal{P}_{i+1})$ with $|\tilde{\mathcal{C}}_{i+1}| > |\mathcal{C}_{i+1}|$. Then, because $\mathcal{P}_{i+1} = S' \cup T'$, it follows that $|\tilde{\mathcal{C}}_{i+1} \cap S'| > |\hat{S}'|$ or $|\tilde{\mathcal{C}}_{i+1} \cap T'| > |\hat{T}'|$.

  We assume $|\tilde{\mathcal{C}}_{i+1} \cap S'| > |\hat{S}'|$. But then, by Lemma 3.47, $\tilde{\mathcal{C}}_{i+1} \cap S'$ would be a valid cover of S', which contradicts the maximality of $\hat{S}'$.

  The case $|\tilde{\mathcal{C}}_{i+1} \cap T'| > |\hat{T}'|$ is analogous. So, $\mathcal{C}_{i+1}$ is a maximum cover of $\mathcal{P}_{i+1}$.

- We prove the leftmostness of $\mathcal{C}_{i+1}$.

  Assume, there is a maximum cover $\tilde{\mathcal{C}}_{i+1} \in \text{maxCover}(\mathcal{P}_i)$ that is lefter than $\mathcal{C}_{i+1}$. Then, since $\mathcal{C}_{i+1} \cap T' = \hat{T}'$ is LMC of T', $\mathcal{C}_{i+1} \cap S' = \hat{S}'$ is not maximum. But it was chosen maximum, so $\mathcal{C}_{i+1}$ is LMC of $\mathcal{P}_{i+1}$.

The cover relationship set $\hat{S}'$ is the set $\mathcal{C}$ in Line 8 after executing POPLAST and T' is the set of possible relationships that the call to PHASELMC considers. So, in this case, LEFTMOSTMAXIMUMCOVER calculates exactly $\mathcal{C}_{i+1}$, the LMC of $\mathcal{P}_{i+1}$.

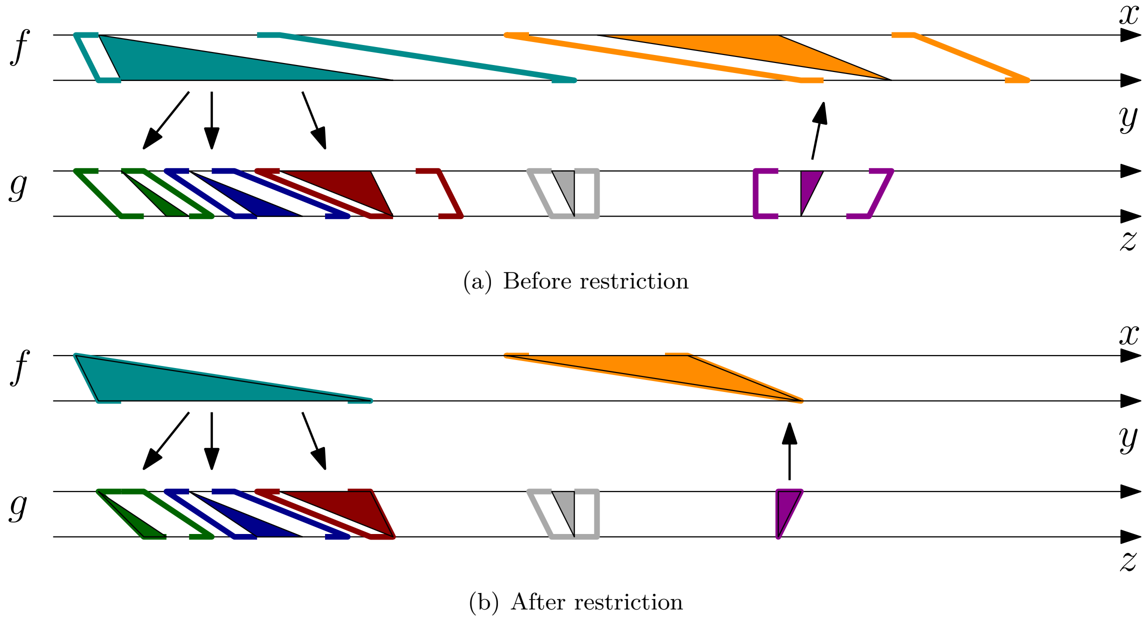
Since a phase can only be coverer or coveree in a relationship or independent of the relationship, the algorithm calculates $\mathcal{C}_{i+1} \in \text{lmc}(\mathcal{P}_{i+1})$ in all cases.

The end result is $\mathcal{C}_n \in \text{lmc}(\mathcal{P}_n)$. By lemma 3.42, $\mathcal{P}_n = f' \cup g'$. So, $\mathcal{C}_n$, as returned by the algorithm, is an LMC of $f' \cup g'$. □

One special case of this algorithm can be handled more efficiently than in the algorithm definition. That is, if p' in Line 1 is coverer in rs, then there exists no more left or better solution that has p' as a coveree. Thus, when calculating the LMC of p' in Line 8, only candidates with p' as a coverer need to be checked.

3.3.4. Computing an Optimal Link

Given two phase functions $f := (P_f, z_f)$ and $g := (P_g, z_g)$. We want to calculate their link $L := f \oplus g$. We first compute their phase function views f' and g' and use them to calculate an LMC with Algorithm 3.4.



(a) Before restriction

(b) After restriction

Figure 3.15.: Exemplary restriction of cover groups.

From this LMC, we extract the cover groups by grouping relationships, that have the same coverer, together. We then restrict the phases inside these cover groups, such that their relationships are enforced in the link. To aid our description of the restriction, we give an example in Figure 3.15.

For this, first, the coverer is restricted. Let the coverers phase view be $p' := ([a, b], c)$. Let its first coverees phase view be $p'_1 := ([a_1, b_1], c_1)$, and its last coverees phase view be $p'_2 := ([a_2, b_2], c_2)$. The coverers phase view core is forced to intersect all coverees phase view intervals by setting $b := \min\{b, b_1 - c_1 + c\}$, and squashing it to the right by setting $a := b - c$.

Then, the coverees are enforced to be inside of p'. This is done by setting $a_1 := \max\{a_1, a\}$ and $b_2 := \min\{b_2, b\}$.

At this point, the cover groups are enforced. The next step is to resolve phase views that are overlapping.

For this, the phase views are iterated in phase view order. If a phase view overlaps its predecessor, then first their phase view intervals are restricted to remove the overlap. Since the LMC algorithm requires phases to not touch within a function, the overlap is removed completely, such that their phase view intervals do not intersect. It might happen that the phase view intervals cannot be restricted enough to remove the overlap.

The restrictions to the phase views are then applied to their corresponding phases.

At this point, the composition of the two phase functions can be computed uniquely. Phases that have slack are either covered or can be copied into the link after shifting their intervals by some offset. Phases that do not have slack have a unique product in the link.

In the composition, phases are iterated in their phase views' order. An offset s for the positioning of phases from g is updated each time a phase from f is processed. The offset starts as the zero z_f of f, and the heights of phases from f are added after they are processed.

Processed phases from f are added to L as they are. Processed phases from g are added to L after subtracting the offset from their interval bounds.

When a phase from a cover group is processed, the whole cover group is processed at once. The heights of the coverees are added to the coverer. Then the coverer is added to L like other phases, depending on if it is from f or g. The offset is updated as if the phases in the cover group were processed normally.

When a phase p is added to L, it might touch its predecessor. This is called a conflict. In this case, since the cover is unique, the phases both do not have any slack. The phases are then separated by reducing their absolute heights by the length of the intersection. In a conflict, one phase always has to be a jump and the other one a constant, because otherwise there would be a cover relationship between them. The jump's interval stays the same, but the constant's interval is reduced to remove all slack introduced by reducing its absolute height. If p is the constant, then it is squashed to the right. If p is the jump, then its predecessor is squashed to the left. This process results in touching phases.

After adding all phases to L, a cleaning step is performed where touching phases are separated, and zero-height phases are removed. Phases are separated by shifting the righter one one unit to the right, or a fixed $\epsilon > 0$ in the continuous case. This might cause phases to intersect in more than one point in a later iteration of the cleaning loop, but those are nevertheless separated.

The zero of the link is the sum of the zeros of the linked functions.

The result of this procedure is a phase function that represents a link of f and g, but with separated phases.

3.4. Merging of Phase Functions

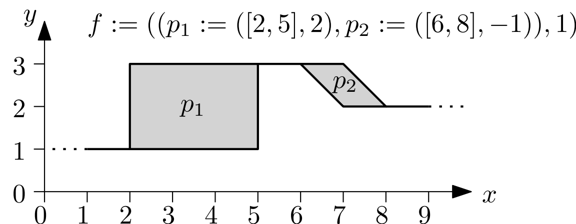


Figure 3.16.: The geometry phase merging is based on. The phases are the union of their traces in travel time representation.

The merging of two phase functions is based on the geometry of their function trace. We calculate the merge interpreting the phase functions as travel time functions. Their phases are two-dimensional shapes on the union of all function traces of all possible phase positionings. Jumps are axis-aligned rectangles, and constants are parallelograms, that are parallel to the x-axis and the function $y = -x$. The phase cores are straight lines within the phase shapes. A jump phase connects to the function on the lower left corner and the upper right corner. A constant phase connects to the function on the upper left corner and the lower right corner. The merge geometry of an exemplary phase function is displayed in Figure 3.16.

When merging, not only phases can cover other phases, but also segments of the function without traffic changes. This is because when minimizing, support points that are above the other function are excluded from the merge. When phase shapes intersect in the two-dimensional space, then it might be possible to shift their cores such that one phase covers the other, or not.

From preliminary experiments we know that merging does not introduce a relevant amount of support points to the shortcut weights. So, we do not optimize the result size when merging, but perform the merge in an arbitrary correct way.

3.4.1. Phase Merge Algorithm

We want to merge two phase functions $f := (P_f, z_f)$ and $g := (P_g, z_g)$. The merge is computed by sweeping over the resulting functions trace from left to right and searching for intersections with phases of one of the functions.

We start at point $(-\infty, z)$, where $z := \min\{z_f, z_g\}$. From there, we search for the first intersection with a phase shape. This phase shape is processed, such that it possibly covers shapes that it intersects or gets covered. The processing is done from left to right as well. Intersecting phases from the other function are iterated from left to right, and phases that can enter a cover relationship with the processed phase are restricted to enforce the cover relationship. The processed phase is restricted as well. If the processed phase is covered, then its coverer becomes the new processed phase, and the old processed phase is not added to the merge. When there are no phases intersecting with the processed one left, then the processed phase is added to the merge. When phases are added, their height might need to be reduced. This is the case if the other function enters the phase shape on its left boundary, or exits the phase shape on its right boundary.

When a phase is processed to completion, then the resulting functions trace is swept again. The lower function might be the same as before, or it might be the other function.

When there are no more phases left to intersect with the swept result, then the algorithm is done. The zero of the resulting phase function is set to z. The restrictions found for each phase are also applied to the input functions.

3.5. Weight Optimization

We use the phase representation with the optimized phase link and the phase merge algorithm to devise a weight optimization scheme. The goal of this scheme is to alter the input weights slightly, to reduce the memory consumption of the TDCCH built with them. There are two variants of this scheme. One is the weight optimization, and the other the expanded weight optimization. We describe those schemes together.

3.5.1. Scheme

We assume we have a preprocessed input graph and a time-dependent metric. We convert the input weight functions into phase functions. For our experiments, we evaluate two different variants of input data. For one, we convert the input weight functions into phase functions, and then feed them into the scheme as they are. For the other, we expand the phase functions after conversion, by shifting the left opening interval bound of each phase, to fix the phases' length to 15 minutes. To do that, given a phase $p := ([a, b], h)$, we set $a := \min\{a, b - 15\text{minutes}\}$.

With these phase functions, we customize the preprocessed graph inexactly. The inexactness stems from the fact, that we restrict phases on links and merges, but we do not propagate these restrictions. Phases can have multiple products in different phase functions in the hierarchy. If we update one of them, then we need to update them all for consistency. Otherwise, the same input phase is restricted differently in different operations, and possibly restricted in conflicting ways. We explain our decision for an inexact Phase Customization in Subsection 3.5.2 below.

When we perform an optimized link operation, we do not only keep the result, but we also keep the changes to the input functions as yielded from the optimization. But, when we perform a merge operation, we do not keep the changes to the input functions, because we never merge functions that we store as weight for some edge directly. They are linked with another function first.

Input edges, which are edges that were not introduced by the CCH preprocessing, are a special case. After executing our scheme, we want to extract the input edges' weight functions, and convert them back to support sequences for exact customization. So, we need them to be valid concerning the allowed slack on their input weights. This means, that we cannot arbitrarily merge into them, because our Customization is inexact, and thus we would then also merge inexactly created phase functions into them. This possibly leads to a weight function, that does not respect the restrictions of the phase intervals in the input data. Since we cannot control in what way the Customization is inexact, our only option is to not merge into input edges' weights.

After doing this customization, we extract the weights on all input edges, and squash their phases to the right. This is done by placing their phase cores as right as possible and then squashing them. The resulting squashed phase functions are converted back to support sequences. These support sequences represent the optimized weights.

3.5.2. Discussion

In both variants of the weight optimization scheme, we only restrict phase functions that are input of a link operation. But we do not propagate changes of the merge following the link to the input functions, or of any further operations performed on the merge result. If we would do that, we would need to be able refer from each phase to its source that produced it, and to all of its products.

If phase functions are not produced from an input function, they are either product of a link, or a set of merge operations. In a merge, phases have at most one source that produced them. In a link, phases might have more than one source. This is the case if a cover group is formed. But then, only the coverer, and the first and last coveree are relevant sources. So a phase has three sources in this case. The elementary link and merge operations take two input functions. But in the CCH, we only store functions that are merges of multiple links. The intermediate results do not need to be stored. Referring to phases that are not stored would pose some difficulties, since they cannot be easily referred to by their location in memory. But phases that are not stored do not require updating, so they can be skipped when propagating restrictions. So, for referring back to the source of a phase, we refer directly to phases that we actually store, such that they have a valid identifier. In effect, a weight function in the CCH has an arbitrary amount of source functions, but each phase in a weight function has at most three source phases.

Furthermore, phases also need to know about their products, since if they are restricted themselves, their products might need to be restricted as well. A phase can have an arbitrary amount of products in an arbitrary amount of phase functions, since it can be used arbitrarily often in other weights. And it can even appear twice in the same weight function, if the weight function is once linked with a very long other weight function and then merged with itself linked with a very short weight function.

We do not store phase sources, products or propagate changes. But as explained further below in 5, this might be a possible future research direction.

4. Experiments

We conduct our experiments on a 2x8-core Intel® Xeon® E5-2680 @ 2.70GHz system without hyperthreading and 252GiB of RAM. We implemented our algorithms in Rust using `Rust 1.34.0-nightly (865cb7010 2019-02-10)` and compiled with link-time-optimization, debug symbols, optimization level 3 and with `target-cpu=native`. All our implementations are single-threaded.

4.1. Implementation

In this section we describe our implementation in detail.

4.1.1. Time-Expansion

While our theoretical results are for non-recurring functions, we interpret our input data as periodic. We only allow support points to have their x-coordinate within day one.

In effect, the evaluation is preceded by a modulo operation to translate the input into day one. And it is succeeded by adding the time that was removed from the input back to the output.

