Alternative Route Techniques

and their Applications to the Stochastic on-time Arrival Problem

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften /
Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Moritz Helge Kobitzsch

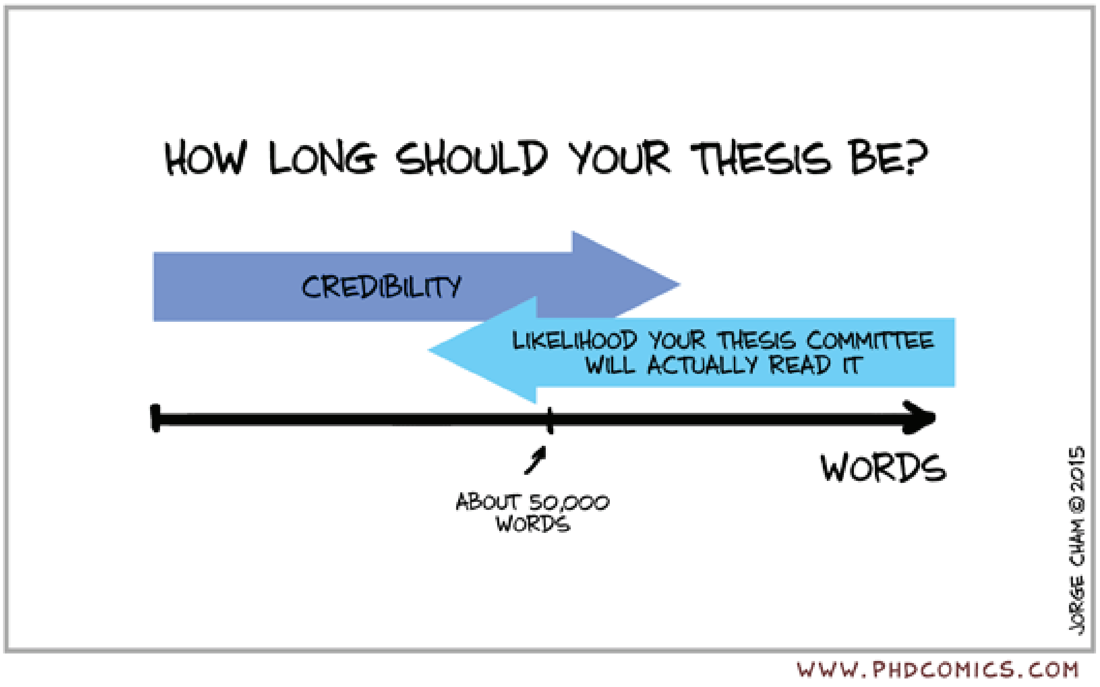
aus Berlin

Tag der mündlichen Prüfung: 06.11.2015

Erster Gutachter: Herr Prof. Dr. Peter Sanders

Zweiter Gutachter: Herr Andrew Goldberg, PhD

In memory of my sister.


HOW LONG SHOULD YOUR THESIS BE?
CREDIBILITY
LIKELIHOOD YOUR THESIS COMMITTEE WILL ACTUALLY READ IT
WORDS
ABOUT 50,000 WORDS
JORGE CHAM © 2015
WWW.PHDCOMICS.COM

"Piled Higher and Deeper" by Jorge Cham (`www.phdcomics.com`)

This thesis is based on 80 914 lines of code[1]. According to `SLOCCount`, this puts the coding work alone at an estimated twenty-one years of development. The credibility (s.a.) is backed by 90 358 words, distributed over 7 448 lines of LaTex code.

[1]Data generated using David A. Wheeler's 'SLOCCount'

Ω Acknowledgements

This thesis presents not only my own work. Many of the topics discussed in this work are strongly influenced by many different people, without whom this thesis would have not been possible. There is no way of putting a clear number on the relevance of their input; the order in which they are listed is purely coincidental and should not reflect any meaning.

First, I would like to thank all of my co-authors. I owe Samitha Samaranayake, for introducing me to the stochastic on-time arrival problem and all the fruitful discussions we had. Dennis Schieferdecker, your input on the different topics we worked on together as well as your support during my own endeavours are greatly appreciated. Dennis Luxen I have to thank for his countless contributions to our work on route corridors. Not only for the route corridors, but also for a great experience during my short stay at Microsoft Research, I owe gratitude to Daniel Delling and Renato Werneck who both taught me a lot. Thank you for all the help you have given me and your support, not only in my professional but also in my personal life.

In addition, I have to thank all of my coworkers and office mates that made life at the institute as good as it could be. I specifically am grateful to Sebastian Schlag for all the fruitful discussions over coffee, to Timo Bingmann for his support in system maintenance and his efforts in the office social events. I thank Christian Schulz for supplying his partitioner and trying out adjustments to better support my own algorithms and Jochen Speck for his general support in managing our compute servers.

Furthermore, I have to give thanks to my students that helped a great deal in accomplishing the results presented in my thesis. Both Marcel Radermacher and Stephan Erb have engaged me in most interesting discussions and have provided a great deal of input. Also, Valentin Buchhold helped a great deal by implementing a lot of useful tools and algorithms for me.

Last, but not least, I have to thank my friends and family for all the moral support over the duration of this work. I especially owe gratitude to my mother for all her support in not only proofreading my publications but also this thesis. Sina Schmitt and Vera Döttling, thank you for all your intense moral support during hard times.

Contents

Chapter 1

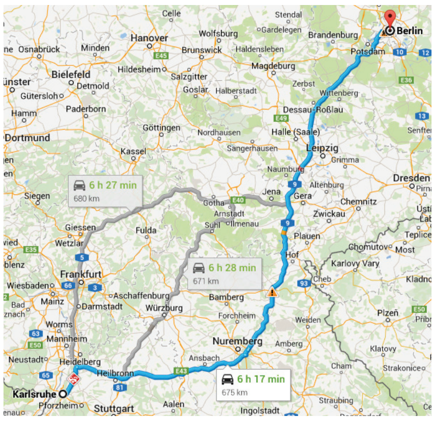
1 Introduction

A motivational view on the contents of this thesis and a navigational guide.

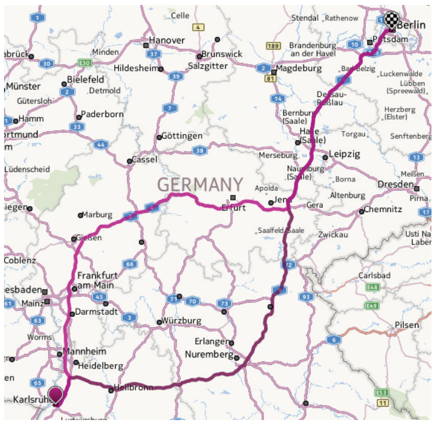
The former US president Franklin D. Roosevelt once said *"There are as many opinions as there are experts."* In the context of route planning, this saying translates into a multitude of *optimal* paths between any two given locations, influenced by personal preferences of different *experts*. While modern algorithms resulted in a wide availability of services that help us to navigate our daily life, these personal preferences can shed a negative light on the quality of the offered service. A simple method of testing a service would be to ask it for its suggestion for a known route. If the service does not offer any suggestion that resembles one's own preferences, one would most likely deem it unfit for personal use. Multiple services – e.g. Google Maps or Nokia Here, see Figure 1.1 – try to alleviate this problem by offering a set of alternative routes to best cover different preferences.

In the research community, however, the problem of finding good alternative routes has seen far less attention than the classic shortest path problem. A long range of publications, a result of the shortest-path horse race that sprung the 9th DIMACS implementation challenge introduced a plethora of *speed-up techniques*; techniques that accelerate shortest path computations in a specific setting. We describe the results of the horse race in Chapter 3. Most of these techniques, however, rely on a unique representation of distance between any given locations. Given this distance, they operate by pruning as much of the road network from a search as possible. This process discards anything that does not describe an optimal path. As a result, these techniques complicate the process of finding good alternative routes. Each of the possible routes may only be *optimal* in a setting described by the personal preferences of a group of users. Offering all possible settings in a single speed-up technique, however, seems complicated.

In this thesis, we take a close look at a wide range of different techniques for the computation of alternative routes on the two most popular speed-up techniques



(a) *Google Maps*



(b) *Nokia Here*

Figure 1.1: *Example of alternative routes in mapping services for a route between Karlsruhe and Berlin.*

currently in use. From the standpoint of an algorithm engineer, we explore how to exploit the different techniques for their full potential.

1.1 Context

This thesis covers techniques for route planning, with a focus on alternative routes, from the view of an algorithm engineer. At this point, we would like to present a general context as well as summarize the contributions presented throughout this work. In general, the main goal of this thesis is to describe techniques for an interactive setting. Such a setting usually allows for a turnaround time between a hundred and two hundred milliseconds.

1.1.1 Algorithm Engineering in Route Planning

Information technology, by now, influences many aspects of our daily life. This influence is made possible by algorithms and data structures that form the heart of every computer application. *Algorithm theory* describes simple models of problems and machines and forms a base for algorithmic research. Typically, algorithm theory is concerned with describing worst case bounds or provable asymptotic guarantees. In the context of route planning, however, no completely satisfying guarantees have been found, yet. Nevertheless, a plethora of different techniques have been developed by the community. These techniques work surprisingly well and manage to handle road networks at the scale of whole continents; offering query times of less than a millisecond. Many of these techniques, as well as the work we present in this thesis, originate from a concept that tries to overcome one problem present in the strictly theoretical point

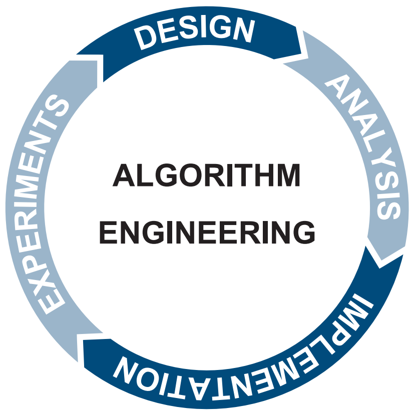


Figure 1.2: *Algorithm Engineering Cycle*

of view. To fully define this concept, known as algorithm engineering, is far beyond the scope of this thesis and we refer to [San09] for a more complete definition. In the context of this work, however, we would like to shortly summarize the general process. Simplified, algorithm engineering can be viewed as a cyclic process (see Figure 1.2) that revisits every step based on deductions originating from the previous step. The implementation and the experimental analysis augment the pure theoretical view of algorithm design and analysis.