For the link of support sequences, the periodicity means, that support points, that have an x-value outside of day one after linking, need to be translated back into day one. The periodicity also causes the value at the beginning of day one to be the same as at the end of day one. Therefore, the link always has support points within an x-interval of one day, and the support points in the link are still sorted. This x-interval might be shifted though, so the support points need to be wrapped, such that they all lie within day one modulo one day.

The link of phase functions cannot be performed as optimal as in the non-recurring case by our algorithm, but it is still near optimal. We allow phases to only start and end within day one. But when building the phase views, output views are shifted to the right compared to their phases input intervals, and might be shifted out of day one. Those are wrapped such that the input and output phase views all start within day one, except for one possible output view p', that might cross the midnight boundary. This phase view is not wrapped, such that it starts within day one, but ends outside. In this case, p' is processed suboptimally by the LeftmostMaximumCover algorithm. Phase views at the beginning of the second function within day one are covered by p' if possible in a separate

step. Then, the resulting LMC is optimal, if it is restricted to all phase views besides p'. The resulting phase function might contain a phase that ends outside of day one. This phase is split, such that one part stays at the end of day one and the other part is moved to the beginning. Also, it is ensured that the phases at the boundaries of day one do not touch in the periodic interpretation.

For the merges of support sequences, we adapt Algorithm 2.5. Merges do not alter the x-coordinates of their structures, but wrapped functions allow the slope between the last and the first support point to be different to 1. This allows the line segments that cross the midnight boundary to intersect in a single point. We handle this intersection with a separate check.

For merges of phase functions, since phases do not cross the midnight boundary, and their x-coordinates are not altered by the merge, the phase merge algorithm described in Subsection 3.4.1 is applied without modification.

For the efficient implementation of the link and merge algorithms for support sequences, we referred to the open-source version of KaTCH [BGSV13, Bat18] implemented by Batz.

4.1.2. TDCCH

We do a standard metric-independent preprocessing as in [DSW16] to acquire an unweighted CCH graph for the input graph. First, we complete the graph into an undirected graph by adding reverse edges to all directed edges. For general efficiency in the customization and query, we order the nodes by their rank in the nested dissection order. We store the graph topology as adjacency array for all outgoing edges. Since the graph is undirected, we do not need a separate adjacency array for incoming edges. For a more efficient query, we also store a mapping from each edge to its reverse.

The basic schemes for the preprocessing and the customization of our CCH is taken from [DSW16]. The nested dissection order in the preprocessing is calculated by Flow-Cutter [HS15]. We contract the nodes one by one, and insert the undirected shortcuts generated by the contraction into a separate sorted set. The contraction is performed on the union of the graph with the edges from the sorted set. Only from time to time, they are inserted into the adjacency array, which needs to be rebuilt for the insertion. This rebuild happens every 2 million new shortcuts, or, if there are more than 10 000 nodes left to contract, and half of the nodes since the last rebuild have been contracted.

In the customization, we iterate the upward facing edges ordered first by their input node and then by their higher output node. We iterate over the lower triangles of each of these edges.

Support Sequence Customization

As described in Section 2.2.1, we first do a time-independent precustomization, where we customize the CCH one time with the lower bounds of the travel times along the input edges, and one time with the upper bounds. In the main customization, we can then skip links or merges if the result would be completely above the upper bound of its destination edge. The upper bounds are updated after each merge into the edge. We link and merge with the algorithms we describe in Section 2.2. We observe, that merging a function into another, such that the lower bound of one function is higher than the upper bound of the other, can be optimized out, since the result would be the function with the lower upper bound. When linking, functions with just one support point can be linked efficiently, by copying and updating one of the coordinates of the support points of the other function. And when merging, functions with just one support point can be merged more efficiently,

Graph	#Nodes	#Edges	% TD Edges	Avg. Points per TD Edge
Germany 2006	5.1M	11M	7.2%	25.7
Luxembourg 2017	54k	120k	34%	47.4
Germany 2017	7.2M	15.8M	29%	45.5

Figure 4.1.: Statistics about the graphs used for evaluation.

as the function can be viewed as a constant in travel time representation. Linking is more complex in the implementation than in theory, because in the implementation we have to account for the wrapping of time. This means, that, if support points fall out of this one day range, the linked function needs to be wrapped. After linking or merging, we perform a clean operation, to ensure, that the results of merges and links are minimal support sequences.

Phase Customization

The phase customization is similar to the support sequence customization. We first do a precustomization with upper and lower bounds, and update these bounds during the customization. As in the support sequence customization, these bounds allow us to skip links and merges in many cases. We exchange the algorithms for linking and merging support sequences with algorithms for linking and merging phase functions. Those are the algorithms we describe in Section 3.3 and Section 3.4 further above. As in the support sequence customization, links with linear phase functions, that are phase functions without phases, can be optimized. The zero of the phase function with phases needs to be updated in this case, and the phase intervals require shifting, if the phase function with phases is the second in the link. As we describe in Section 3.5, we restrict the input functions of the link and merge operations.

Query

We do a standard elimination tree query as in [DSW16] without early abort condition. We cannot do a reverse search though, since we do not know the arrival time at the destination node in advance. Alternatively, we first search the elimination tree upwards from the destination node. During the search, we collect all downward-facing input edges for each visited node. Then we do an upward search from the source node s. Last, we do a downward search using the collected edges. As described in Section 2.2.1, when relaxing an edge (u, v), we only evaluate its weight function f if $l(u) + b \leq l(v)$, with b being the lower bound of the travel time representation of f and $l(\cdot)$ being the current lower bound travel time from s.

4.2. Results

We test our algorithm on three different input graphs. One of them is an old time-dependent graph of the German road network from 2006. The other two are new time-dependent graphs with travel time functions for cars from 2017. One is for Germany, the other for Luxembourg. The data was kindly provided by the PTV Group for research purposes. Due to the small size of the country, we refer to the Luxembourg graph is a city-sized instance. The Germany graphs are country-sized instances. We show some statistics about the graphs in Table 4.1. The Germany 2006 graph has long segments with ascend not equal to one in arrival time representation. As defined in Definition 2.25, we call those segments traffic changes. When we convert them to phases, we do not limit their maximum length. This makes the Germany 2006 graph exert slightly different behavior in our experiments, besides being less time-dependent. A continent-scale instance of Western Europe does not

fit into the 252GiB of RAM of our hardware when trying to customize it, so we exclude it from our experiments.

We use three different variants for the customization, and four different resolutions for the weight functions. The default variant of the customization uses the input data as-is, but restricts slopes to 0, 1 and ∞ by squashing the traffic changes. The optimized variant does a basic weight optimization step as described in Section 3.5 in advance, while using phase functions as they are. The expanded variant is similar to the optimized variant, but it expands the phases to the left, such that the phase intervals have a length of 15 minutes. We use the resolutions 1, 10, 100 and 1 000 milliseconds.

For each of the three input graphs, we generate 10 000 shortest path queries for each Dijkstra rank $2^6, 2^7, 2^8, \dots, 2^r$, where $r := \max\{r \mid 2^r \leq n\}$, where n is the amount of nodes in the graph. For each of these queries, we measure the resulting exact minimum travel time with Dijkstra in the input graph with *unsquashed* traffic changes, and use the results as ground truth for our error plots. Using unsquashed traffic changes as the ground truth means that queries for the default variants are inexact as well.

4.2.1. Customization

We customize each input graph once for each variant of the customization and for each resolution. The customization times and the sizes of the customized graphs are displayed in Tables 4.1(a) to 4.1(c).

For both the Luxembourg and the Germany 2017 graph, we note that the optimized variant yields slight advantages in terms of memory consumption. The customization time doubles compared to the default variant though. This is mainly caused by the weight optimization step taking so long. The exact customization takes less time than before, because the functions that are linked and merged are less complex.

While the optimized variant gives the traffic changes a little slack, the expanded variant sets this slack to 15 minutes for every traffic change. This yields a much better memory consumption with reductions of up to 50%. The customization time also improves because of the lower function complexity of weights on overlay edges that consist of many original edges.

For the Germany 2006 graph, we note that both the optimized and the expanded variant yield good improvements in terms of memory consumption. The customization time is worse in both cases. The optimized and the expanded variant are nearly equal, because most phases in the optimized variant are 15 minutes long without expansion. The customization of the Germany 2006 graph is much faster than the customization of the Germany 2017 graph in all cases, because it contains much less time-dependent edges, and its time-dependent edges are less complex. Compared to the much lower complexity of the Germany 2006 graph, its memory savings of 41.3% seem quite high, keeping in mind that the Germany 2017 graph saves 50.6% at most.

The resolution has a strong effect on the results. Travel times along single edges are usually short. Non-zero edges have an average travel time of 12.3 seconds at midnight in the Germany 2017 graph. Also, the absolute heights of traffic changes are very small, with an average of 0.23 seconds. This causes the function complexity to be greatly reduced by setting the resolution to 1 or 0.1 seconds. And, this also explains, why there is next to no difference in space consumption between a resolution of 10ms and 1ms. They are both well below the resolution of the input data.

By using a resolution of 1 second and the default variant, we save 78.7% of space compared to a resolution of 1ms for the Luxembourg default variant, and 84.8% for the Germany

Table 4.1.: Customization times and sizes of the customized graphs. O is the time for the weight optimization and E is the time for the exact customization with the optimized weights. The phase space is the memory consumed by the customized phase TDCCH after the weight optimization step. The unit KiB/n means Kibibytes per node. The space reduction is calculated for each variant compared to the default variant with the same resolution.

(a) Luxembourg 2017

Variant		Time [s]	Time O + E [s]	Phase Space [KiB/n]	Space [KiB/n]	Space Reduction [%]
Default	1s	3.4	-	-	2.39	0
	100ms	11.8	-	-	8.51	0
	10ms	16.4	-	-	11.04	0
	1ms	15.3	-	-	11.21	0
Optimized	1s	8.7	5.5+3.2	1.67	2.25	5.5
	100ms	27.0	17.0+10.0	4.91	7.24	14.9
	10ms	34.4	21.0+13.4	5.95	9.01	18.3
	1ms	33.3	20.7+12.6	6.01	9.08	19.0
Expanded	1s	6.7	3.7+3.0	1.18	2.07	13.5
	100ms	17.4	9.7+7.7	2.98	5.59	34.3
	10ms	21.6	12.1+9.5	3.57	6.47	41.4
	1ms	20.7	11.8+8.9	3.58	6.50	42.0

(b) Germany 2017

Variant		Time [h:m]	Time O + E [h:m]	Phase Space [KiB/n]	Space [KiB/n]	Space Reduction [%]
Default	1s	0:40	-	-	4.17	0
	100ms	2:55	-	-	18.42	0
	10ms	4:37	-	-	26.65	0
	1ms	4:51	-	-	27.45	0
Optimized	1s	1:39	1:01+0:38	2.77	3.96	4.9
	100ms	6:16	3:48+2:28	9.99	15.68	14.9
	10ms	8:52	5:11+3:41	13.12	21.23	20.3
	1ms	8:55	5:08+3:47	13.34	21.69	21.0
Expanded	1s	1:12	0:38+0:34	1.84	3.61	13.3
	100ms	3:42	1:58+1:44	5.46	11.27	38.8
	10ms	4:50	2:36+2:14	6.95	13.44	49.6
	1ms	4:52	2:35+2:17	7.03	13.57	50.6

(c) Germany 2006

Variant		Time [min]	Time O + E [min]	Phase Space [KiB/n]	Space [KiB/n]	Space Reduction [%]
Default	1s	21	-	-	3.50	0
	100ms	37	-	-	6.07	0
	10ms	39	-	-	6.27	0
	1ms	39	-	-	6.27	0
Optimized	1s	36	20+16	1.46	2.61	25.3
	100ms	51	29+22	2.10	3.62	40.3
	10ms	52	30+22	2.14	3.68	41.3
	1ms	52	30+22	2.14	3.68	41.3
Expanded	1s	36	20+16	1.46	2.61	25.3
	100ms	51	29+22	2.10	3.62	40.3
	10ms	52	30+22	2.14	3.68	41.3
	1ms	52	30+22	2.14	3.68	41.3

Variant		Luxembourg 2017	Germany 2017	Germany 2006
Default	1s	78.7	84.8	44.2
	100ms	24.1	32.9	3.2
	10ms	1.5	2.9	0.0
	1ms	0.0	0.0	0.0
Optimized	1s	75.2	81.7	29.1
	100ms	20.3	27.7	1.6
	10ms	0.8	2.1	0.0
	1ms	0.0	0.0	0.0
Expanded	1s	68.2	73.4	29.1
	100ms	14.0	16.9	1.6
	10ms	0.5	1.0	0.0
	1ms	0.0	0.0	0.0

Table 4.2.: Total space saved in percent by reducing the resolution, per graph and customization variant.

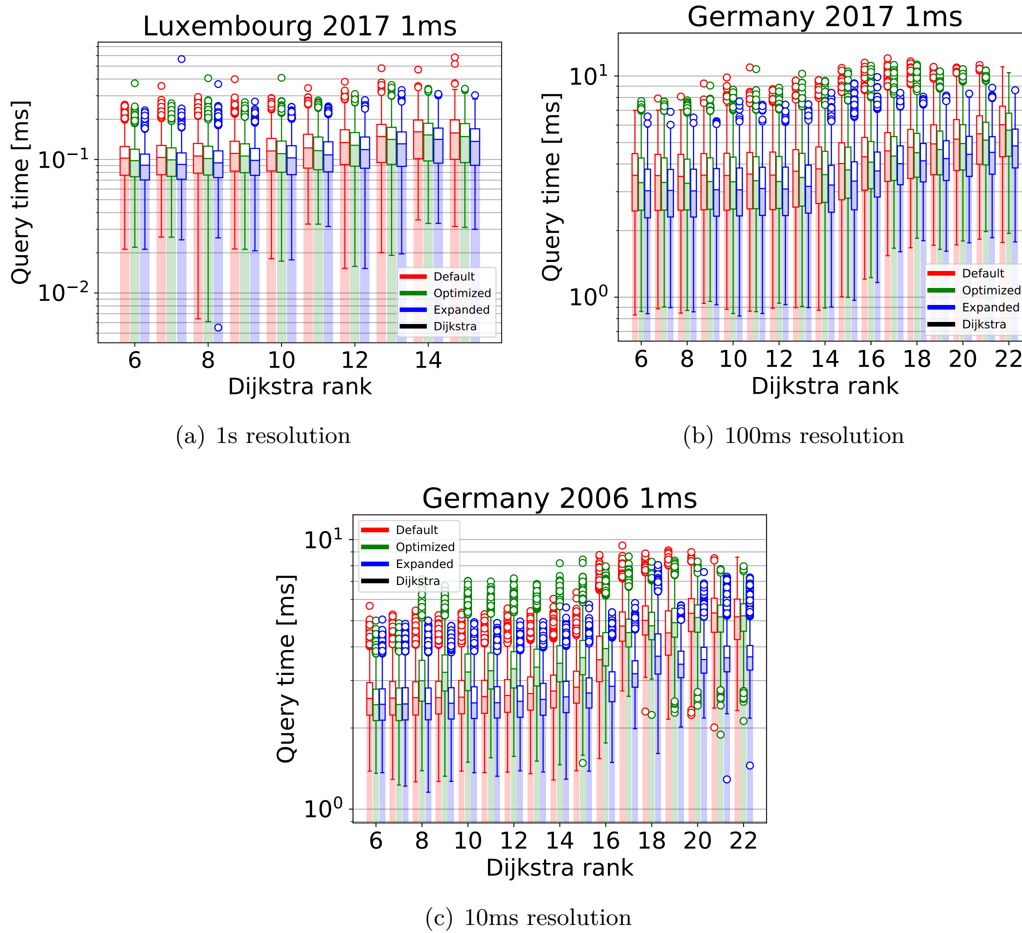
2017 default variant, as displayed in Table 4.2. For the Germany 2006 graph, the same comparison shows a saving of 44.2% of space. The savings for the Germany 2006 graph are lower than the savings for the other two graphs, because the Germany 2006 graph is less time-dependent. For the 2017 graphs, reducing the resolution has a much stronger effect than using the expanded customization with 1ms resolution. But, as analyzed below, using a resolution of 1 second introduces a median query error of about 5% for all three customization variants for the Luxembourg graph, while using a resolution of 1 or 10ms keeps those errors below 0.1% for all three variants. A resolution of 0.1s still keeps the median errors below 1%. The median errors on the Germany 2017 graph are of similar nature. Reducing the resolution from 1ms to 100ms reduces absolute space by another 17.0% for the expanded variant of Germany 2017. For the Germany 2006 graph, the median errors are similar again, but with around 1% significantly higher for higher resolutions. The higher error is caused by the long traffic changes of this graph, that introduce higher errors when squashed. Also, the 1.6% of space saved by reducing the resolution from 1ms to 100ms in the expanded variant is much lower than for the Germany 2017 graph. This is caused by the lower time-dependence of the Germany 2006 graph.