In algorithm engineering, we focus strongly on actual inputs rather than considering all possible input possibilities; especially, as many of them might never be relevant. In shortest-path calculations for example, arbitrary *graphs*[1] are far from the reality that we have to handle. In road networks, the combination from an embedding in the plan and the presence of speed limitations severely restricts possible connections and introduces a natural hierarchy; this hierarchy ranges from arterial roads to highways.

The success of these techniques aroused the interest of algorithm engineers in the theoretical background as well. This interest gas led to new discoveries that try to consolidate the practical results theoretically. We discuss these theoretical results in Chapter 3.

1.2 Contribution and Thesis Outline

This work is based on a multitude of contributions that stem from the publications listed in Appendix 14. At this point, we present a brief summary of their content. We do not cover the introductory Chapters (2 – 5) and the discussion Chapter (12) in detail at this point, but focus on the distinct contributions.

[1]see Chapter 2 for a definition

Parallel Computation of Pareto Optimal Routes – Chapter 7. Pareto optimal routing is probably the most direct approach to consider multiple criteria. Instead of only considering travel time, multiple possible metrics can be handled directly at the same time. Any of the possible combinations is considered a part of the solution space if it is not dominated by another solution; in this setting, a solution that is better in every aspect dominates another solution. We present the first practical implementation of a parallel algorithm that computes Pareto optimal routes.

Penalty Method – Chapter 8. The penalty method is a previously introduced iterative process to compute alternative routes. Speaking on a high level, it repeatedly computes shortest paths in an ever changing model of the road network by applying some form of penalization to the found shortest path. The highly dynamic method, however, does not go well with classic techniques for fast computation of shortest paths that usually rely on a static network. Adapting a recently rediscovered method, we present the first implementation of the penalty method that is suited for interactive scenarios, offering query times close to or below 100 ms.

Hierarchy Decomposition for Alternative Routes – Chapter 9. The most common approach to the computation of alternative routes on road networks is the so-called *via-node* approach. The technique relies on specifying an intermediate destination to compute a multi-hop path from the starting point, through the intermediate destination and ending at the final destination. Finding those intermediate destinations is a difficult problem on its own. Testing a potential candidate for its quality only adds to the cost. We present a new approach that manages to handle both of these problems in an elegant fashion and manages to compute alternative routes of a higher quality than previously presented via-node approaches.

Alternative Routes via Segment Exclusion – Chapter 10. Both previous methods have a significant drawback. Choosing an appropriate penalization that does not affect wrong regions of the path seems difficult. The same can be said for finding a reasonable intermediate destination. To alleviate these difficulties, we explore an intermediate solution: instead of penalizing a full path, we only consider a small set (e.g. a short segment) of intermediate locations on the path and exclude it from further considerations. As such, the approach can somewhat be imagined as the dual approach to the via-vertex paradigm. Our previously unconsidered approach offers a simple concept, fast implementation and surprisingly good results.

Route Corridors – Chapter 11. As a final concept, we introduce route corridors; a concept that has been developed to handle a hybrid routing scenario that connects a mobile device to a server. Originating from the idea of limited connectivity on a device with low compute power, corridors were designed to ensure a reliable service. The concept is based on the idea to provide the mobile device with just enough information to be able to handle some limited amount of human error. The resulting corridors provide viable means to extract alternative routes from a severely reduced graph, thus enabling usually very expensive techniques at a low cost.

An Application to the Stochastic on-time Arrival Problem – Chapter 13. Most techniques in classic route planning assume a static travel time for the traversal of every road segment. This is, at best, a very optimistic view on reality. One approach to achieve a more realistic model is to assume the travel time to follow some probability distribution and to consider the reliability of different paths. This is done in the stochastic on-time arrival problem. These considerations are very complex, though, and require rather high computational effort to find solutions. A lot of this effort is wasted on parts of the network that now, in the more difficult model, cannot be discarded without losing guaranteed correctness. In an experimental study, we observe in how far alternative routes can be used to approximate this setting. We follow the notion that we need only consider important subsets of the road network when traveling sufficiently long, even though we cannot guarantee the restriction to be sound.

Part I

An Introduction to Speed-up Techniques and Alternative Routes

Chapter 2

2 Preliminaries

Beneath every building – hidden from the eye – lies a solid foundation. This thesis, as well, requires this initial building block in the form of basic concepts and notations, provided in this chapter.

So far, we have kept to a very abstract level of detail. For the following formal discussions, we need to introduce our terminology and the basic notations. Both are additionally summarized in the glossaries.

2.1 On Road Network Representations

In practice, navigation is concerned with roads and the connecting intersections. Before we can go on and describe our algorithms, we have to describe how the road network maps to the terminology used in theoretical computer science. The main concept used in shortest path computations is a graph. A graph is a collection of elements, usually named vertices, and a set of arcs that describe some form of relation between these vertices. We provide Definition 2.1.

Definition 2.1 (Vertex, Arc, (reversed) Graph)**.** *The tuple* $G =: (V, A \subseteq V \times V)$ *defines a graph* G*, consisting of a finite set of vertices* V *and a finite set of arcs* A*. Vertices are numbered from* 0 *to* $|V| - 1$*, with* $|V|$ *describing the number of vertices. An arc is an ordered pair* (u, v) *of vertices,* $u, v \in V$*; we call* u *the origin and* v *the destination of the arc. A graph is considered undirected if* $\forall a = (u, v) \in A : (v, u) \in A$*, else it is considered directed. Any function* $\mu : A \mapsto \mathbb{R}$ *is called a metric, or cost function. If a metric is defined on the arcs, the graph is called weighted, else unweighted. Drawing vertices as dots and arcs as connections between them, a graph is called planar if an embedding into a 2D-surface exists without any two arcs crossing each other.*

The reversed *graph to* $G\,(V, A)$ *can be obtained by replacing any arc* (u, v) *with the arc* (v, u) *and setting* $\mu\,(v, u)$ *to* $\mu\,(u, v)$*).*

Typical examples of a metric are the travel-time or travel-distance. Within the context of graphs, we have to introduce the concept of overlay graphs.

Definition 2.2 (Overlay Graph)**.** *An overlay graph G^+ to a graph $G(V, A)$ is defined on a subset $V^+ \subseteq V$ of vertices. In contrast to a subgraph of G, though, an overlay graph may contain any arc of the set $A^+ \subseteq V^+ \times V^+$. The arcs are chosen in such a way that they characterize the graph G, e.g. in a weighted graph, we choose the metric on the arcs in A^+ in such a way that shortest path distances within G^+ are the same as in G.*

In the context of these graphs, our discussion focuses on the following set of measures that further describe a graph.

Definition 2.3 (Graph Measures)**.** *We specify the number of vertices by $|V| =: n$ and the number of arcs as $|A| =: m$. We name $\delta(v \in V) = \delta_{out}(v) := |(v, \cdot) \in A|$ the (out-) degree of a vertex v. For directed graphs, we additionally define $\delta_{in}(v) := |(\cdot, v) \in A|$, the in-degree of v.*

A graph is considered sparse if $\delta(\cdot) \leq c$ for some constant value or asymptotically speaking if $m \in \mathcal{O}(n)$ [1].

On is own, this definition seems rather indistinct. It will be clearer the moment we finish describing how graphs relate to navigation in road networks, though. In general we distinguish between two different representations:

Vertex-Based Representation. The *vertex-based* representation is the most straight forward representation when you consider a classic map. Intersections form the vertices and arcs represent the road segments in between. If you draw the intersections according to their geographic locating and draw a connecting line for every arc, the representation would –at least in some sense – resemble a map in a travel guidebook.

Modeling restrictions for specific turns or even integrating penalties for specific turns does not integrate naturally into this representation. One possibility to integrate turns is the arc-based representation.

Arc-Based Representation. The arc-based representation [Win02] of a road networks considers the road segments as basic elements (vertices) of the graph rather than the intersections. An arc in the graph represents the traversal of an intersection and the following road segment. This modification essentially replicates a vertex from the vertex-based representation multiple times, each representing the space right before one would enter the intersection itself. Formally speaking, given a graph $G(V, A)$, and a function $\mathcal{T} : A \times A \mapsto \mathbb{R}$, assigning a turn cost to every traversal of an intersection and ∞ to forbidden turns, the arc-based representation $G'\left(V', A'\right)$ of $G(V, A)$ is defined by:

[1] We refer to [MS08] for a definition of the Landau/asymptotic notation.

$V' := A$ and $A' = \{(a_i, a_j) : a_i, a_j \in A \wedge \mathcal{T}(a_i, a_j) < \infty\}$. In case of a weighted graph, the metric to $G'\left(V', A'\right)$ on $(a_i, a_j) \in A'$ is defined as $\mu'(a_i, a_j) := \mu(a_j) + \mathcal{T}(a_i, a_j)$.

In this thesis we use the vertex-based representation, unless otherwise stated.

This fundamental description translates into other relevant concepts like paths and distances within a road network:

Definition 2.4 (Path, Shortest Path, Length, Distance)**.** *Given a weighted graph $G(V, A)$, a metric μ, and two vertices $s, t \in V$: A path (P,$P_{s,t}$) is a sequence of connected arcs $a_i = (v_i, v_{i+1})$, with $v_i \in V$. The path is called simple if $\forall i \neq j : v_i \neq v_j$. Its length ($\mathcal{L}$) is defined as $\sum_i \mu(a_i)$. If the length of P is minimal over all possible paths, P is called a shortest path and we refer to it by Π $(\Pi_{s,t})$. Additionally, we label $\mathcal{L}(\Pi_{s,t}) := \mathcal{D}(s,t)$, the distance between s and t. The length of the longest shortest path is called the diameter (D) of G. We additionally define the concatenation of paths for two paths $P_{s,v}$ and $P_{v,t}$ as $P_{s,v} \cdot P_{v,t} := P_{s,v,t}$, the path that firsts visits all vertices of $P_{s,v}$ and follows $P_{v,t}$ after reaching v.*

2.2 Data Structures

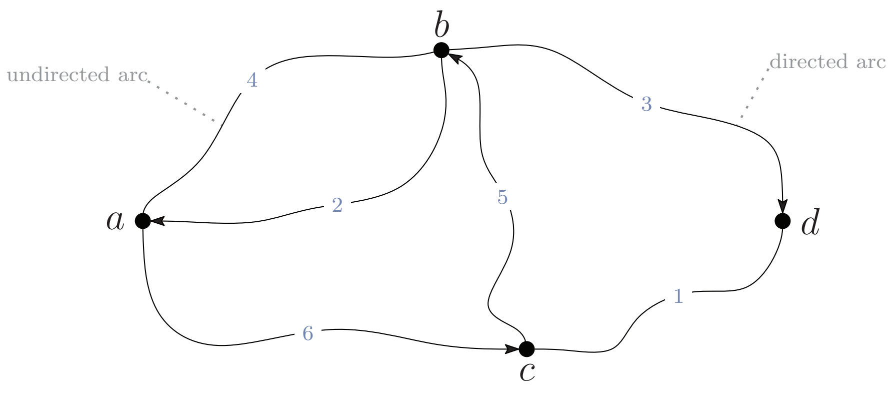
Throughout this thesis, we make use of some basic data structures and data representations. These will be described in the upcoming section.