We also measure the size of the customized phase TDCCH for all three graphs. It is lower than the size of the customized support sequence TDCCH. The difference cannot be explained by the different representation, as converting a support sequence TDCCH to a phase TDCCH must not reduce size by more than 25%, as explained in Section 3.1.3. But it can be explained by the fact that we do not propagate phase restrictions in the weight optimization, so products of the same phase might be restricted differently.

4.2.2. Query

For each customized graph, we execute the generated queries and measure the runtime and the relative error. The plots in this section are scaled small, such that they do not consume too many pages and related plots fit onto one page. This causes the readability of the plots to suffer. Therefore, in Appendix A, we include larger versions of the plots for each variant and for each resolution.

There are time and error plots. The time plots use the unit milliseconds and log scale. We give boxplots for each customization variant and Dijkstra rank. We also run Dijkstra's algorithm without any speedup techniques on the squashed input graph with the respective resolution. This variant is called Dijkstra. We do not include the times for Dijkstra in the



(a) 1s resolution

(b) 100ms resolution

(c) 10ms resolution

Figure 4.2.: Query times on the different graphs with 1ms resolution.

small plots in this section, since they reduce the readability, and its runtime simply scales linear with the Dijkstra rank. Refer to Appendix A for the complete plots. There is one time plot per input graph and per resolution.

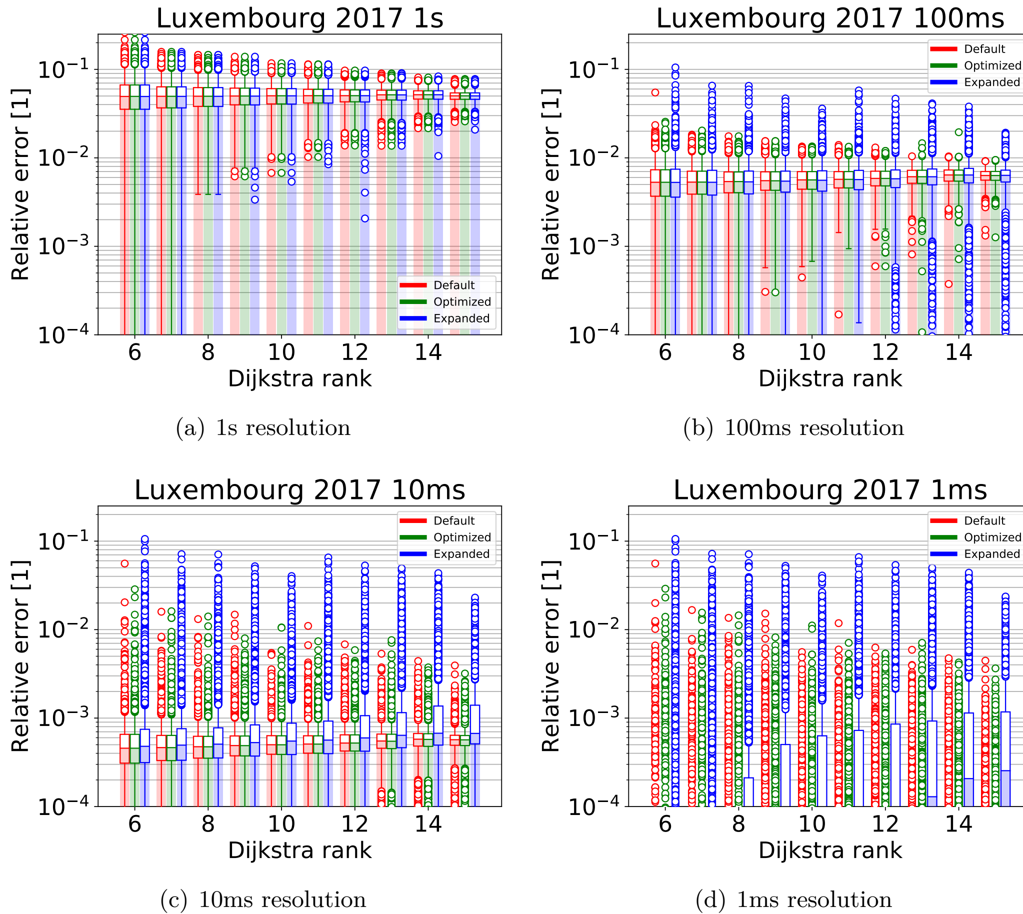
The error plots show the relative deviation to the ground truth and use log scale. We give boxplots for each customization variant and Dijkstra rank. We choose log scale, because our median and maximum errors differ by more than two orders of magnitude per Dijsktra rank, and we want to show this difference accurately. As for the time plots, there is one error plot per input graph and per resolution.

In both plot variants, the boxplots are calculated normally and then transformed into log scale, opposing to first taking the logarithm of all values and then calculating the boxplot. The whiskers of the boxplots show the lowest and highest date still within 1.5 times interquartile range. This is a common variant of boxplots introduced by [Tuk77].

Query Times

The query times do not vary significantly between the different resolutions for a single graph, so we only show the highest resolution here in Figure 4.2. The other plots are shown in the appendix.

As displayed in Figure 4.2(a), the query times for the city-scale instance are very good, even for the highest resolution. Comparing with other resolutions, the query times are about the same, as visible in the appendix in Figures A.1 to A.3. While Dijkstra has a



(a) 1s resolution

(b) 100ms resolution

(c) 10ms resolution

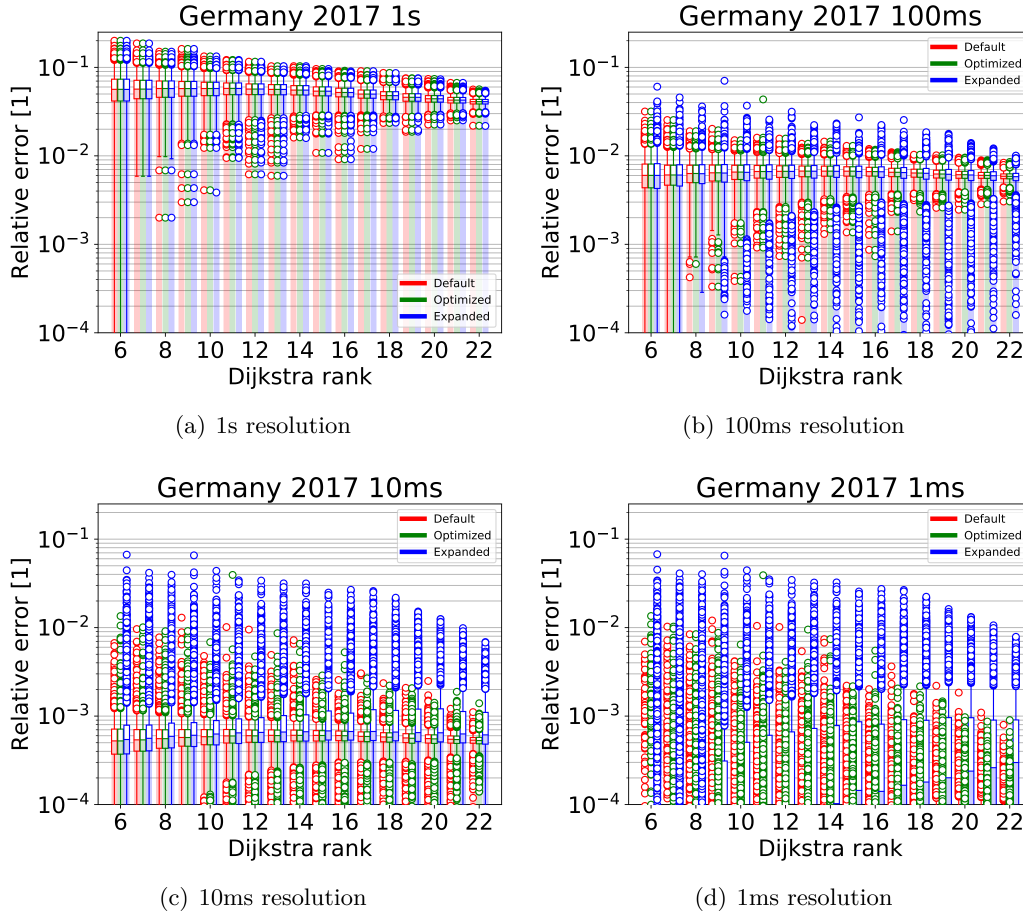
(d) 1ms resolution

Figure 4.3.: Query errors on the Luxembourg 2017 graph with different resolutions. We cut off each plot at 0.01%, as we consider errors lower than that as not relevant, especially compared to the maximum errors.

runtime linear to the Dijkstra rank, the elimination tree queries have the same runtime independent of the rank of the query. This is caused by having no early abort condition for the elimination tree queries, but always searching the whole elimination trees. The query times are not spread out a lot, but some variance is visible. The query times for the Germany 2017 graph are plotted in Figure 4.2(b) and in the appendix in Figures A.9 to A.11. The median query time is about six milliseconds and the speedup more than two orders of magnitude. The query times for the Germany 2006 graph are plotted in Figure 4.2(c) and in the appendix in Figures A.17 to A.19. The median query time of about four milliseconds is a little bit lower than the median query time for the Germany 2017 graph. This is caused by the lower time-dependence of the graph.

Query Errors

The median errors for the Luxembourg graph with a resolution of 1s are around 5%, but fall below 1% for 100ms and below 0.1% for 10ms for all three variants, as displayed in Figure 4.3. For the Germany 2017 graph, the errors exert the same behavior. We plot them in Figure 4.4. For the Germany 2006 graph, the errors exert a similar behavior, but the median errors, even for the highest resolution, are at 1%, as displayed in Figure 4.5. This is due to the long traffic changes in this graph, that introduce larger errors when being squashed than the short traffic changes of the other two graphs.



(a) 1s resolution

(b) 100ms resolution

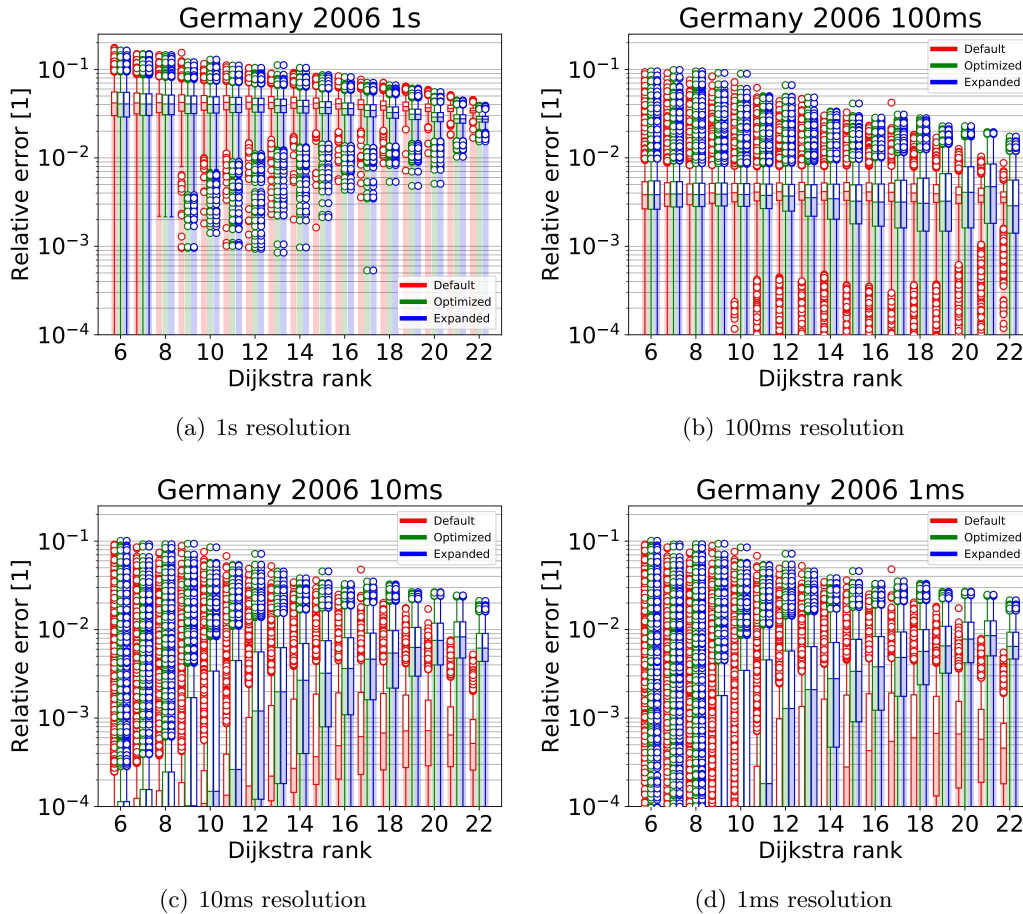
(c) 10ms resolution

(d) 1ms resolution

Figure 4.4.: Query errors on the Germany 2017 graph with different resolutions. We cut off each plot at 0.01%, as we consider errors lower than that as not relevant, especially compared to the maximum errors.

In the Luxembourg graph, the maximum query errors are as high as 5%, even for long queries for the expanded variant with 1ms resolution, while being below 0.5% for the default variant, as shown in Figure 4.3(d). This strong deviation even for long queries possibly hints to some system behind the errors as we discuss in Section 4.3. These potentially systematic errors seem to happen only rarely, because the 75-percentile of relative errors is 0.1% for the expanded variant. The maximum errors on the Germany 2017 graph have the same properties, while being a little bit lower overall as visible in Figure 4.4(d). The maximum errors on the Germany 2006 graph only exert these properties for long queries, but the graph is less time-dependent, which weakens any systematic errors. This is visible in Figure 4.5(d).

Looking at the maximum errors for lower resolutions in Figures 4.3(a) to 4.3(c), it is visible, that the potentially systematic errors are mostly dominated by the errors introduced by a resolution of 1 second. But for a resolution of 100ms, there are a lot of outliers in the boxplots already, and especially more extreme outliers for the expanded variant than for the default variant. This hints that a resolution of 100ms is too high to hide the potentially systematic errors. The outliers become more dominant for higher resolutions. The maximum errors on the Germany 2017 graph for lower resolutions have the same properties again, while being a little bit lower overall, as displayed in Figures 4.4(a) to 4.4(c). Again, the maximum errors on the Germany 2006 graph for lower resolutions only exert these properties for long queries, as visible in Figures 4.5(a) to 4.5(c).



(a) 1s resolution

(b) 100ms resolution

(c) 10ms resolution

(d) 1ms resolution

Figure 4.5.: Query errors on the Germany 2006 graph with different resolutions. We cut off each plot at 0.01%, as we consider errors lower than that as not relevant, especially compared to the maximum errors.

4.3. Comparison & Evaluation

In this section, we evaluate our results and compare them to those of other authors. The expanded variant with a resolution of 100ms is the most competitive variant of our scheme. So we evaluate this variant against others.

In this variant, the TDCCH uses 11.27B/n for the Germany 2017 graph and takes 4:52 hours to customize. The median query time is around 5ms, with a median error of around 0.7%, and a maximum error of around 4% for longer queries.

In Section 4.3.1, we compare our results to those of other authors. In Section 4.3.2, we compare our results to the TCH, which is the most similar acceleration technique to the TDCCH that already exists, and that might benefit from our results. In Section 4.3.3, we argue about the errors our technique introduces.

4.3.1. Other techniques

Conveniently, the authors of the publications we introduce in Section 1.1 all evaluate their algorithms on the Germany 2006 graph. So we can compare their results with ours, assuming that they did their experiments on similar hardware than we did. The publications we took their results from are all younger than ten years, and our hardware was released around five years ago, so we suspect that the difference is not too high.

With exact TD-ALT, query times of 100ms can be achieved [NDLS12]. Combining TD-ALT with contraction (Core-TD-ALT) yields query times of 5ms with a preprocessing time of 9 minutes and a memory consumption of 50B/n [NDLS12]. Combining Arc-Flags with a hierarchy (TD-SHARC) yields query times of 25ms with a preprocessing time of 76 minutes and a memory consumption of 150B/n [Del11]. Adding landmarks (TD-L-SHARC) yields query times of 6ms with a preprocessing time of 78 minutes and a memory consumption of 220B/n. The TCH yields query times of 1.2ms with a preprocessing time of 25 minutes or 5 minutes with 8 threads and a memory consumption of 1KiB/n [BGSV13]. The ATCH yields query times of 1.5ms with a preprocessing time of 25 minutes or 5 minutes with 8 threads and a memory consumption of 120B/n [BGSV13]. Using a distributed variant of the customization of a TDCH yields a preprocessing time of less than 3 minutes using 64 processes [KLSV10].

All these techniques calculate queries exactly. TDCRP, the time-dependent expansion of CRP, uses approximation that leads to errors in time-dependent shortest path queries of 0.68% on average and a maximum error of 3.6% [BDPW16]. It allows for query times of 1.2ms with a preprocessing time of 8 seconds using 16 threads and a memory consumption of 77B/n.

Of these techniques, Core-TD-ALT, TD-L-SHARC and the variants of the TDCH are comparable to or better than our technique in terms of query times. We suspect that by optimizing our query more, we would get query times around those of the ATCH, but definitely better than Core-TD-ALT and TD-L-SHARC. But the big advantage of the two goal-directed techniques is the low memory consumption compared to the TDCCH, and Core-TD-ALT has a really fast preprocessing. We talk about the TDCHs more in Section 4.3.2.

TDCRP is another direct competitor, but it is not exact. Still, our maximum query errors are slightly higher than those of TDCRP, while our median is similar to their average. So, TDCRP seems to be on the same level error-wise. But, we are able to simply characterize and restrict our errors. TDCRP uses much less memory and customizes within 8 seconds on 16 cores, which is much lower than our customization time, even when rolled out to $8s * 16 = 128s$ for one core.

Concluding this subsection, for the old data, TDCRP and the ATCH seem to be the techniques of choice for time-dependent road network routing. TDCRP is superior to the ATCH in terms of stability against difficult metrics and in terms of dynamic routing capabilities. The ATCH is superior to TDCRP in terms of query errors, as it is exact.

4.3.2. TCH

Even though we successfully reduced the memory usage of the customized TDCCH, it does not compete against the TCH [BGSV13]. We evaluated their open-source implementation of the TCH [Bat18] on our machine. Our experiments show, that the TCH uses 4.2KiB/n of memory for the Germany 2017 graph with 100ms resolution and its preprocessing takes 79 minutes without parallelization, while the TDCCH uses 11.3KiB/n of memory in the expanded variant with the same resolution. We did not optimize our query algorithm as much as [BGSV13], so it is unknown if we could beat their query times. Also, the TCH does not restrict slopes to 0, 1 and ∞, and thus potentially solves a more complex problem.

The TCH achieves lower memory consumption by optimizing the contraction order greedily, by contracting out those nodes first, that produce a low amount of support points and overlay edges, together with other criteria. Contrary, the TDCCH cannot do that, as it determines the order without knowledge of the metric. The TCH beats the TDCCH in the customizable aspect, as its whole preprocessing is faster than the customization of the

TDCCH on the same graph. But, the TDCCH has a metric independent preprocessing, that makes it more robust against difficult metrics. And, the TDCCH is efficient in terms of inserted edges, as long as the input graph has small separators. On the contrary, CHs have proven to be less robust against difficult metrics, including metrics with turn costs [DGPW11], so the TCH inherits the lack of robustness.

It is an open question if restricting slopes and applying our optimization scheme would improve the TCH significantly in terms of preprocessing time or memory consumption.

4.3.3. Potentially Systematic Errors

Giving the phases a lot of slack produces much more significant errors than simply squashing them on the 2017 graphs. We did not conduct experiments to figure out what effects lead to high errors. But we have some hypotheses.

The far outliers might be caused by the higher variation of errors that we introduce by shifting the traffic changes instead of squashing them in-place. A traffic change might be up to 15 minutes off, and if a query hits multiple such cases, then it becomes significantly erroneous. This effect causes a random variation of the errors.

Also, the more erroneous queries might be those, that run during the rising traffic of the morning rush our, or the falling traffic of the evening rush hour. At these times, the amount of traffic changes that influence the query is especially high. This effect causes a stronger variation in the query errors.

These two effects do exist, but we did not conduct experiments to measure their strength. They also do not introduce systematic errors, despite errors being higher at certain times of day.

There is a candidate effect for introducing more systematic errors. In a real-world road network, consecutive road segments of main streets usually have correlated traffic. This means, that there are paths, that all have traffic changes at for example 8 o'clock in the morning. In our optimization scheme, we squash them. The result of that is displayed in Figure 4.6(a). When performing the optimized link, if the edges of the path are processed consecutively, their 8 o'clock traffic changes will be moved on top of each other. This is displayed in Figure 4.6(b). Then, when linking the complete path, the shortcut gets only one traffic change for the 8 o'clock traffic changes of all road segments on the path. In the figure, it is visible, that for the traffic changes to actually cover each other, the first one needs to be earlier than the last one. This might make single queries hit a lot of weight functions at an erroneous departure time. We call this effect the *systematic path effect*.

If our optimization scheme does not optimize this one path, a situation similar to that in Figure 4.6(c) might occur. This can be caused by actually placing the traffic changes at random, or optimizing for other links than this one path. This is the situation we would want for non-independent errors, but not for an optimal solution in terms of space consumption.

We did not conduct experiments to verify the strength of the systematic path effect. But this is an effect that does introduce systematic, non-independent errors, that would void our argument for the accuracy of our technique, as discussed in Section 2.1.2. We stated, that our technique can only be called accurate, if it only introduces minor random independent noise into the input functions. This gives us a direction for future research, as it might prove or disprove the accuracy of our technique.

Nevertheless, our data shows, that the amount of systematic errors is small, if they exist. So our technique is accurate enough in practice.

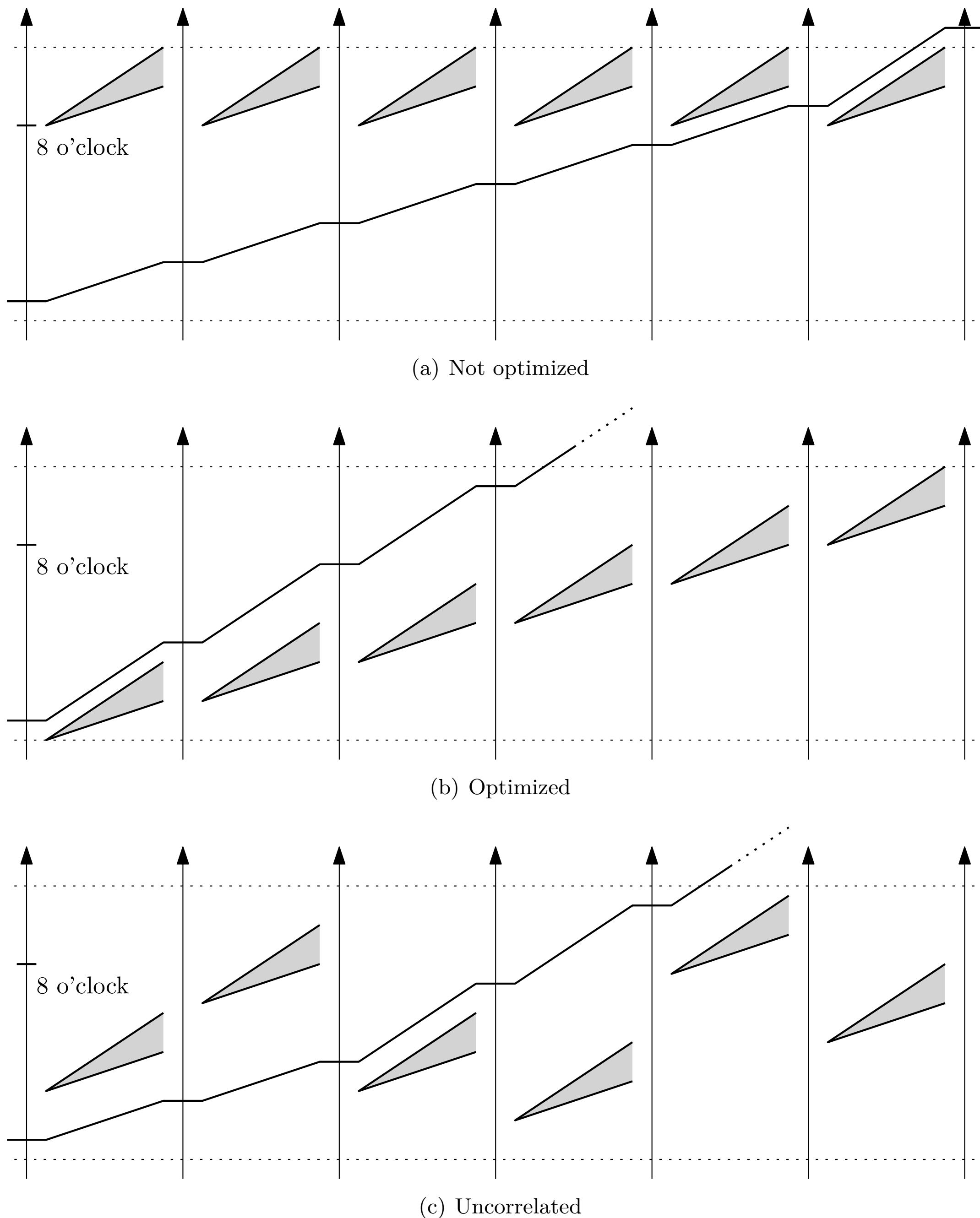


Figure 4.6.: A path with squashed phases that can cover each other. Phases are displayed as grey triangles. In Subfigure (a), the phases' positions are not optimized. Driving along the path is fast in that case, as visible by the solid line from left to right. In Subfigure (b), the phases' positions are optimized. In this case, departing at the same time at the start of the path yields a much higher travel time than in the unoptimized variant. But, when linking the path, the optimized variant yields just one phase, while the optimized variant does not allow any cover relationships. In Subfigure (c), the phases' positions are changed at random. So their relative position towards each other is uncorrelated. This yields less cover relationships than the optimized variant, but also lower errors.

5. Conclusion & Outlook

In this work, we expanded the CCH with time-dependent edge weights. We examined it from a theoretical and a practical perspective.

Our theoretical results include a proof for the correctness of an algorithm to link non-decreasing and non-continuous piecewise linear functions. We introduced the phase model and used cover potentials to examine how support points and phases behave on link. With the knowledge gained from the cover potentials, we proposed an algorithm to link functions in the phase model with near-optimal size. Using this algorithm, we introduced an optimization scheme that modifies the input weight functions within a user-defined threshold. We evaluated this scheme in practice.

The optimization scheme reduces the memory consumption of the TDCCH by up to 50% on country-scale instances, while keeping queries efficient and query errors small. We argued that it is okay to introduce more noise into the input data, as long as it does not dominate the noise it already contains. Under this aspect, we reduced the memory consumption of the TDCCH while keeping queries exact. In the current state our TDCCH does not achieve competitive performance to the TCH or TDCRP, both in terms of memory consumption and customization time.

In the future, we would like to examine, if the errors introduced into the input data by the optimization scheme are systematic. And, we would like to apply our weight optimization scheme to the TCH, and assess, if it also yields improvements. Moreover, we also would like to compare a TDCCH with restricted slopes to a TDCCH with unrestricted slopes, to figure out how much memory is saved by restricting slopes alone.

For the theoretical part, we introduced a very simple optimization scheme, that opens up a lot of possibilities for improvements. We would like to propagate phase restrictions and make the link algorithm work with touching phases as well, to allow for an exact phase customization. For this, we would like to examine cover potentials in more detail. It is unclear how an exact phase customization would affect customization times. On one hand, we could drop the exact support sequence customization step, and thus make the overall customization time faster. On the other hand, it might also increase customization times, as at the moment linking phase functions is slower than linking support sequences, and the inexact phase customization leads to smaller functions than an exact one would. It might also be possible to abort the weight optimization early, for example by ignoring weight functions with phases that do not have slack, possibly reducing customization time. Another direction for improvement is computing optimal weights not just for a single link,

but also for paths. And, including the merge operation as well, compute optimal weights for directed acyclic graphs and general graphs.

Bibliography

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, Lecture Notes in Computer Science, pages 230–241. Springer Berlin Heidelberg, 2011.

[Bat18] G. Veit Batz. KaTCH – Karlsruhe Time-Dependent Contraction Hierarchies: GVeitBatz/KaTCH, December 2018. original-date: 2015-07-02T14:58:43Z.

[BD08] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. In *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, Proceedings, pages 13–26. Society for Industrial and Applied Mathematics, January 2008.

[BDG+16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering: Selected Results and Surveys*, Lecture Notes in Computer Science, pages 19–80. Springer International Publishing, Cham, 2016.

[BDPW16] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic Time-Dependent Route Planning in Road Networks with User Preferences. In Andrew V. Goldberg and Alexander S. Kulikov, editors, *Experimental Algorithms*, volume 9685, pages 33–49. Springer International Publishing, Cham, 2016.

[BGSV13] G. Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *Journal of Experimental Algorithmics*, 18:1.1–1.43, December 2013.