2.2.1 Graph Representation

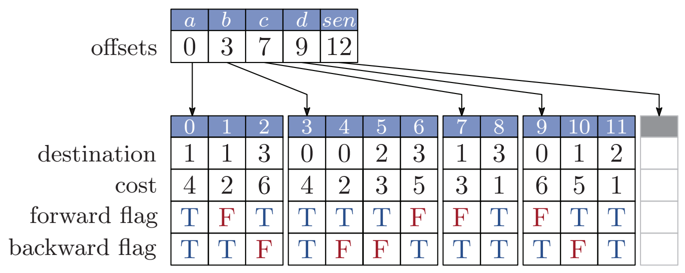
We represent our graph data structure as a sorted array that contains all arcs. The arcs are sorted by their origin and left unordered with respect to other sorting criteria. This order enables us to describe the full set of arcs that originate at a vertex with a single additional array. Each entry in this additional augmenting array – we call it an *offset array* – represents a vertex and holds an offset into the arc-array. The offset for each vertex points to the first arc that belongs to it. The list of arcs for a given vertex ends at the first arc of the next vertex. To avoid special cases, we keep an additional sentinel entry in the offset array that points to the end of the list of arcs.

Since access to arc is based on its origin, we do not store it explicitly with the arcs. The array representation makes the structure efficient for scanning all arcs of a given vertex; modifications of the graph, however, are expensive.

Figure 2.1 presents a small example graph as well as its adjacency array representation. This type of graph representation is known by different names. [MS08] describes it as adjacency array, but it is also known as the *forward star* representation. We were unable to determine the exact origin of the representation and direct to [DP84] for a list of early algorithms utilizing it.



(a) *Graph*



	a	b	c	d	sen
offsets	0	3	7	9	12

	0	1	2	3	4	5	6	7	8	9	10	11	
destination	1	1	3	0	0	2	3	1	3	0	1	2	
cost	4	2	6	4	2	3	5	3	1	6	5	1	
forward flag	T	F	T	T	T	T	F	F	T	F	T	T	
backward flag	T	T	F	T	F	F	T	T	T	T	F	T	

(b) *Adjacency Array*

Figure 2.1: *Illustration of the adjacency array graph representation as implemented for most of our algorithms. If all arcs share the same flags, the flags are omitted in the representation.*

2.2.2 Priority Queue

A priority queue is an abstract concept of a data structure, fundamental to all the algorithms presented throughout this thesis. It presents an ordered collection of elements, ordered according to a priority function. As this thesis is concerned with shortest path computations, we chose tentative shortest path distances as priority; as a direct result, we assume low values to be of higher priority. Next to trivial interfaces, a priority queue provides the following functionality:

`insert:`	insert an element into the queue
`extractMin:`	extract the minimal element from the queue, according to the assigned priorities

Addressable Priority Queues allow for an important additional operation:

`decreaseKey`: decrease the key of a stored element, moving it closer to the front of the queue.

The abstract concept has been implemented in various ways, usually called *heaps*. At this point, we cover three different variants used within this thesis. All three do not support addressing elements in a natural way and we have to augment them with additional functionality.

Binary Heap. Originally used in sorting [Wil64], the binary heap forms a simple yet elegant data structure. It is used in many of the techniques we discuss starting in Chapter 3. A binary heap is represented as a balanced binary tree, often times serialized to an array. The tree is designed to fulfill the so-called *heap property*: a property that ensures each parent to be of higher priority than its children. Inspecting the element of highest priority can be done in $\mathcal{O}(1)$. Inserting an element into a binary heap is implemented by appending it to the array at the last position and moving the item up the tree until the heap property is fulfilled. As the tree is fully balanced, this requires at most $\mathcal{O}(\log n)$ comparisons. A similar procedure is performed for the removal of an element: as the minimal element is at the top of the tree, the hole has to be filled. In a binary heap, we take the element at the end of the array and move it to the top of the tree, temporarily violating the heap property. We restore the property by moving the element down the tree, swapping is with the child-element of higher priority. This process is continued until the heap property is restored; the hight of the tree guarantees a cost of $\mathcal{O}(\log n)$ for this operation. Next to a potential overhead of the addressing part (possible in $\mathcal{O}(1)$), decreasing the key of an element is essentially identical to insertion, both in asymptotic running time as in its implementation.

Due to its simplicity, the binary heap is the most often used priority queue data structure. In some cases, it is generalized to the k-ary heap by increasing the branching factor of the heap from 2 to k. The binary heap is the standard heap used in this thesis.

Bucket Heap. If we consider keys to be nonnegative integers, which most often is the case when it comes to route planning, binary heaps offer some optimization potential. Denardo and Fox [DF79] present an alternative approach to the concept of a priority queue:

For any algorithm that operates on monotonously increasing integer labels in the priority queue, Denardo and Fox propose to utilize a set of buckets: The bucket heap of Denardo and Fox does not distinguish between elements with equal labels. Elements that are assigned the same label are kept in a single bucket and can be operated on in an arbitrary order (or even asynchronously and in parallel). The moment a bucket runs empty, the next non-empty bucket is localized by linearly scanning over all buckets. Often, the range that can contain non-empty buckets is limited. For example, in a static graph, the maximal cost of an arc is limited by an cost bound $\Delta = \max_{a \in A} \mu(a)$. An algorithm that only offsets labels by the cost of arcs therefore has to scan at most

Δ buckets to find the next non-empty bucket if such a bucket exists. For all further algorithms, this property will be given and `extractMin` operates in $\mathcal{O}(\Delta)$. Inserting a vertex into the priority queue can be done in $\mathcal{O}(1)$, as we only have to append it to the respective bucket. The same holds for `decreaseKey`, as we can directly move the vertex to its final position.

In the presence of a Δ limiting the maximal offset, we can implement the list of buckets as a cyclic structure and avoid the reoccurring allocation of new buckets. Whenever a bucket runs empty, the pointer to the bucket of elements with minimal assigned label is moved and the bucket is reused for the next maximal possible label.

The bucket heap is used in the code of Chapter 9.

A drawback of this approach is that Δ can become quite large and buckets can be sparsely populated. A similar approach that handles this situation in a way better suited to most of our purposes is given in the form of radix heaps.

Most of our techniques require only a few accesses to the heaps. If more accesses should be required, a radix heap [AMOT90, CGS97] could offer a way of increasing performance.

In the following sections, we discuss some fundamental algorithms for graph traversal. The history of this problem, reaching back into the late eighteen-hundreds.

2.2.3 Graph Traversal

Path finding itself is one of the classical graph problems. In the context of this thesis, we also make use of additional graph traversal algorithms not directly related to shortest paths on weighted graphs. The graph traversal algorithm relevant to the content of this thesis is depth-first search (DFS). Precursors of DFS reach back into the 18th century to the French engineer Charles Pierre Trémaux (unpublished) and have been picked up by Tarry [Tar95] and Lucas [Luc94]. The Trémaux method describes an algorithm to navigate a mace. DFS follows basic ideas of this method and operates as follows. Using $\mathcal{O}(n)$ additional space to mark vertices as visited, DFS follows an arbitrary arc originating at the current vertex after marking it as visited. When the algorithm encounters a previously visited vertex, it backtracks to the previous vertex and another arc is chosen. We present an implementation in pseudocode for DFS-traversal – which can be done in $\mathcal{O}(n+m)$ – in Algorithm 2.1.

Usually, the DFS algorithm is not executed to explicitly generate an order but rather to execute some task directly when encountering a vertex. Multiple schemes are possible to execute a task or for ordering the vertices. The most popular ones are preorder and postorder. Algorithm 2.1 shows a preorder numeration in which the order is assigned when first visiting a vertex. It is also possible to assign a number to a vertex when last leaving it. The latter numbering would generate a postorder numeration of the vertices. For further readings on DFS, we refer to [MS08].

We use the DFS algorithm to extract reasonable sub-graphs and during our preprocessing to extract Strongly Connected Components(SCCs); the algorithm for SCCs

Algorithm 2.1 Depth First Search

Input: A weighted graph $G(V, A)$ and a source (s) vertex
Output: An order the vertices were visited in

```
procedure REC_DFS($order\,[], v, next$)
    if $order\,[v] \neq \infty$ then                    ▷ $v$ has been visited before
        return $next$
    else
        $order\,[v] = next{+}{+}$                        ▷ Assign order to unvisited vertex
        for all $a := (v, w) \in A$ do
            $next = \texttt{REC\_DFS}(order, w, next)$   ▷ Recursively traverse neighbors
        end for
        return $next$
    end if
end procedure

$order\,[] = \{\infty, \ldots, \infty\}$                 ▷ Initialize order
$order\,[s] = 0$

$\texttt{REC\_DFS}(order, s, 0)$                         ▷ Recursively calculate the order

return $order[]$
```

loops over all vertices and performs an iteration of the DFS algorithm on previously unvisited vertices.

2.3 Shortest Paths

The problem of finding shortest paths through a network is one of the fundamental problems in algorithm engineering. In the following section, we give a formal definition of the problem and introduce the main variations relevant to the contents of this thesis.

2.3.1 Problem Definition

Already around 1956, Bellman, Ford, and Moore describe a routing problem [Bel58, For56, Moo59], and offer a solution based on dynamic programming. A bit later, in 1959, Edsgar W. Dijkstra [Dij59] described two problems in the connection with graphs, one of which is similar to the routing problem presented by Bellman. Together, Bellman and Dijkstra describe the central problem we cover extensively in this thesis: the routing problem (Bellman), or *Problem 2* (Dijkstra), the task to find the path of minimum length between two given vertices, known by now as the Single Source Shortest Path Problem (SPP). Formally, we define it as follows:

Definition 2.5 (Single Source Shortest Path Problem (SPP))**.** *Given a weighted graph $G(V, A)$, with assigned metric μ, mapping $a \in A \mapsto \mathbb{R}$ as well as two vertices $s, t \in V$, called source and target: Calculate the shortest path $\Pi_{s,t}$ between s and t.*

This fundamental problem, which is well defined for graphs that do not contain negative cycles, is studied as plain version but also in a lot of variations, some of which we shortly describe next.