[CH66] K. Cooke and E. Halsey. The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times. *Journal of Mathematical Analysis and Applications*, 14(3):493–498, 1966.

[Dea04] Brian C Dean. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. page 13, 2004.

[Del11] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, May 2011.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

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

[Dre69] Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. *Journal of Experimental Algorithmics*, 21(1):1–49, April 2016.

[FHS11] Luca Foschini, John Hershberger, and Subhash Suri. On the Complexity of Time-Dependent Shortest Paths. In *Proceedings of the 22nd Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'11)*, pages 327–341. SIAM, 2011.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A Search Meets Graph Theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics. event-place: Vancouver, British Columbia.

[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, *Experimental Algorithms*, Lecture Notes in Computer Science, pages 319–333. Springer Berlin Heidelberg, 2008.

[Gut04] Ronald J Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. *ALENEX/ANALC*, 4:100–111, 2004.

[HS15] Michael Hamann and Ben Strasser. Graph Bisection with Pareto-Optimization. *arXiv:1504.03812 [cs]*, April 2015. arXiv: 1504.03812.

[KLSV10] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 83–93. Springer, May 2010.

[NDLS12] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search on Time-Dependent Road Networks. *Networks*, 59:240–251, 2012. Journal version of WEA'08.

[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.

[Tuk77] John W. Tukey. *Exploratory Data Analysis*. Pearson, 1 edition, January 1977. Published: Paperback.

Appendix

A. Experiment Plots

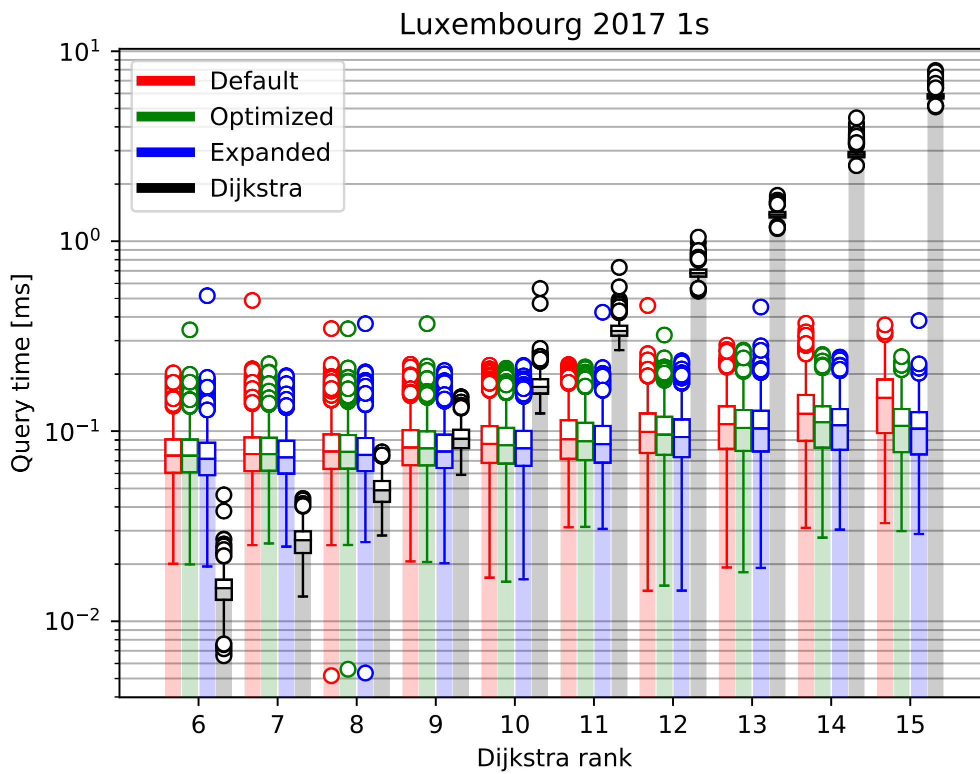


Figure A.1.: Query times on the Luxembourg 2017 graph with 1s resolution.

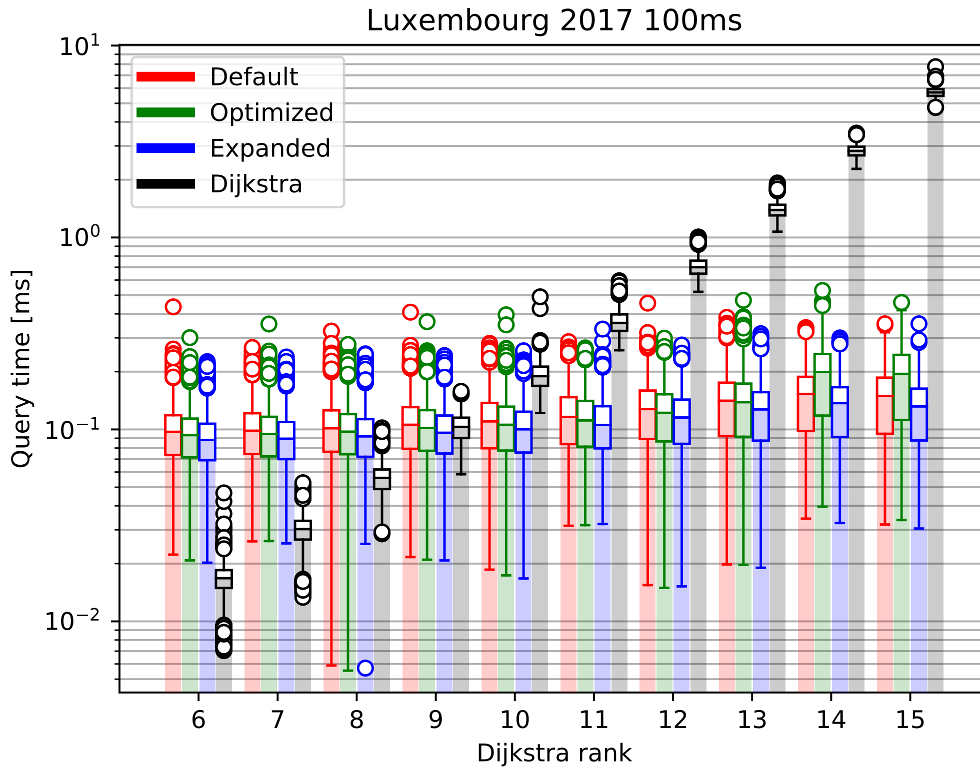


Figure A.2.: Query times on the Luxembourg 2017 graph with 100ms resolution.

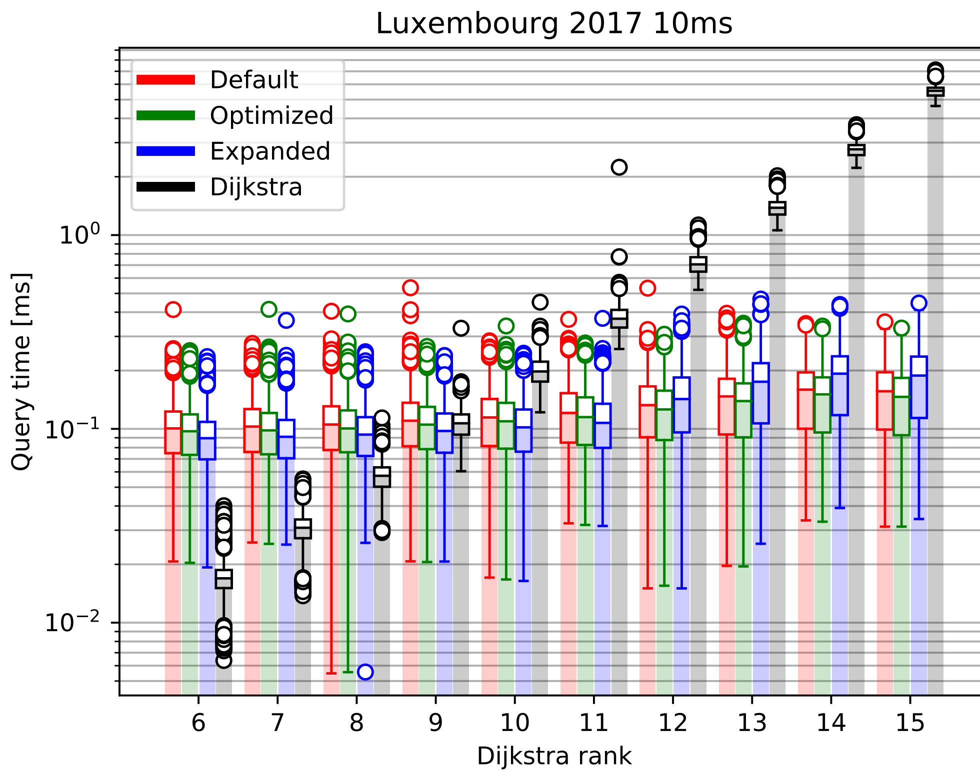


Figure A.3.: Query times on the Luxembourg 2017 graph with 10ms resolution.

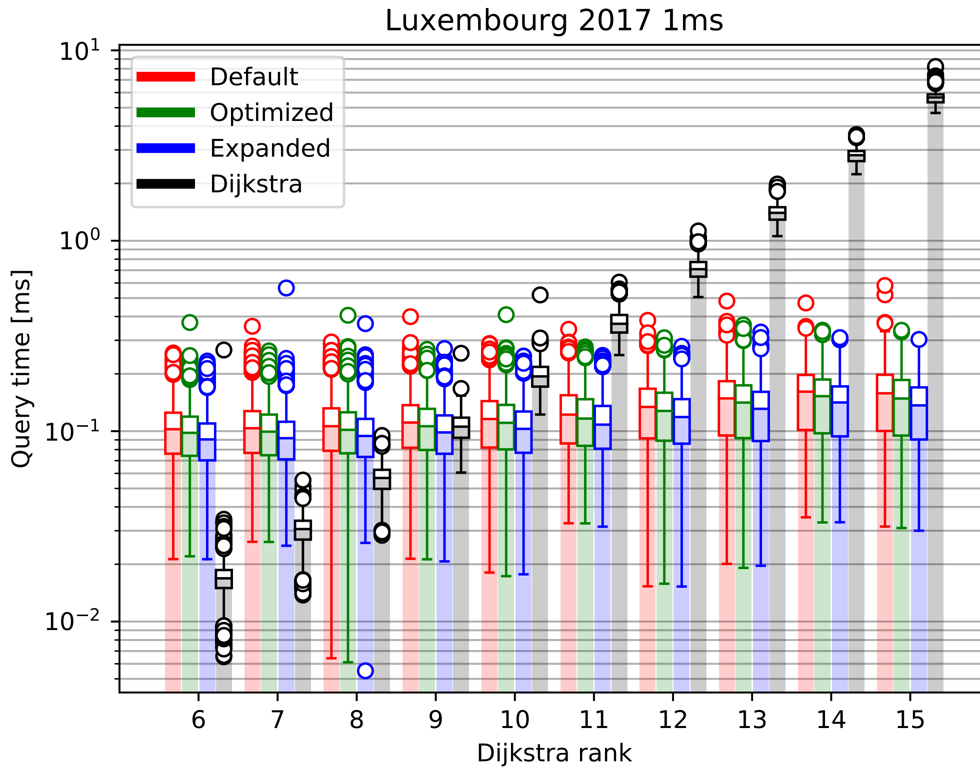


Figure A.4.: Query times on the Luxembourg 2017 graph with 1ms resolution.

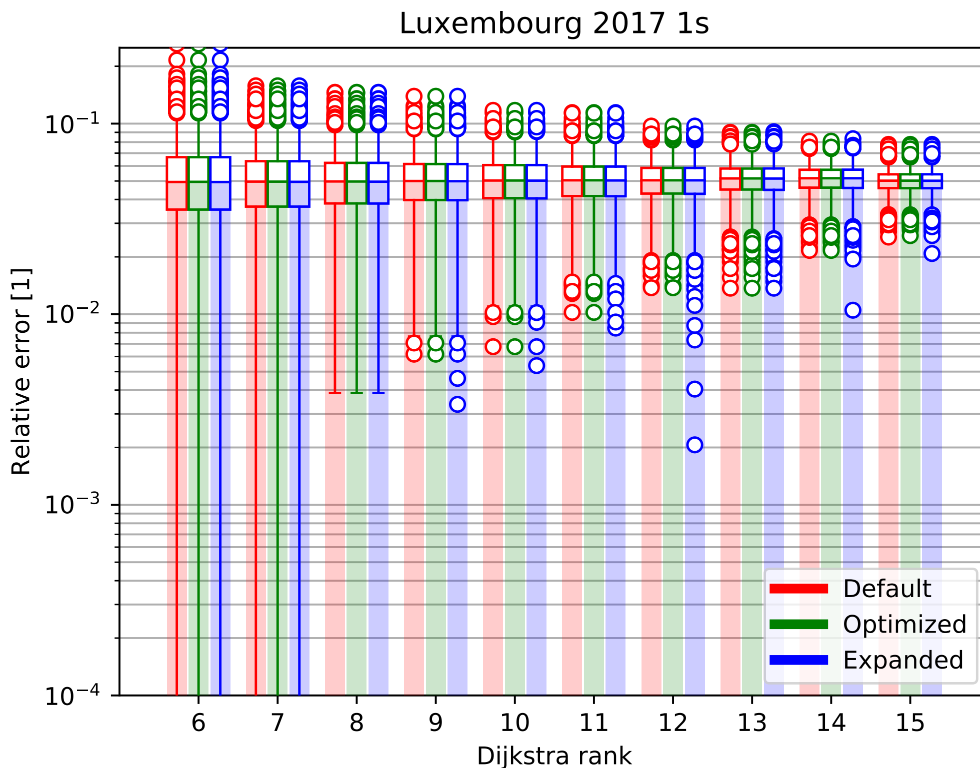


Figure A.5.: Query errors on the Luxembourg 2017 graph with 1s resolution.

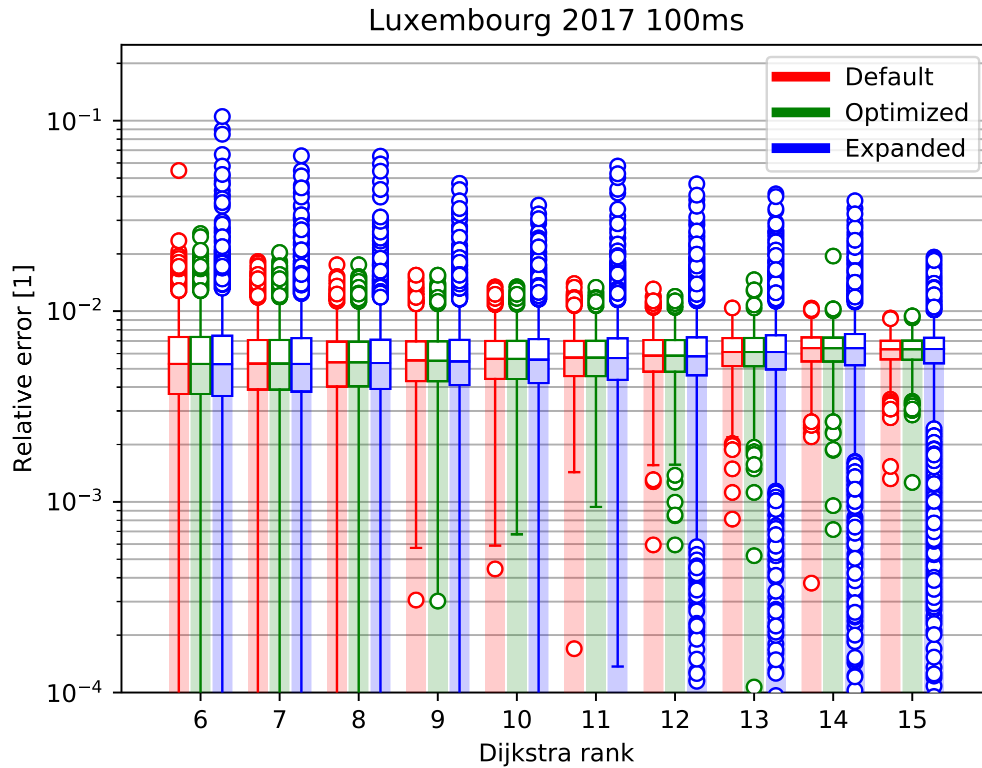


Figure A.6.: Query errors on the Luxembourg 2017 graph with 100ms resolution.

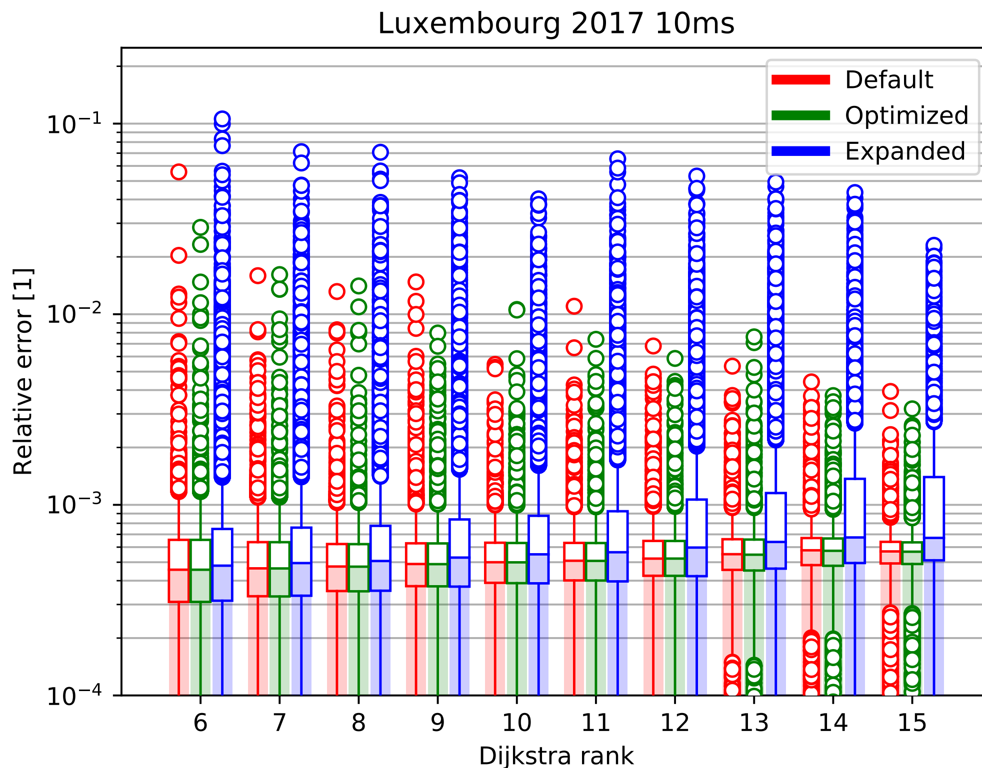


Figure A.7.: Query errors on the Luxembourg 2017 graph with 10ms resolution.

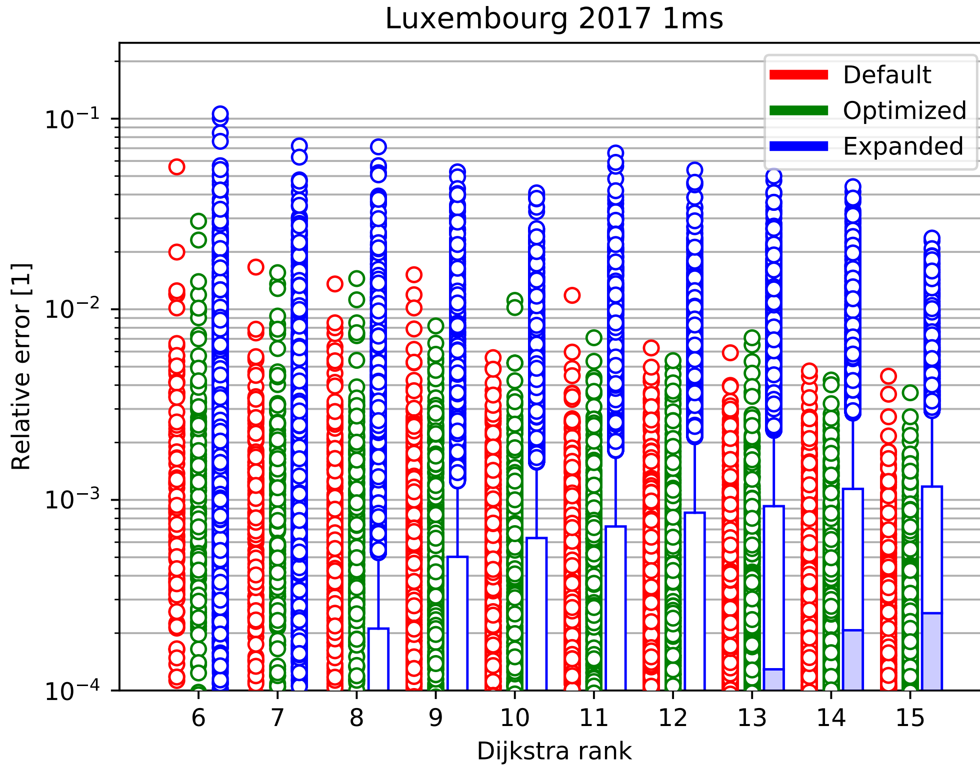


Figure A.8.: Query errors on the Luxembourg 2017 graph with 1ms resolution.

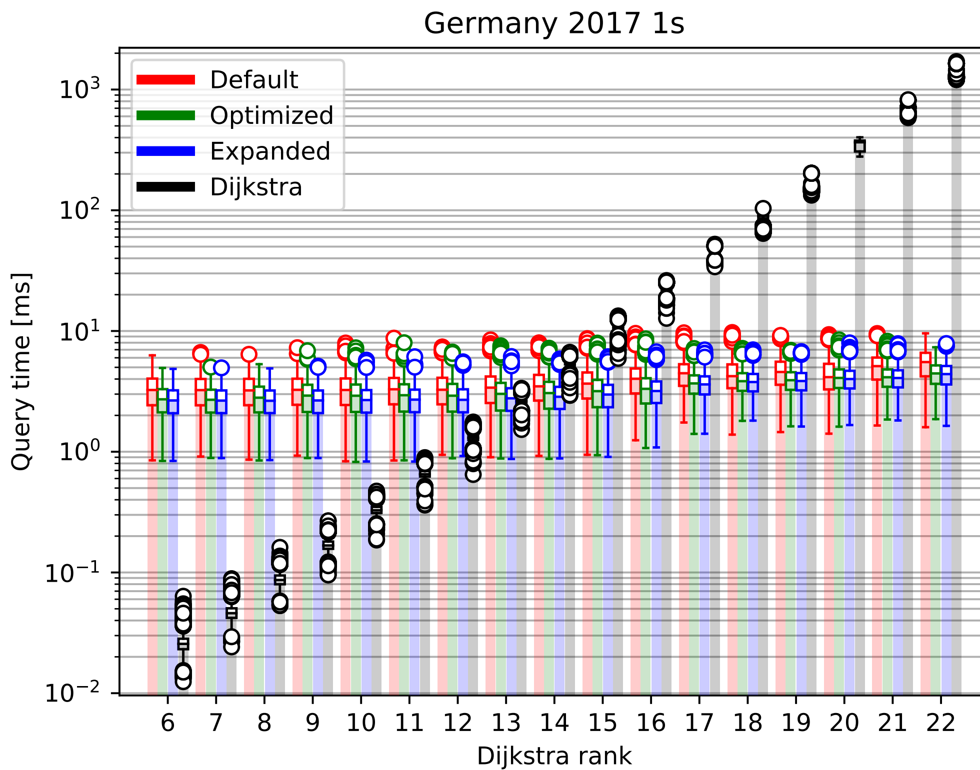


Figure A.9.: Query times on the Germany 2017 graph with 1s resolution.

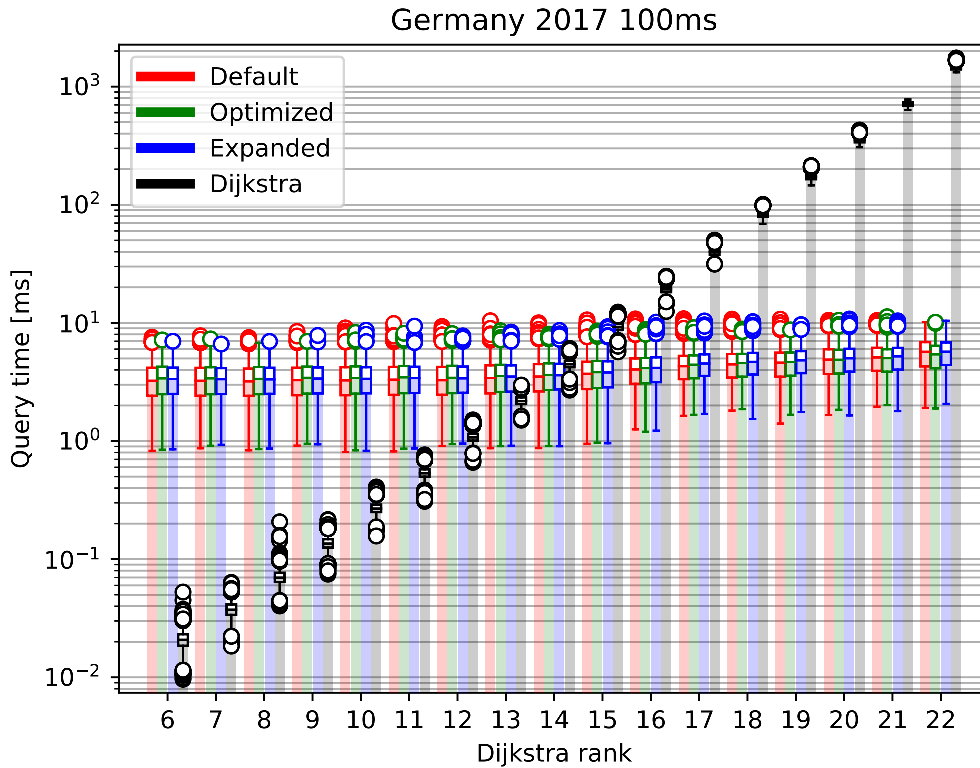


Figure A.10.: Query times on the Germany 2017 graph with 100ms resolution.

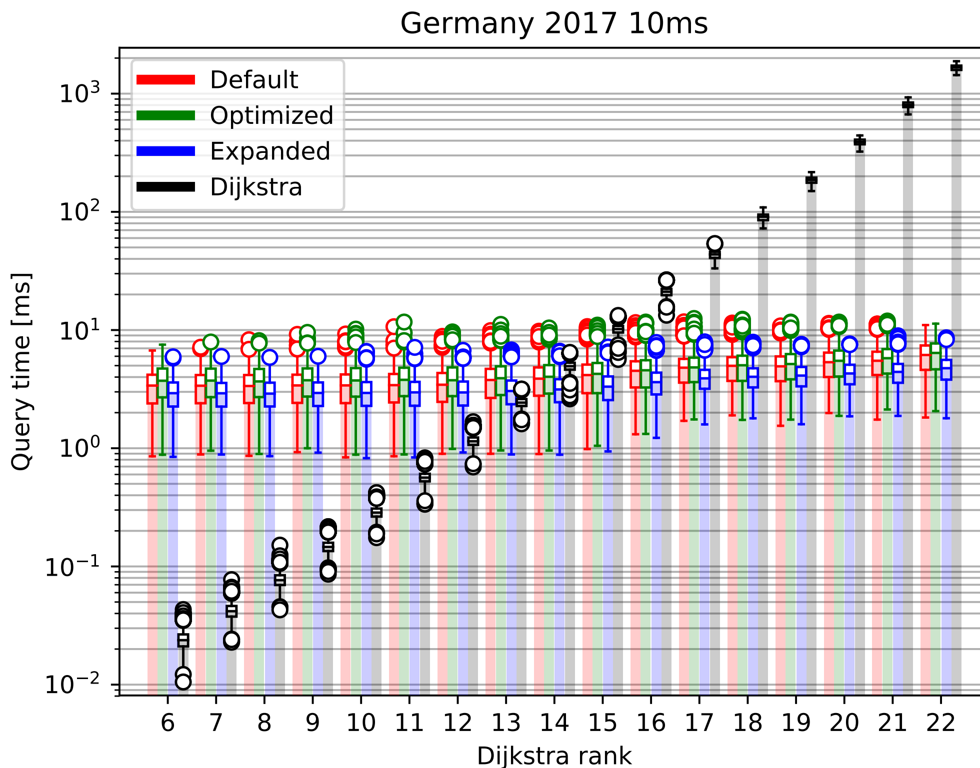


Figure A.11.: Query times on the Germany 2017 graph with 10ms resolution.

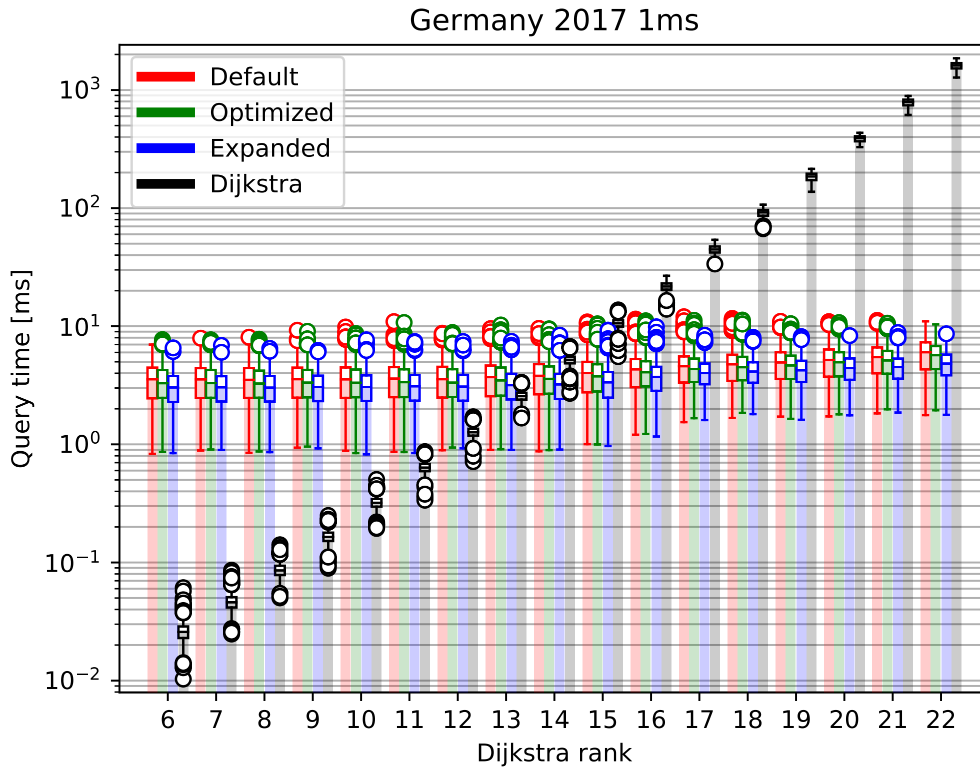


Figure A.12.: Query times on the Germany 2017 graph with 1ms resolution.