2.3.2 SSSP Variations

The variations relevant to our work focus on considerations of different metrics. Here, we shortly describe three versions of the SPP related to this thesis.

Pareto-optimal Shortest Path. Pareto-optimal shortest path operates on a multi-criteria metric. A simple example would be the simultaneous optimization of both traveled distance and travel time. The goal is, instead of a single shortest path, to find all non-dominated solutions to the SPP. Formally, dominance for Pareto-optimal solutions is defined in the following way:

Definition 2.6 (Pareto-Dominance in Shortest Paths)**.** *Given a weighted $G(V, A)$, a metric $\mu : A \mapsto \mathbb{R}^d$ as well as two paths P_a, P_b between two vertices s and t of respective lengths $\mathcal{L}_{\mathbf{a}}$ and $\mathcal{L}_{\mathbf{b}}$: Assuming metrics are better for smaller values, P_a dominates P_b if $\forall_{i<d} : \mathcal{L}_{\mathbf{a}}[i] \leq \mathcal{L}_{\mathbf{b}}[i]$.*

This augmentation of the metric-space into higher dimensions is a natural approach that, however, has significant impact on the SPP for multiple criteria: Garey and Johnson [GJ79] show that even for only two simultaneous criteria, the calculation of shortest paths becomes $\mathcal{NP}$-hard; nevertheless, the problem remains solvable for many relevant instances [MHW01].

Time-Dependent Shortest Path. In contrast to Pareto-optimal shortest path, the time-dependent shortest path problem, for the most part, considers only a single metric to be optimized. This metric, however, is not considered constant but instead varies over the course of a day. The metric, in this scenario, is usually referred to as travel time function (TTF). Orda and Rom describe how to handle quite general TTFs [OR91], especially TTFs that violate the FIFO property, a property that essentially describes a scenario in which waiting at a given position is never beneficial. Time-dependent shortest path aims at incorporating information that is reoccurring regularly; such restrictions could be speed-limits that are valid at specific times of the day – e.g. due to noise limitations – or the morning and evening rush-hour. For a detailed view of time-dependent routing, we refer to [Bat14].

Stochastic Shortest Path. The final shortest path model this thesis is concerned with is the presence of unreliable travel times that cannot be predicted and therefore cannot be distilled into TTFs. Unreliable disturbances cannot be distilled into a time-dependent model that predicts travel times in a reliable way. A different way of dealing with such unreliable variations is to model the travel time as a probability distribution. Various studies that discuss distributions to best match measured travel times are covered in Chapter 13. In the presence of probability distributions for a metric, two variations of the SPP are considered. The first variation is to calculate the path of least expected travel time for time-varying probability distributions[2] [MHM00], the other variation considers a full strategy that aims at optimizing the chance for on-time arrival under the constraints of a given budget [SBB12b]. We study the latter concept in Chapter 13 as a use-case of alternative routes. At that point, we define the problem more closely and discuss related work.

2.4 Algorithms

The history of shortest path algorithms is difficult to trace; Schrijver gives some valuable insights into their development [Sch10] in a discussion that outlines the following sections. In classic algorithms, before the algorithm engineering community entered into the SPP horse race[3], two variations of the labelling algorithm describe the state of the art for the SPP. At this point, we give a short introduction to the basic concepts of these algorithms before we go on to describing more recent results on shortest paths.

2.4.1 The Labelling Algorithm

The labelling algorithm, due to Ford [For56], is a mostly theoretical construct in the context of this work and the most simplistic algorithm for computing shortest paths that we discuss. Nevertheless, it describes the general concept of labels and provides a simple basis for discussion.

The labelling algorithm follows a very intuitive concept. At first, we assign a label of ∞ to every vertex and a label of 0 to the source of the search. Now, as long as such a vertex exists, the algorithm selects an arc that allows to *improve* another label by relaxing it; the process of relaxation extends a path by an additional arc, creating potentially improved labels for the new endpoint of the extended path. As long as no cycle of negative length exists that is reachable from the source, the algorithm converges to optimal distance labels for every vertex. The ability to pick any arc that results in an improvement of another label, however, can result in long running times

[2]without the varying time component, the problem can be reduced to a static setting by evaluating each probability distribution for its expected value

[3]see Chapter 3

[Edm70]. Slight modifications of this simple algorithm allow for guaranteed asymptotic running times in specific settings:

2.4.2 Dijkstra's Algorithm

In his publication on the SPP, Dijkstra introduced an algorithm [Dij59] that, for a long time, has been the fastest algorithm for a wide range of shortest path settings and still is the foundation for the highly optimized techniques we describe in Chapter 3. Even though, initially Leyzorek et al. [LGJ+57] described a rudimentary version of the same algorithm, today it is mostly known as Dijkstra's method. For the algorithm to work, the graph cannot contain arcs of negative length ($\mu(a) \geq 0, \forall a \in A$). Dijkstra's algorithm operates in rounds and keeps three sets of vertices. The first set describes the vertices that already were assigned a final distance label. These vertices are referred to as settled. The second set contains all vertices that have been assigned a tentative distance label. These vertices are named reached and form the set that the algorithm operates on. The final set is considered unreached and contains all vertices with ∞ as distance label. During the execution, the algorithm keeps all reached vertices in a priority queue. Initially, this priority queue consists only of the source vertex that was assigned the tentative label of zero.

As we depict in Algorithm 2.2, in each iteration, the vertex with minimal distance label is extracted from the queue and, if possible, the adjacent arcs are relaxed. Tentative distances are updated (`decreaseKey`, compare Section 2.2.2) and formerly unreached vertices are inserted into the priority queue with a priority of their now less than infinite tentative distance. The vertex that has been removed from the queue is considered settled. During the whole process, one can keep an array of parent pointers; each of these pointers names the vertex that last improved the distance label for the associated vertex. This allows to extract a (reverse) representation of the shortest path by following the parent pointers, starting at the target, until you reach the source.

Due to the fact that the algorithm never touches a settled vertex again, it is called label-setting. The characteristic of a label-setting algorithm allows for halting the execution the moment the target is settled. In the presence of negative cost at some arcs, the algorithm cannot operate as a label-setting algorithm anymore.

The asymptotic running time of Dijkstra's algorithm depends on the actual priority queue implementation. Without specifying the priority queue, the algorithm runs in $\mathcal{O}\left(n \cdot (T_{insert} + T_{extractMin}) + m \cdot T_{decreaseKey}\right)$. The algorithm inserts and removes each vertex exactly once from the priority queue and each arc results in at most a single `decreaseKey` operation.

Technically, Dijkstra's algorithm solves an even more general problem than the SPP. The algorithm computes the paths and distances from the source to all vertices that are not farther away than the target. Executed without specifying a target, Dijkstra's algorithm computes the shortest paths to all vertices in the graph.

Even though it has been discovered in 1957, the general method still remains the

Algorithm 2.2 Dijkstra's Algorithm

Input: A weighted graph $G(V, A)$ with associated metric μ, a source (s) and a target (t) vertex

Output: The shortest path between s and t encoded as a sequence of pointers as well as its length

```
1:  $\mathcal{L}[] = \{\infty, \dots, \infty\}$                ▷ Initialize tentative distance
2:  $\mathcal{L}[s] = 0$
3:  $\mathcal{Q} = \{s, 0\}$                                  ▷ Initialize the priority queue
4:  $parent[] = \{\infty, \dots, \infty\}$
5:  while $\mathcal{Q}$.size() and not isSettled($t$) do
6:      $v = \mathcal{Q}.extractMin()$                         ▷ Settle $v$
7:      for all arcs $a := (v, w) \in A$ do
8:          if $\mathcal{L}[w] = \infty$ then                  ▷ Previously unreached vertex
9:              $\mathcal{L}[w] = \mathcal{L}[v] + \mu(a)$
10:             $\mathcal{Q}.push(w, \mathcal{L}(w))$          ▷ Set $w$ to reached
11:             $parent[w] = v$
12:         else
13:             if $\mathcal{L}[v] + \mu(a) < \mathcal{L}[w]$ then    ▷ Improved tentative distance
14:                 $\mathcal{L}[w] = \mathcal{L}[v] + \mu(a)$
15:                 $\mathcal{Q}.decreaseKey(w, \mathcal{L}(w))$
16:                 $parent[w] = v$
17:             end if
18:         end if
19:     end for
20: end while
21: return $\mathcal{L}[t], parent[]$     ▷ Path can be extracted by following the parent pointers
```

standard approach for shortest-path queries. The advanced techniques we discuss in Chapter 4 follow the same design during the search-phase.

An alternative to Dijkstra's algorithm is given in the form of the Bellman-Ford algorithm:

2.4.3 The Bellman-Ford Algorithm

The Bellman-Ford algorithm, sometimes also referred to as Bellman-Ford-Moore Algorithm [Bel58, For56, Moo59], is an algorithm that employs a dynamic programming paradigm, comparable to the ideas presented by Shimbel [Shi54], to compute shortest paths in the presence of negative cost. The algorithm operates in at most n rounds. The state in round i can be described as the distance between the source and each vertex using paths of at most i arcs. In every round, the algorithm extends these known paths by an additional arc. As such, the algorithm follows the labelling paradigm described earlier by checking labels for eligibility in a fixed order. We illustrate the most basic implementation of the process in Algorithm 2.3. As the pseudo-code shows, the algorithm operates in a fixed number of n rounds. After these rounds, a final check can be performed to detect negative cycles. Without negative cycles, all shortest paths are simple paths. As such, the longest, in terms of hops, shortest path cannot be longer than n vertices in length. As such, if the final loop over all arcs can find an additional improvement, the associated path contains at least one vertex twice which implies the presence of a negative cycle in the graph. Even though the algorithm is not used in queries on speed-up-techniques, it is a major component of the update mechanisms used in Chapter 8 and Chapter 10.

Algorithm 2.3 Bellman Ford Moore Algorithm

Input: A weighted graph $G(V, A)$ with associated metric μ, and a source (s) vertex
Output: The shortest path between s and all other vertices encoded as a tree of pointers as well as their length or $-\infty$ if a negative cycle exists

```
1:  $\mathcal{L}[] = \{\infty\}$                                   ▷ Initialize tentative distance
2:  $\mathcal{L}[s] = 0$
3:  $parent[] = \{\infty\}$
4:  for $i := 1$ to $n$ do
5:      for all $a = (u, v) \in A$ do
6:          if $\mathcal{L}[u] + \mu(a) < \mathcal{L}[v]$ then
7:              $\mathcal{L}[v] = \mathcal{L}[u] + \mu(a)$
8:              $parent[v] = u$
9:          end if
10:     end for
11: end for
12: for all $a = (u, v) \in A$ do                             ▷ Check for negative cycles
13:     if $\mathcal{L}[u] + \mu(a) < \mathcal{L}[v]$ then
14:         $\mathcal{L}[] = \{-\infty\}$
15:         $parent[] = \{\infty\}$
16:         return $\mathcal{L}[t], parent[]$
17:     end if
18: end for
19: return $\mathcal{L}[t], parent[]$     ▷ Path can be extracted by following the parent pointers
```

We employ the algorithm as one of the processing variants discussed in Chapter 8 and Chapter 10.

Floyd-Warshall. An algorithm that operates in a similar way to the Bellman-Ford Algorithm is the Floyd-Warshall algorithm. Conceptually very simple, it solves the all-to-all shortest path problem in $\mathcal{O}\left(n^3\right)$ by inductively computing all distances of shortest paths in iteration i that only have vertices labeled $l <= i$ as inner vertices; a precursor to this algorithm for graphs with unit-metric is due to Shimbel [Shi53]. Initially, all arcs can be considered shortest paths without inner vertices. The algorithm itself only operates in a triple nested loop without a requirement for additional data structures apart from a distance matrix. [Flo62].

Algorithm 2.4 Floyd Warshall

Input: A weighted graph $G\left(V, A\right)$ with associated metric μ
Output: A distance matrix for all shortest path distances between vertices of G

```
$d\left[u,v\right] = \infty, \forall u, v \in V$
$d\left[u,v\right] = \mu\left(u,v\right), \forall (u,v) \in A$
$d\left[u,u\right] = 0, \forall u \in V$
for $k := 1$ to $n$ do
    for $i := 1$ to $n$ do
        for $j := 1$ to $n$ do
            $d\left[i,j\right] = \min d\left[i,j\right], d\left[i,k\right] + d\left[k,j\right]$
        end for
    end for
end for
return $d$
```

The algorithm is used in Chapter 8 and Chapter 10 for fast updates in a modified and vectorized form.

2.5 Further Concepts and Notations

Some other concepts beside the ones related to shortest paths also influence this thesis. In the following, we discuss the most important ones.

Partition. Some of the techniques we describe starting in Chapter 3 are based on separators or partitions. We define these related subjects as follows:

Definition 2.7 (Separator)**.** *In graph theory, we distinguish between vertex-separators and arc-separators. A vertex-separators describes a set of vertices that, if removed from the graph together with their adjoint arcs, fragment the vertices into a specific number of disconnected components. In the same way, an arc-separator describes a set of arcs whose removal segregates the graph into multiple disjoint components.*

The definition of separators allows for the, in a for us relevant version, definition of partitions:

Definition 2.8 (Partition, Cell)**.** *A partition ($\mathcal{P}$) is a arc-separator induced fragmentation of a graph's vertices into separated components. We refer to the distinct components as cells ($\mathcal{C}$). As such, the union of all cells forms the full set of vertices. Any vertex adjacent to an arc whose destination vertex is located in a different cell than the vertex itself is called a border vertex.*

Similar to the SPP, finding good partitions has seen a renaissance with the increased work in algorithm engineering. To describe these techniques in a fitting manner would be far beyond the scope of this thesis. Schulz [Sch13], however, gives a great overview over the work related to the calculation of partitions and also describes one of the best algorithms currently available to calculate partition in general graphs. Further information can also be found in the survey article of Buluç et al. [BMS+13].

Multi-Level A less concrete concept that manifests itself in multiple aspects of this thesis is the concept of multi-level approaches. In general, a multi-level approach operates by applying the same concept recursively or iteratively to the solution(s) found by the first execution. For example, we can think of multi-level partitions in which each cell is used to induce a new sub-graph as an input for a second partitioning step. This approach of multi-level partitions – see Chapter 3 – has been utilized in multiple approaches to speed up algorithms for the SPP. The concept can also be utilized to lessen the required computational effort by heuristically shrinking the input in a meaningful way. For example, the partitioner described in [Sch13] shrinks its input graph by matching well connected vertices until the graph is small enough to employ costly algorithms. In a complex process of shrinking and expanding, this initial solution is used to construct a final partition over multiple levels in which refinement is used to optimize the previously constructed solution to find a solution for the expanded graph.

Chapter 3

3 Speeding up Dijkstra's Algorithm

Speeding up the computation of shortest paths is one of the most prominent subjects in the algorithm engineering community. At this point, we provide a brief history of the efforts to speed-up Dijkstra's algorithm.

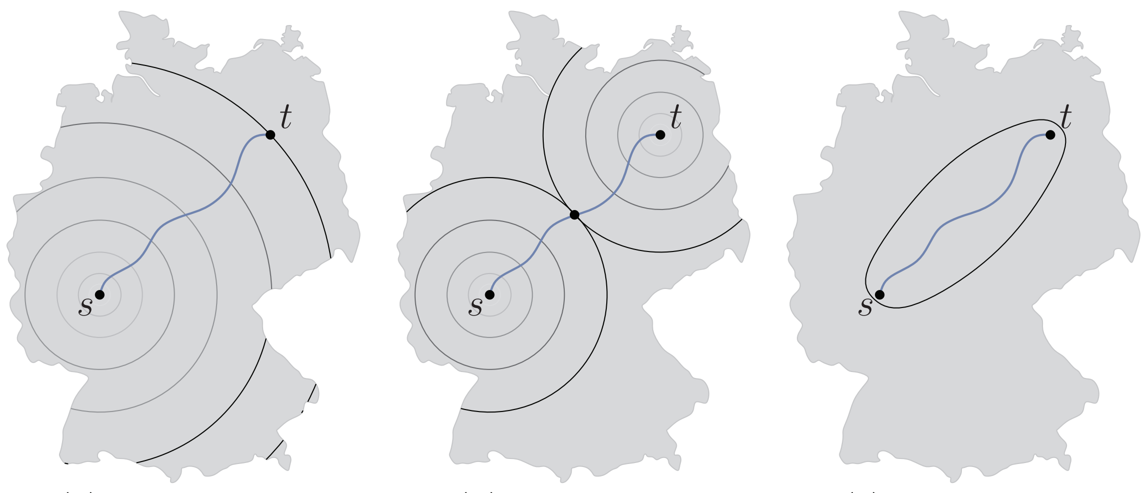
The SPP was initially solved by Dijkstra [Dij59] in 1959[1] from a worst case perspective and optimized by several others with better priority queues (e.g. [Tho04], bringing the asymptotic running time down to $\mathcal{O}\left(m + n \log\log\min\left\{n, \Delta\right\}\right)$. This near linear algorithm is still far beyond practicability for our desired scenario of less than a hundred milliseconds, if we consider large-scale use on continental-sized networks. On these large-scale instances, running times of Dijkstra's algorithm can reach into the order of seconds, classifying it as unfit for interactive use. As a result, the task of finding shortest paths in road networks has become a topic of interest in the algorithm engineering community.

One example is the 9th installment[2] of the DIMACS challenge. The Center for Discrete Mathematics & Theoretical Computer Science organizes the series of DIMACS challenges, and the 9th time the challenge focused on the topic of shortest path computation [DGJ09]. The challenge follows along a series of publications, also referred to as the shortest-path horse race, that peaked interest into techniques to speed-up shortest path algorithms. In the following paragraphs, we take a brief look at the results of the horse race and the challenge.

As we discussed previously, Dijkstra's algorithm does not only compute the distance from the source to the target but also to all vertices reachable by a shorter path than the path from the source to the target. A simple way to reduce the overhead hidden in that fact is described in [Dan62] and schematically illustrated in Figure 3.1. Instead of just searching from the source to the target, we can search starting from both the

[1]We refer to the most used reference, despite the considerations mentioned in Chapter 2.

[2]Held in the DIMACS Center, Rutgers University, New Jersey, in November 2006



(a) *Unidirectional* **(b)** *Bidirectional* **(c)** *Goal-Directed*

Figure 3.1: *Schematic illustration of the classic implementation of Dijkstra's algorithm, the bidirectional variant and a goal-direction technique.*

source and the target at the same time. Imagine the search to operate roughly in a cyclic manner around the source (or the target respectively in the bi-directional case): executing the algorithm from both vertices at the same time about halves the necessary computational effort. Schematically, we present the reason for this reduction in Figure 3.1 (left and middle).

On a Euclidean plane, we can imagine Dijkstra's algorithm to operate in a circular manner around the source (and target). The algorithm finishes when the circle around the source reaches the radius $\mathcal{D}(s,t)$. In a bidirectional search, ideally the searches reach each other directly in the middle. So instead of looking at all vertices in the circle around s of radius $\mathcal{D}(s,t)$, we look at all vertices in the two circles around s and t, both of radius $\frac{\mathcal{D}(s,t)}{2}$.

As the concept of bidirectional search forms a central element of the following techniques or can at least be applied to many of them, we give a few more details at this point.

Bi-Directional Search. In a bi-directional implementation, we perform a forwards directed search from the source and a backwards directed search starting at the target. The search that begins at the source operates as usual, in the search from the target we consider the reversed graph. Both searches operates as usual on their own priority queues. However, in addition to the normal procedure, we also issue a request for the current tentative distance of the arc's destination in the priority queue of the opposite search. In case the vertex has been reached in the opposite priority queue as well, we combine both distances to a full path and remember the vertex as a potential meeting vertex. The minimal combined distance over all found vertices marks the shortest path. The search can be stopped the moment a vertex v has been settled in both directions. It is guaranteed that the algorithm will have encountered a fitting meeting vertex at the time the stopping criterion triggers, the path does not necessarily contain v, though.

The choice of which search to advance can be made freely. The most common choice is a simple alternating concept. Another approach, could be to choose the priority queue based on its size or the currently contained minimal label.

Still, even when executed as a bi-directional algorithm and using highly optimized priority queues, Dijkstra's algorithm takes too long for many interactive applications. For some time, navigational providers have been turning to heuristic methods to speed up shortest path computations; one example would be to explicitly limit the search to highways after a given travel time or preselecting different detail levels (e.g. [Sed96]). The research community, on the other hand, has slowly entered into a race for the fastest computation of shortest paths. Following the assumption of a static network for which a large amount of queries has to be processed, researchers have found different ways to process the graph in advance to hasten the query algorithm later on. The algorithms discovered follow a range of different concepts. In the following sections, we take a look at these concepts and discuss some important contributions to the respective paradigms. First, however, we introduce an additional concept: search spaces.