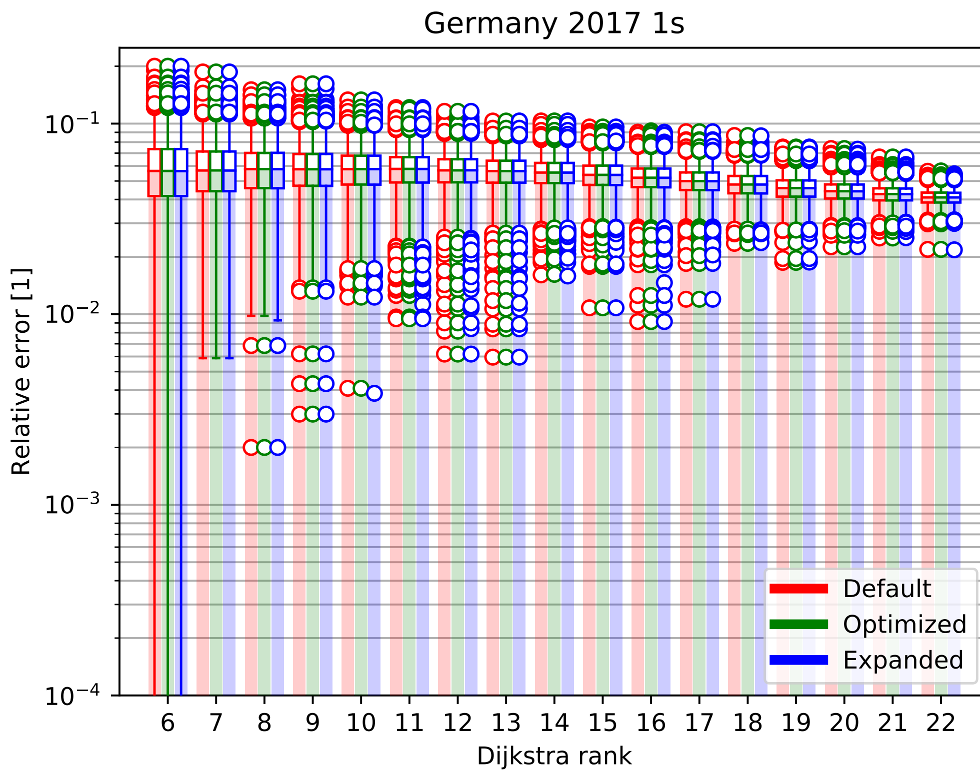


Figure A.13.: Query errors on the Germany 2017 graph with 1s resolution.

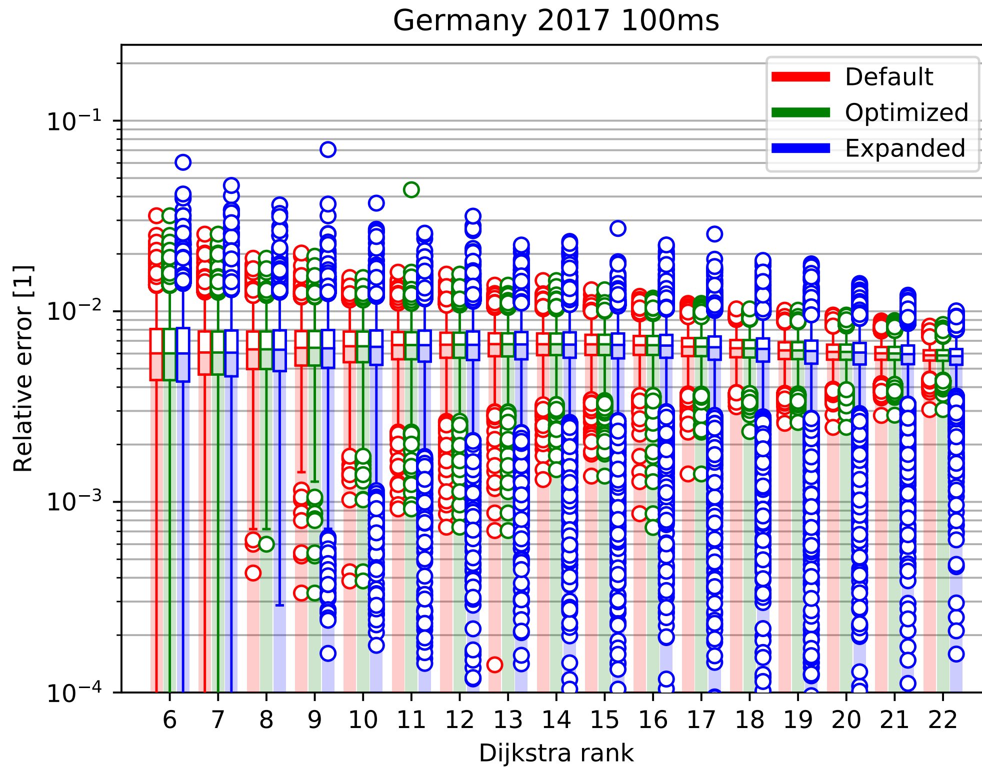


Figure A.14.: Query errors on the Germany 2017 graph with 100ms resolution.

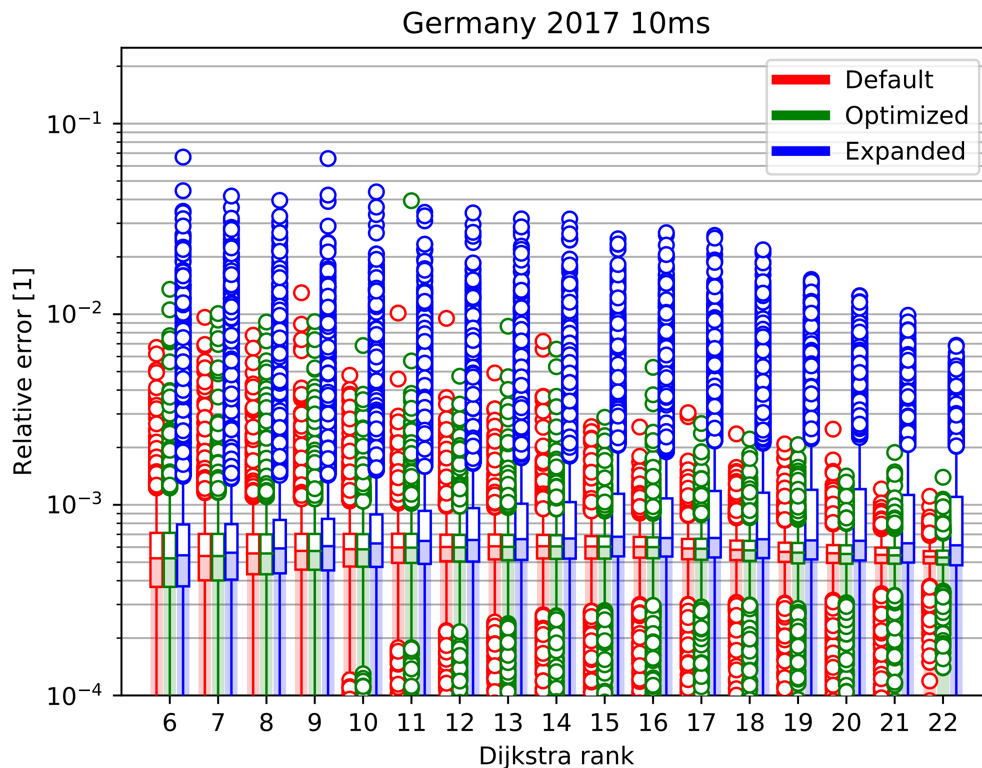


Figure A.15.: Query errors on the Germany 2017 graph with 10ms resolution.

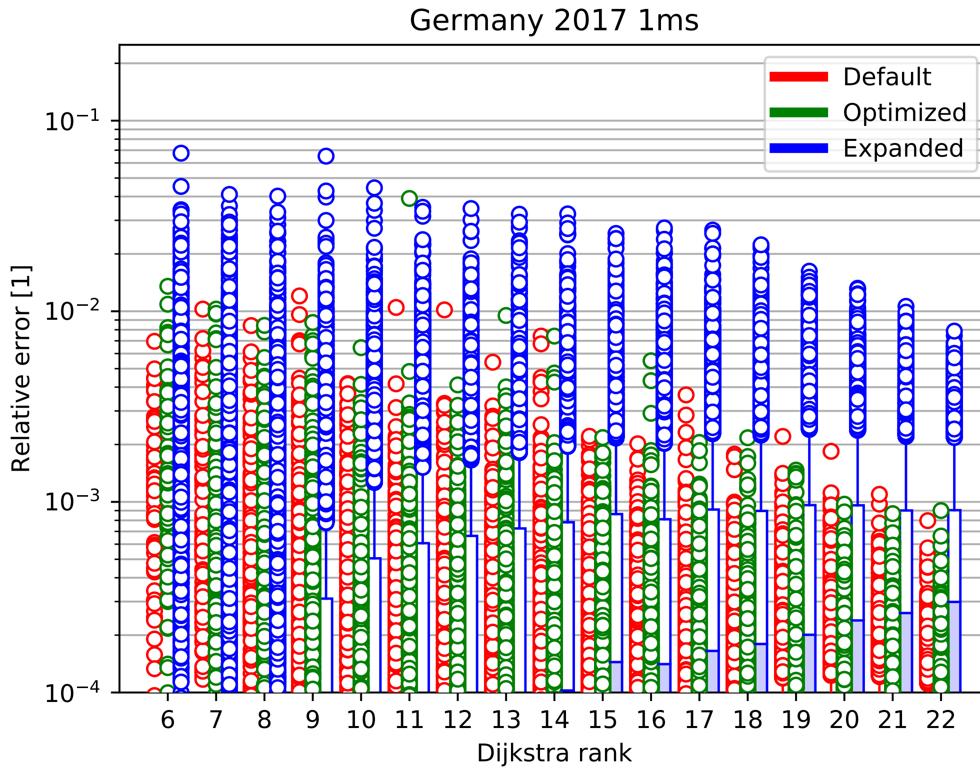


Figure A.16.: Query errors on the Germany 2017 graph with 1ms resolution.

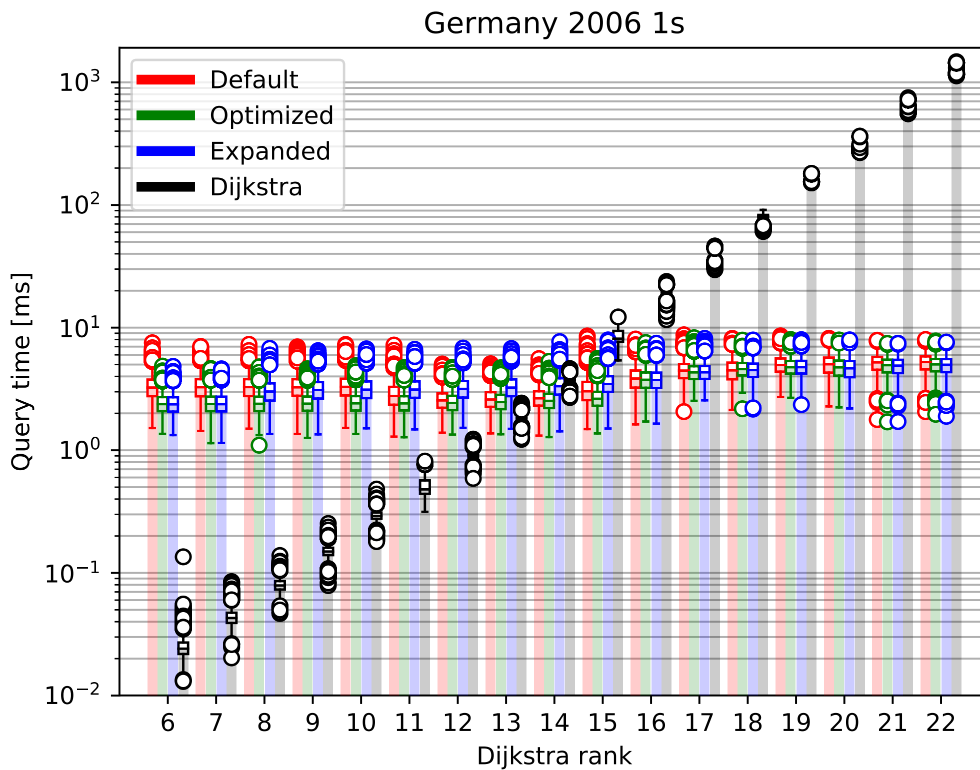


Figure A.17.: Query times on the Germany 2006 graph with 1s resolution.

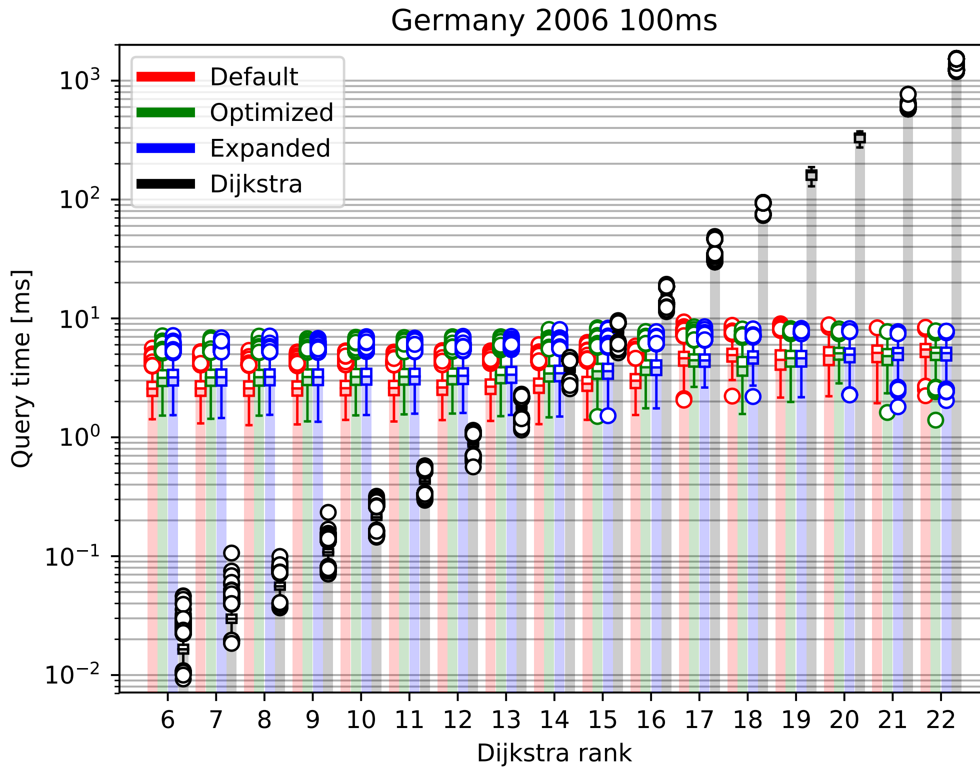


Figure A.18.: Query times on the Germany 2006 graph with 100ms resolution.