Definition 3.1 (Search Space)**.** *The search space with respect to a speed-up technique or pruning technique is the set of vertices of a graph $G(V, A)$ that is considered during the search for a shortest path. It describes a subset to V.*

Most techniques still employ a bi-directional instance of Dijkstras' method. They operate on a modified graph, though.

3.1 Goal Direction

The first concept studied in the literature was goal-direction. Whereas Dijkstra's algorithm searches into all possible directions (in the classic sense, a graph does not necessarily come with an embedding in the plane), goal-direction techniques allow for a more natural way of shortest path computation. On a classic map, the natural approach is to identify the source and target and subsequently look for a path between the two. In the process, we would naturally only consider routes that are located in the region between them.

At this point, we cover two main techniques: A* and Arc Flags. For other techniques – namely Precomputed Cluster Distances, Geometric Containers, and Compressed Path Databases – we refer to [BDG$^+$15].

A*. A classic method for goal-direction search is the A* algorithm [HNR68]. The algorithm employs a potential function $\pi := V \mapsto \mathbb{R}$ that usually forms a lower bound on $\mathcal{D}(v,t)$. This function offsets vertices in the priority queue during an execution of Dijkstra's algorithm. The offsets changes the processing order to prefer vertices during the search that are closer to the target; see Figure 3.1 for a schematic illustration. If π were an exact lower bound ($\pi(v) = \mathcal{D}(v,t)$), this method would scan only vertices that lie on a shortest path between the source and the target.

Due to the underlying Dijkstra's algorithm, a requirement for correctness is the difference $\mu(v,w) - \pi(v) + \pi(w)$ being larger or equal to zero[3]. In that case, we call the potential function *feasible*.

Similar to Dijkstra's algorithm, an A* search can be executed as a bi-directional search, the correctness of the algorithm requiring *consistency* between the potential for the forward (π_f) and backward search (π_b) [IHI+94] or a modified stopping criterion [GH05, KK97]. One way of achieving this consistency is to create dual feasible potential functions by using $(\pi_f - \pi_b)/2$ for the forward search and $(\pi_b - \pi_f)/2)$ for the backward search.

The obvious lower bound to use is the geographical distance between two vertices, divided by the maximum travel speed [Poh70, SV86]; this heuristic, however, performs poorly in practice and the performance gain is negligible [GH05]. As a different approach with a far better performance, Goldberg and Harrelson introduced special vertices called *landmarks.* Such landmarks can be used in combination with the triangle inequality to derive a feasible potential function [GH05]. Based on the small set of precomputed distances for all landmarks, the triangle inequality produces far better bounds than the previous approach. The combination came to be known as the A* Landmarks Triangle Inequality (ALT) algorithm. It is one of the few algorithms that has been configured for an external memory setting [GW05].

Arc Flags. Arc Flags [HKMS09, Lau09] form another approach to goal-direction that is currently in use in some mobile navigation systems. A partition into K roughly *balanced cells*[4] forms the foundation to this technique. For each of these cells, a single bit is stored at every arc. The bit indicates whether the respective arc is part of any shortest path directed into the associated cell. During a search to a vertex in cell i, any arc can be omitted for which the $i-$th bit is not set. The obvious trade-off between the number of bits to be stored at every arc and the size of the partitions manifests itself in strong fanout when the search closes in on the target-partition. A multi-level approach [MSS+06] and a bi-directional search [HKMS09] alleviate this problem, though.

Computing these flags for the currently fastest goal-direction technique, however, is

[3]Executing Dijkstra's algorithm by offsetting the respective keys by $\pi(v)$ is equivalent to the reduction of the arc cost as specified here

[4]cells with about the same number of vertices

expensive. [HKMS09] and the recent PHAST algorithm [DGNW13] improve the cost to manageable regions, making the technique more competitive.

3.2 Separator-Based Techniques

Separator-based techniques construct an overlay graph to a graph $G(V, A)$ with the help of separators as described in Definition 2.7. In general, a separator-based technique computes a – preferably small – set of vertices or arcs to decompose the graph into a set of (ideally) balanced cells. Even though road networks in general are not planar, think of bridges or tunnels, small separators have been shown to exist [EG08, DGRW11, SS12b], similar to planar graphs [LT79]. As already discussed, we distinguish between two basic concepts: vertex-separators and arc-separators.

Vertex-Separators. A vertex-separator is a – preferably small – subset $V_s \subseteq V$ of vertices that decomposes $G(V, A)$ into a set of cells. This subset can be used to create an overlay graph by introducing shortcuts [VV78] that preserve distances between any two vertices in V_s. Since distances between vertices in the overlay graph represent distances in G, we can use the small overlay graph to speed up the query algorithm. Initially, a bi-directional search starts in G both at the source and at the target. When the search reaches a border vertex, it switches to the overlay graph and continues within it, bridging entire regions of the graph via a small set of shortcuts. The literature shows multiple variations of this technique. For example, Schulz et al. [SWW99] choose a small set of important vertices (V_s) – not necessarily a separator – and construct an overlay graph from this set. The general concept can also be extended to a multi-level version [SWZ02]. The quality of the results is subject to an obvious trade-off between memory consumption and experienced speed-up [GSVGM98, JHR98, DHM$^+$09].

Arc-Separators. Closely related to the vertex-separators, the arc-separator approach is based on a partition of the graph into k cells $(\mathcal{C}_1, \ldots, \mathcal{C}_k)$. For each cell, shortcuts represent the distances between the respective border vertices of the cell. A query algorithm runs a bi-directional implementation of Dijkstra's algorithm that, whenever it reaches a border vertex, switches to the overlay-graph comprised by the border arcs and the newly introduced shortcuts. Similar to the vertex-separators, the general approach can be extended to multi-level separators.

An early implementation, *Hierarchical MulTi* (HiTi) [JP02] implements this idea on grid graphs and only results in moderate speed-ups. Recent advances in the field of graph-partitioning (e.g. [DGRW11, SS12b, SS13, Sch13]) have led to a rediscovery of this approach in the form of Customizeable Route Planning (CRP) [DGPW11, DGPW13]. CRP is tailored to road networks and splits the preprocessing into two phases, a metric-independent preparation phase and a customization phase. The metric-independent stage concerns itself only with the structure of the graph, not with the actual metric. This phase requires a good multi-level partition and can take a long time. The customization phase, however, is optimized to be fast and allows to update metrics almost instantly or even to incorporate new ones.

3.3 Hierarchical Approaches

A third approach to speeding up Dijkstra's algorithm is to try and exploit the inherent hierarchy of road networks. Shortest paths of sufficient length tend to converge to a small arterial network, e.g. interstates or highways. As an intuitive heuristic [EPV11, HM11, FSR06], we could restrict the search to predefined road categories after a given distance traveled.

In the following paragraphs, we shortly introduce some techniques that employ the general idea of a natural hierarchy in the graph to speed up queries while maintaining correct results.

Reach. Initially, reach is a centrality measure defined in relation to the vertices of a graph. With respect to a shortest path $\Pi_{s,t}$, the reach of a vertex $v \in \Pi$ is defined as the minimum of $\mathcal{D}(s, v)$ and $\mathcal{D}(v, t)$. Based on this definition, we name the maximum reach with respect to any shortest path that contains v the global reach of v. The method, originally introduced by Gutman [Gut04], operates similar to Dijkstra's algorithm but prunes the search at any vertex v for which both $\mathcal{D}(s, v)$ and $\mathcal{D}(v, t)$ are larger than $r(v)$. To compute exact reach values for all vertices, a full all-to-all shortest path computation is required. The search remains correct, however, if $r(v)$ only depicts an upper bound on the exact value. Based on this premise, Gutman [Gut04] argues that even reach values deducted form partial shortest path trees are able to speed up the computation of shortest path distances. Goldberg et al. [GKW09] show that the inclusion of shortcuts can both speed up the preprocessing and provide better bounds by effectively reducing the reaches of most vertices.

Highway Hierarchies. Highway Hierarchies (HH) are basically a clever implementation of the formerly described simple heuristic for hierarchical route planning. Rather than relying on road categories, HH build their own (multi-level) highway network [SS05, SS12a]. To put it simply, HH define all arcs that are part of some shortest path but not close to their respective source/target to be highway arcs. In an iterative process, *localized searches* find such highway arcs. The result are being used to shrink the graph by restricting it to the highway network and contracting vertices (see below for a closer description of vertex contraction) with low degree. For a detailed description of this technique, we refer to [Sch08a].

Highway Node Routing. Highway Node Routing (HNR) takes a different approach to the idea of an overlay graph to represent an important subnetwork. Sanders and

Schultes [SS07] describe the overlay graph as the (ideally) minimal graph that preserves shortest path distances between a (preferably small) set of *important* vertices. These vertices are chosen in such a way that they cover as many shortest paths as possible. The technique can be extended to a dynamic scenario [SS07] if the metric does not change completely and keeps most of the inherent hierarchy of the road network intact.

Contraction Hierarchies. As a successor to HH and HNR, Contraction Hierarchies (CHs) [GSSV12] follow the same basic algorithmic ideas but are both faster and conceptually simpler. The basic idea, which also lends its name to the whole technique, is a procedure called vertex-contraction. The contraction of a vertex v creates an overlay graph G^{+} to a graph $G\left(V, A\right)$ by removing v from G and inserting shortcuts to preserve shortest path distances between the remaining vertices.The necessity of a shortcut is usually determined by a local query in $G \setminus v$ between every pair of neighbors u, w and a comparison to the two hop path u, v, w. These local searches are called witness-searches and a path that proves a shortcut to be unnecessary due to its shorter length is called a witness. After each contraction, the resulting overlay graph is used as an input for the next contraction step.

The localized view of the problem, distilled in the single contraction operation, results in a surprisingly efficient and elegant speed-up technique. At first glance, it might seem that an n-level hierarchy could not capitalize on the hierarchical structure of a road network efficiently; on a closer look, however, the independence of two contractions in different not directly connected parts of the graph results in a rather shallow hierarchy. CH can handle dynamic scenarios to a certain degree. In the dynamic scenario, a shortest path is checked for potential delays present in the network. If such a delay is present on the path, the contract operation is partly undone and recomputed with the updated information.

The simplicity of CHs resulted in a fair range of different extensions and continuative concepts. For further reading, we direct to [BDG^{+}15].

3.4 Additional Techniques

Next to the different speed-up techniques for one-to-one queries, some additional techniques offer benefits beyond the pure point-to-point queries, some of which we discuss in the following paragraphs.

PHAST. Even though there has been a lot of research regarding point-to-point shortest path computations, 50 years after the invention of Dijkstra's algorithm it was still the main single core algorithm (compare [MS03] for a parallel implementation) to compute complete shortest path trees. Modern Central Processing Units(CPUs), offering fast linear memory access by means of prefetching, allow for a faster approach described in [DGNW13]. The basic approach requires the construction of a CH. By ordering the vertices with respect to their hop distance from the topmost vertex, Parallel Hardware Accelerated Shortest Path Trees (PHAST) can operate in two phases. In the first phase, a classic upwards search within the CH initializes a subset of the distance entries. Due to the bi-directional search mechanism, it especially sets the correct distance value for the topmost vertex. In the second phase, PHAST *sweeps* (or scans) over all distance values: for each vertex, all arcs in the backward graph are scanned and distance values updated when appropriate.

The main observation behind PHAST, described in the process above, is that we can exploit the up-down characteristic of a shortest path in a CH. The first phase looks for the upwards parts of all the possible shortest paths. Even though some distances found this way may not be optimal, one can easily accept that the (by requirement) optimal parts of all up-down paths will be found. The second phase only concentrates on the downwards part, extending previously found upwards paths downwards. By operating from the top of the hierarchy to the bottom, we guarantee that the distances we operate on are always optimal.

The simple access pattern with its linear write access allows for speeding up Dijkstra's algorithm in its classical one-to-all form. Even better (amortized) speed-ups are possible when using *vector instructions*[5] to compute multiple shortest path trees at the same time.

Bounded Hop Techniques. A further concept involving shortest paths are bounded hop techniques. These techniques usually aim at computing only $\mathcal{D}(s,t)$ rather than the full path; they can, of course, be used to compute shortest paths by simply describing a sequence of local decisions. The literature presents two different approaches to this paradigm. At this point, we only give a brief overview of both techniques, as they are beyond the scope of this work. For an in-depth overview of these techniques, we refer to [BDG⁺15].

A first representative is the *labeling algorithm* [Pel00, GPPR04]. The algorithm computes a *label* L to each vertex in such a way that $\mathcal{D}(s,t)$ can be determined by intersecting $L(s)$ and $L(t)$. Recently, this approach has been rediscovered and improved [ADF⁺11, ADGW11, ADGW12, DGW13], making it the currently fastest method to determine $\mathcal{D}(s,t)$. Some of its applications even reach into shortest path distances in databases [ADF⁺12].

The second paradigm of bounded hop techniques that has seen quite a lot of attention is Transit Node Routing (TNR) [SS06, BFSS07, BFM⁺07, Sch08b, BFM09, ALS13]. In TNR, we select a small set of vertices, the so-called transit nodes, and compute a full distance table between them. For every other vertex, i.e. the ones not in the set of transit nodes, a small set of *access nodes* (a subset of the transit nodes) is computed and the distance to and from these access nodes is stored. The query TNR distinguishes between *local* and *global* queries. A global query is any query that passes

[5] A set of instructions that apply the same operation to multiple data-elements at the same time

to both an access node of the source as well as an access node of the target. The desired distance value can be determined by a quadratic combination of the (small) sets of access nodes and is the result of three table lookups per possible combination. The local queries cannot be answered using the precomputed distance tables but rely on an execution of a classic implementation of Dijkstra's algorithm. Due to the local character of the request, however, the computational overhead is low. The decision whether a query is global or local is usually subject to a heuristic (locality filter) that is allowed false positives on the locality but no false negatives. Similar to a wide range of other techniques, TNR can be extended to multiple levels of transit nodes.

Chapter 4

4 Shortest Paths Details

As they say, the devil is within the details. This thesis heavily relies on a deep insight into two current speed-up techniques. The following chapters aims at providing these details.

In the previous chapter, we introduced a series of techniques to speed up Dijkstra's algorithm. Two of these techniques, CH and CRP, form the foundation for the techniques we present in Chapters 7 through 14. To be able to give a thorough description and analysis of our approaches, we now describe both techniques in far greater detail than in the previous chapter.

4.1 Contraction Hierarchies

Unless otherwise specified, we base most of the description in this section on the work of Geisberger et al. [GSSV12].

4.1.1 Construction

Previously, we already described the core of a CH to be the vertex-contraction operation. Formally described in Algorithm 4.1, the operation builds an overlay graph through the removal of a single vertex. For the overlay graph to describe the input graph, all shortest path distances between the remaining vertices must be preserved. This can be achieved by creating so-called shortcuts. In a CH, a shortcut represents a – not necessarily shortest[1] – path comprised of two arcs (or by extension shortcuts).

To decide on the necessity of a shortcut, localized searches, called *witness*-searches, are used: during the contraction of the vertex v, we compare the distances in the tentative

[1]Some shortcuts might be sup-optimal, due to a usually restricted search that decides on the necessity of a shortcut.

Algorithm 4.1 Vertex-Contraction

Input: A weighted graph $G\left(V, A\right)$ with associated metric μ, a vertex v
Output: An overlay graph $G^+\left(V^+, A^+\right)$ with $V^+ := V \setminus v$ and $\mathcal{D}\left(s,t\right)$, $s,t \in V^+$ in G^+ is equal to $\mathcal{D}\left(s,t\right)$ in G for all s, t

```
1: $V^+ = V \setminus v$                                          ▷ The final vertices
2: $A^+ = A \setminus \{(u,v)|u \in V\}$                          ▷ Temporary set of arcs
3: for all $u, w : u, w \in V^+ \wedge (u,v),(v,w) \in A$ do      ▷ Loop over all pairs of neighbors
4:     if $\mathcal{D}\left(u,w\right)$ in $G^+\left(V^+, A^+\right) \neq \mathcal{D}\left(u,w\right)$ in $G$ then
5:         $A^+ = A^+ \cup (u,w)$                                 ▷ If distance changes, insert a shortcut
6:         $\mu(u,w) := \mu(u,v) + \mu(v,w)$
7:     end if
8: end for
9: return $G^+\left(V^+, A^+\right)$
```

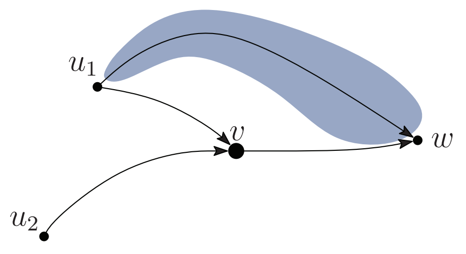
overlay graph between all vertices $u_{in} := \{u|(u,v) \in A\}$ and $w_{out} := \{w|(v,w) \in A\}$ to $\mu\left(u \in u_{in}, v\right) + \mu\left(v, w \in w_{out}\right)$. If the direct path through v is shorter than the witness, a shortcut is constructed and added to the overlay graph. If a witness-search prevents the addition of a shortcut, the respective path is called a witness.

Figure 4.1 illustrates the process of contracting a vertex v. The arc u_1, w forms a witness to u_1, v, w and a shortcut can be omitted. The shortcut u_2, w has to be created, due to the only shortest path between u_2 and w requiring v. During the contraction process, the length-values of the remaining (shortcuts) arcs get increasingly larger and the graph gets denser. As a result, witness-searches get increasingly expensive. This problem is usually met by restricting the search for witnesses to a maximal number of hops. While this restriction may result in an increased number of shortcuts that are added to the overlay graph, the heuristic does not affect the correctness of the approach.

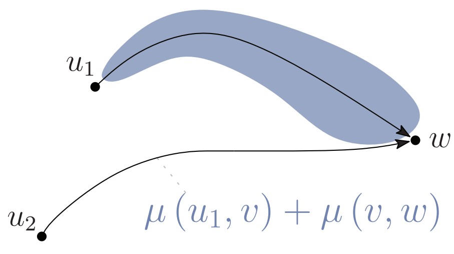
To construct a CH, the vertex-contraction operation is used to create a series of overlay graphs by iteratively contracting all vertices. The vertices are contracted following an *order of precedence* ($\prec$) that classifies the *importance* of vertices in the hierarchy; for $u \prec v$, we say that v is more important than u.

Finding a good order of precedence for the contraction process is hard [BCK+10], however on-line heuristics, involving for example the (currently) necessary number of shortcuts, can determine reasonable orders during the contraction process. If desired, we can parallelize the contraction by operating on independent sets of vertices in parallel [Vet09].

The result of the full process is both the order and a list of additional shortcuts which, in combination, allow for the definition of a *forward graph* $\left(G^{\uparrow}\right)$ as well as a *backward graph* $\left(G^{\downarrow}\right)$.



(a) *Input*



(b) *Contracted Result*

Figure 4.1: *Contraction of vertex v: arcs are assumed to have unit cost. The path u_1, v, w is not a (unique) shortest path and no further action has to be taken. u_2 and w would become disconnected after the removal of v. A new arc (shortcut) of weight $\mu(u_2, v) + \mu(v, w)$ keeps shortest path distances within the resulting overlay graph the same as in the original graph. u_1, w does not necessarily represent a direct connection but can stand for a whole path that has to be discovered by means of a witness-search.*

Definition 4.1 (CH graphs)**.** *Given an order of precedence $\prec$ on the vertices of a weighted graph $G(V, A)$ and the list of all shortcuts A_s created during the iterative contraction process of the CH preprocessing:*

We define $G^{\uparrow}$ as $G^{\uparrow}\left(V, A^{\uparrow}\right)$ with $A^{\uparrow} := \{a = (u, v) | a \in A \cup A_s \wedge u \prec v\}$. Analogously, we define $G^{\downarrow}$ as $G^{\downarrow}\left(V, A^{\downarrow}\right)$ with $A^{\downarrow} := \{(v, u) | (u, v) \in A \cup A_s \wedge v \prec u\}$. The metric for both graphs remains identical to the metric of the input graph G and the respective shortcuts A_s.

Staring from this definition we can move on to describe the query algorithm in the next section.

4.1.2 Query Algorithm

The query algorithm employed for finding a shortest path within a CH is very similar to the bi-directional implementation of Dijkstra's algorithm.