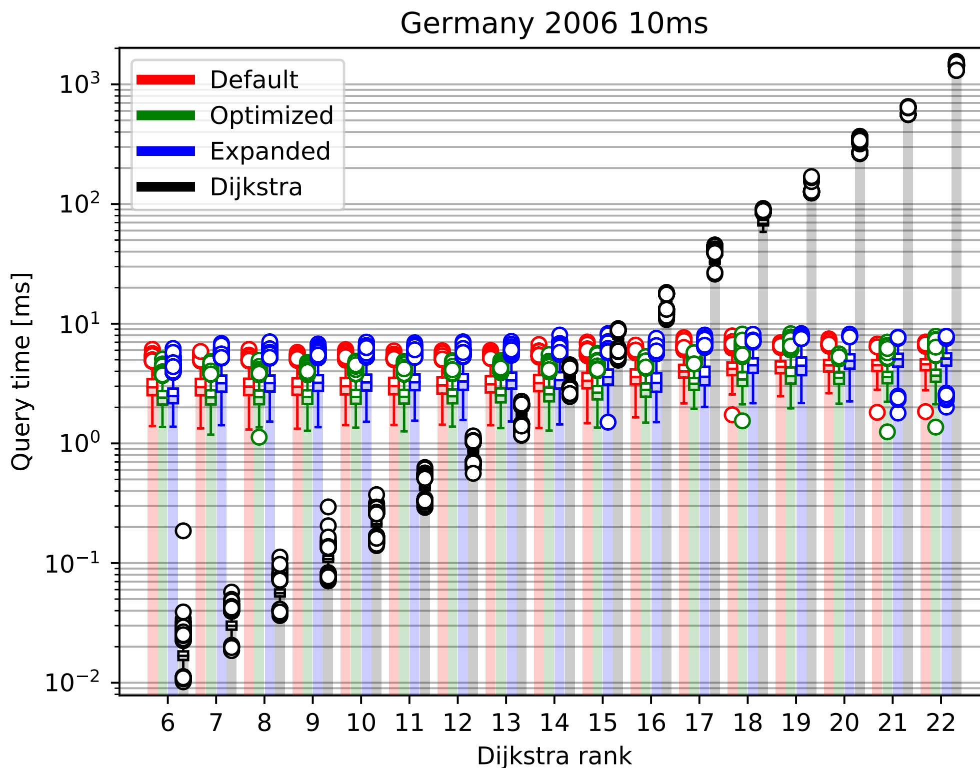


Figure A.19.: Query times on the Germany 2006 graph with 10ms resolution.

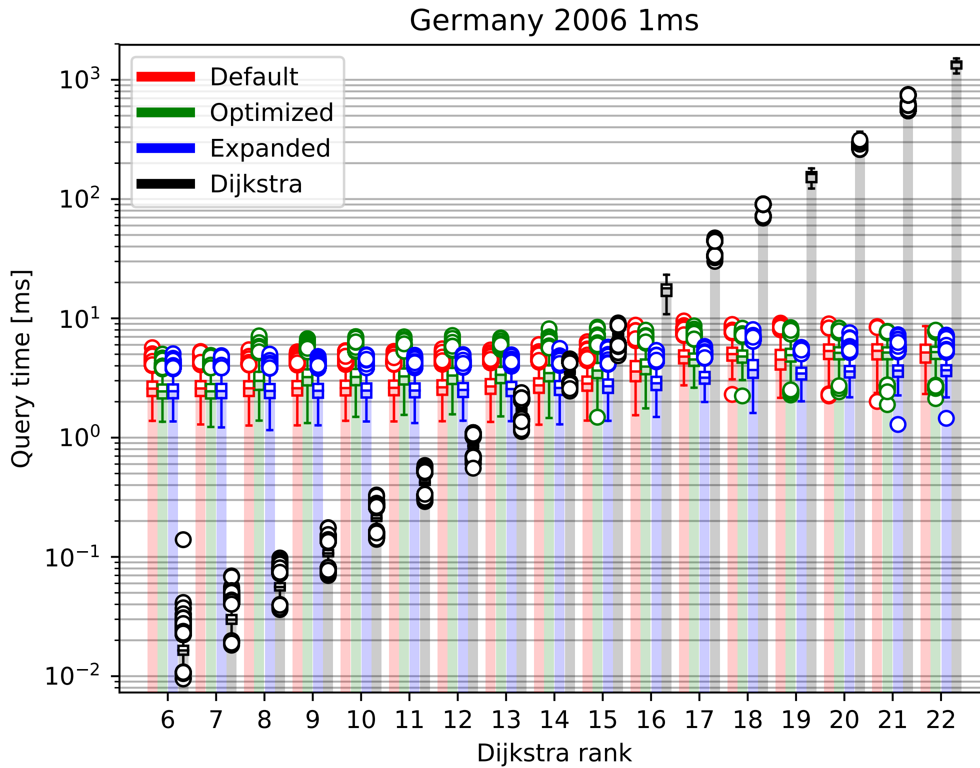


Figure A.20.: Query times on the Germany 2006 graph with 1ms resolution.

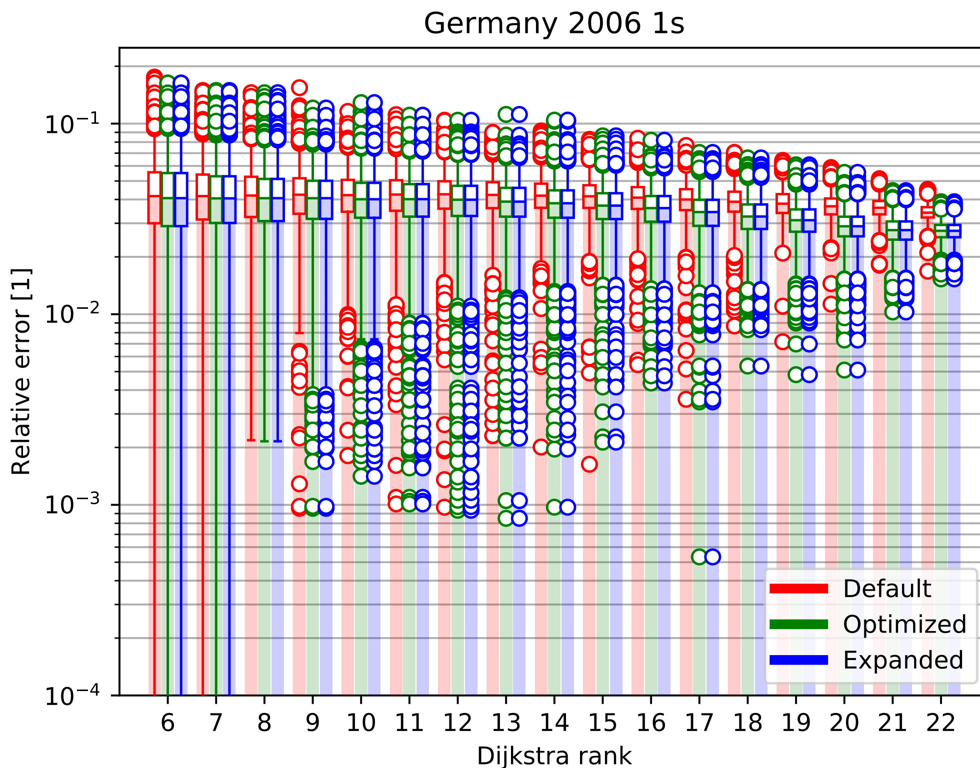


Figure A.21.: Query errors on the Germany 2006 graph with 1s resolution.

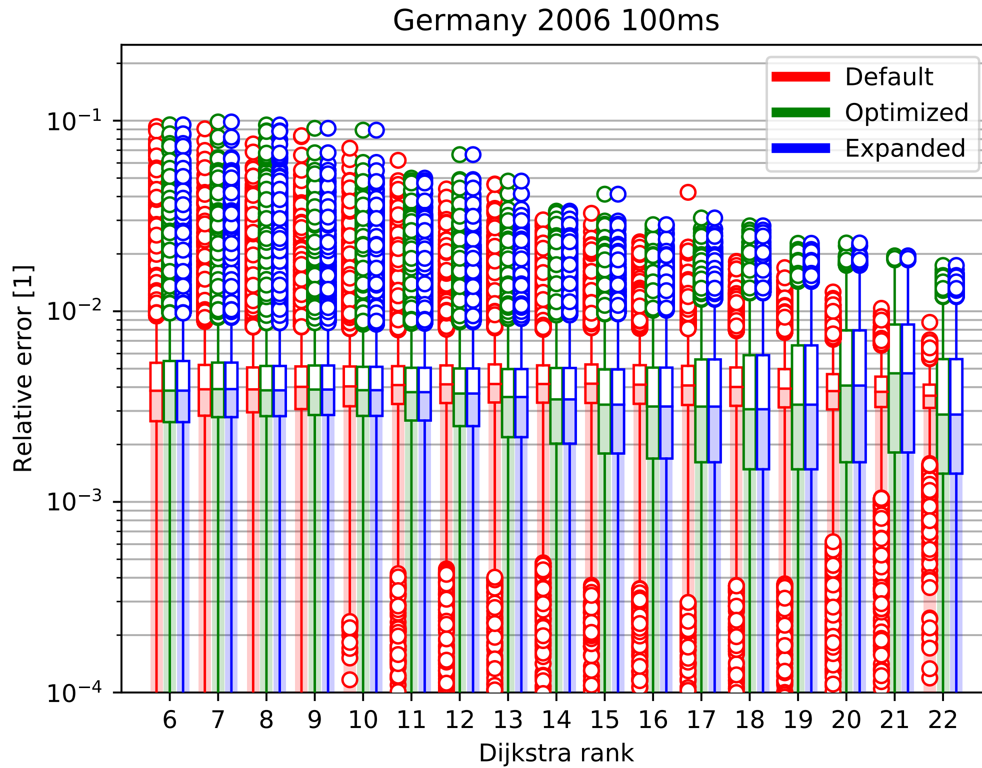


Figure A.22.: Query errors on the Germany 2006 graph with 100ms resolution.

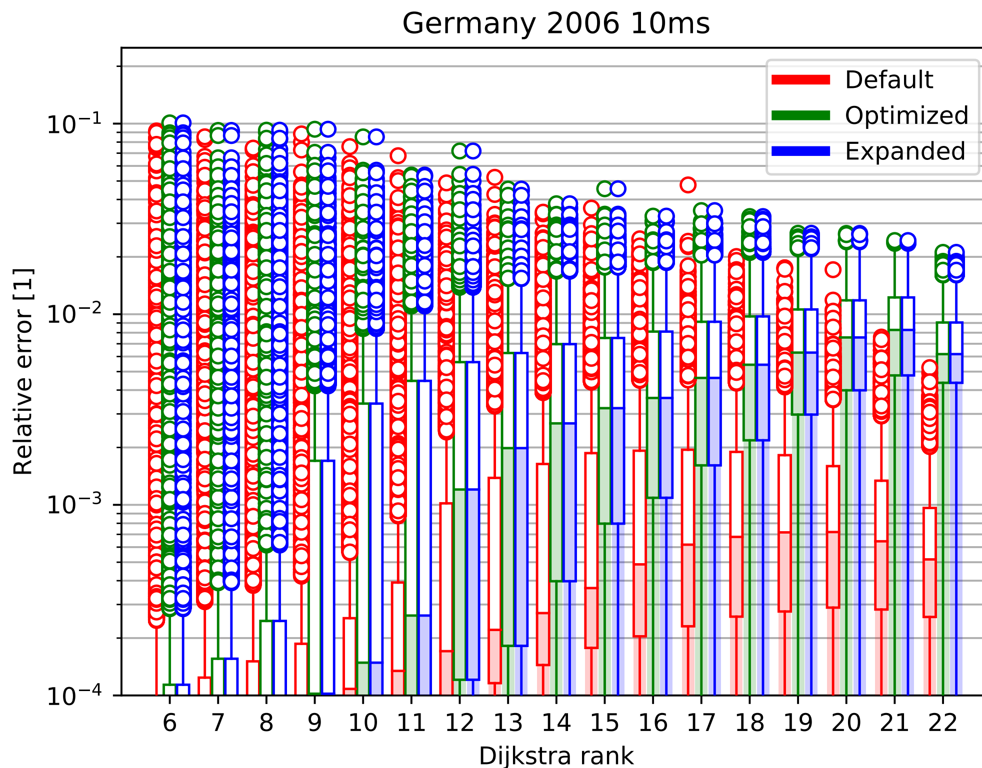


Figure A.23.: Query errors on the Germany 2006 graph with 10ms resolution.

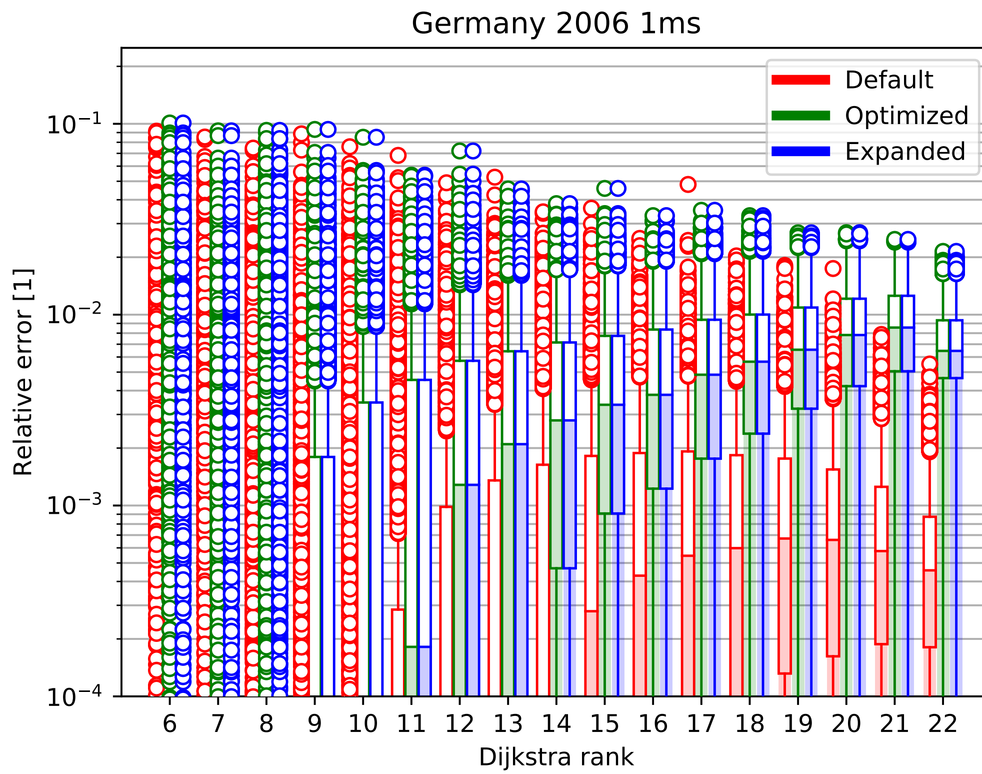


Figure A.24.: Query errors on the Germany 2006 graph with 1ms resolution.