Query. To perform a query within a CH, we employ a *forward search* in $G^{\uparrow}$, starting at the source s, and a *backward search* in $G^{\downarrow}$, starting at the target t. Due to Definition 4.1, both directions run a classic implementation of Dijkstra's algorithm as specified in Algorithm 2.2. In the same way as for every bi-directional algorithm presented, we check for a potential meeting vertex that has already been reached in the opposite search direction and keep the best vertex. The structure of $G^{\uparrow}$ and $G^{\downarrow}$ restricts all paths to a very specific form, called an *up-down-path*, as both searches only go up in the hierarchy, looking at more and more important vertices. The combined path

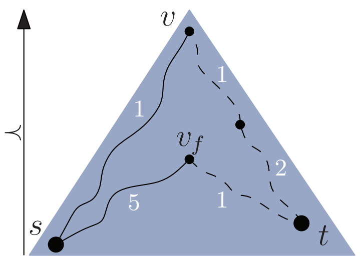


Figure 4.2: *Schematic diagram of a CH search from s to t. When settling vertices in an alternating order, starting at s, only the path s, v_f, t has been found at the moment both directions have settled v_f. The correct shortest path uses v, though. This illustrates the typical structure of a shortest path in a CH (up-down path) as well as the requirement for an adjustment to the stopping criterion. Arcs of the forward search are illustrated as solid lines, arcs of the backward search as dashed lines.*

from s over the meeting vertex v to t goes upwards from s to v and, since we have to reverse the path found in $G^{\downarrow}$, down from v to t. The construction method of a CH guarantees [GSSV12] that for any shortest path in the input graph, such an up-down path exists in the CH with the same cost. One thing, however, is different to the classic bi-directional implementation of Dijkstra's algorithm: the CH requires the separation of the forward graph $G^{\uparrow}$ and the backward graph $G^{\downarrow}$. As a result, we cannot stop the search the moment a vertex has been settled in both search directions.

Figure 4.2 illustrates both the typical up-down characteristic as well as the reason for the adjusted stopping criterion. Instead of searching downwards in the direction of t, the forward search only considers v and v_f. During a ordinary instance of Dijkstra's algorithm, t would be settled before we even settle v_f in the forward search. In the CH query, however, we cannot descend into the direction of t (or s). It is possible to simulate the classical bi-directional search, though, by first marking all arcs reachable by source and target and to allow the descent in the direction of the target. The overhead required to first explore the entire search space of both directions, however, does seem wasteful for short-ranged queries.

Instead, in a ordinary CH query, we focus on a different kind of stopping criterion. When the search spaces meet, we continue the search until the minimum weight in the priority queue represents a longer path than the currently found best one. This approach is reasonable as, if the shortcuts are distributed evenly, the shortcuts of arcs higher up in the hierarchy get longer and longer, resulting in a far lower overhead than a general full exploration of the search spaces.

Path Extraction. Opposed to Dijkstra's algorithm, extraction of a shortest path from a CH is not as trivial as following the parent pointers. Arcs most often do not represent actual road segments in the network. Most often, they stand for a whole sequence of these segments combined into a single shortcut. Still, parent pointers form the initial base to extract a path from a CH. If an encountered arc turns out to be a shortcut, the shortcut is unpacked.

Unpacking a shortcut can be handled in different ways. Most common are the two extremes of either storing a complete list of all unpacked shortcuts to retrieve the represented path from, or of recursively unpacking the shortcut. This process to recursively unpack the shortcut on the fly only requires a single additional piece of information that has to be stored with the shortcut: the contracted vertex that resulted in its creation.

For the extraction of a shortcut arc (u, w) and middle vertex v, we scan the associated arcs of v. As v was contracted before u and w, $v \prec u, w$ holds. As a result, we can scan v for two arcs (u, v) and (v, w) . This process replaces a shortcut with two arcs that originate at its middle vertex (one in $G^{\uparrow}$, the other on in $G^{\downarrow}$) and that represent the same path as the original shortcut. The process terminates, as for every arc one of its vertices is replaced with a vertex lower down in the hierarchy (when unpacking (u, w) to (u, v) and (v, w), we have $v \prec u, w$).

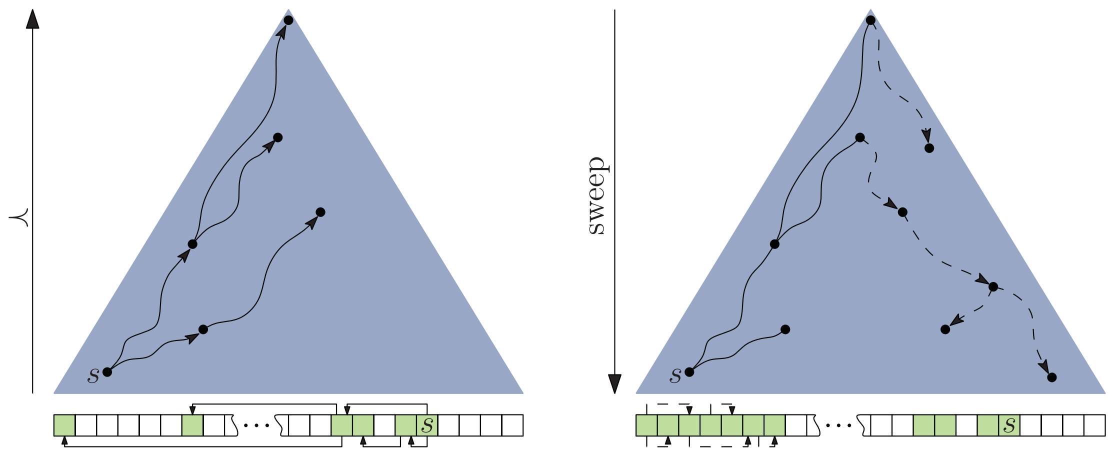
4.1.3 Batched Shortest Path using Contraction Hierarchies

The characteristic of all shortest paths found in a CH – remember the up-down structure – lends itself to a totally different concept: batched shortest path computation [DGNW13]. Delling et al. present PHAST, an algorithm that, in the same way as Dijkstra's original algorithm, computes full shortest path trees (the distance from a given source to all other vertices). Their groundwork is a specially preprocessed CH that relabels vertices with respect to the reversed order of precedence; the last vertex contracted is labeled as zero, the first one is labeled $n - 1$. The proposed algorithm operates in two phases. The first phase is comprised of a ordinary forwards directed search in $G^{\uparrow}$. It labels all reachable vertices by assigning them their shortest path distance in $G^{\uparrow}$. The number of vertices touched in this phase is rather small, due to the small search space size of a CH.

In a second phase, all vertices are scanned, starting at the most important vertex, considering only arcs in $G^{\downarrow}$. Remember that a shortest path within a CH consists solely of a single upward and a single downward part (both of which may be missing entirely). The upward parts of all shortest path in the full graph are found by the first phase already. In the second phase, PHAST now extends all existing upward paths by the respective downward parts. The order in which the assignments are performed guarantees that all distances are updated correctly. The main benefit of this approach is the linear access pattern employed in the costly second phase.

Even though Dijkstra's algorithm can be implemented in almost linear time [Tho04], the, in modern CPUs most effective, linear scanning pattern in combination with low overhead due to simple data structures allows for a very fast computation of full shortest path trees.

The combination with vector instructions or even General Purpose Graphics Processing Units(GPGPUs) allows for the parallel computation of multiple such trees at a time. The low number of vertices in the topmost levels of a CH leaves a large change for some of the required distance values to reside in a cache. In this way, the technique is somewhat related to time-forward processing [CGG+95].



(a) *Upward Search* **(b)** *Downward Sweep*

Figure 4.3: *The two phases of the PHAST algorithm. The first stage is comprised of a standardized upward search in the forward graph. In the second stage, all arcs of the backward graph are relaxed (if necessary) to compute shortest path distances in a top-down manner. The sweep phase is depicted in progress, having set distance values for a part of the result array.*

Schematically, we illustrate the way PHAST operates in Figure 4.3. The first phase, the upward search in $G^{\uparrow}$, operates just as in a standardized CH query. Instead of checking another priority queue for potential meeting vertices, though, the search is performed to exhaustion; to prepare for the second phase, it is setting an initial tentative distance for all vertices reachable from s in $G^{\uparrow}$.

From these initial distance values, two are known to be exact. The source (s) of a search is always known to be reachable in zero time. In addition, the topmost vertex is no origin to any arc, neither in $G^{\uparrow}$ nor in $G^{\downarrow}$. Following an iterative argument, we can extend all shortest paths from top to bottom in the second phase: During the computation of $\mathcal{D}\left(s, v\right)$ for any vertex v that is not reachable optimally via a shortest path in $G^{\uparrow}$, a vertex exists that is the topmost vertex on an up-down path from s to v. The distance of the topmost vertex is known to be correct and all vertices reachable from v in $G^{\downarrow}$ are known inductively to be correctly assigned their distance. This implies that v is either reachable optimally via a shortest path in $G^{\uparrow}$ or we can extend any up-down path via a single arc to find the correct distance for v.

4.1.4 Dynamic Implementation

For the purpose of many speed-up techniques, road networks were thought of as static entities. At least for the structure of the network, this assumption is mostly sound, as new roads do not come into existence over night and preprocessing modern speed-up techniques can be done in a rather short amount of time: many of them only take minutes or at most hours and can be adjusted to incorporate structural changes that occur rarely in comparison with traffic jams.

These structural changes are in contrast to both expected and unforeseeable delays. The somewhat predictable part, e.g. the morning or evening rush, is usually modeled in form of time-dependent queries (cf. Section 2.3.2). Next to specific implementations aimed at time-dependent queries – see [Bat14] for a detailed discussion – CH has been adopted in some instances to support dynamic queries due to unpredictable delays, traffic jams, or road closures [GSSV12]. The authors present two different approaches to enable dynamic changes to the network. The first follows the idea to locally correct the CH to incorporate dynamic changes on the fly; to do so, one needs to keep track of the dependencies between shortcuts to partially unwind the contraction process and reconstruct the CH locally. In a second approach, the authors propose to change the query algorithm to allow for any form of delay. Instead of a simple query, the algorithm checks for sanity of the found route by checking all used arcs for associated penalties. If no arc is affected, the result is correct and can be presented to the user[2]. Should one of the arcs be linked to a current delay, however, the query is rerun and allowed to look downwards in the hierarchy around the affected part of the path.

4.1.5 Theoretical Results

The concept of a CH is surprisingly simple, yet it is even more surprising, at least from the standpoint of a theoretician, that it works at all. For some time, no theoretical justification existed that explained why a CH could offer sub-millisecond queries.Even though the actual practicability is the main goal of any speed-up-technique mentioned, in algorithm engineering (see Figure 1.2), not only the implementation and experiments are important but also, to close the cycle, the theoretical justification.

For general graphs, it is most unlikely that any theoretical justification for any of the presented techniques exists. In fact, there are inputs for which most techniques are known to be ineffective. That said, for specific graph classes non-trivial bounds have been proven. In the following paragraphs, we give a short overview on the theoretical results, which we pick up again in Section 11.3.1.

Special Graph Classes. Independently, different authors [Mil12, BCRW13] observed a natural connection of CH to filled graphs [Par61] and elimination trees [Sch82].

[2]In contrast to locally correcting the CH, only delays can be modeled this way and no arc can be reduced in its cost.

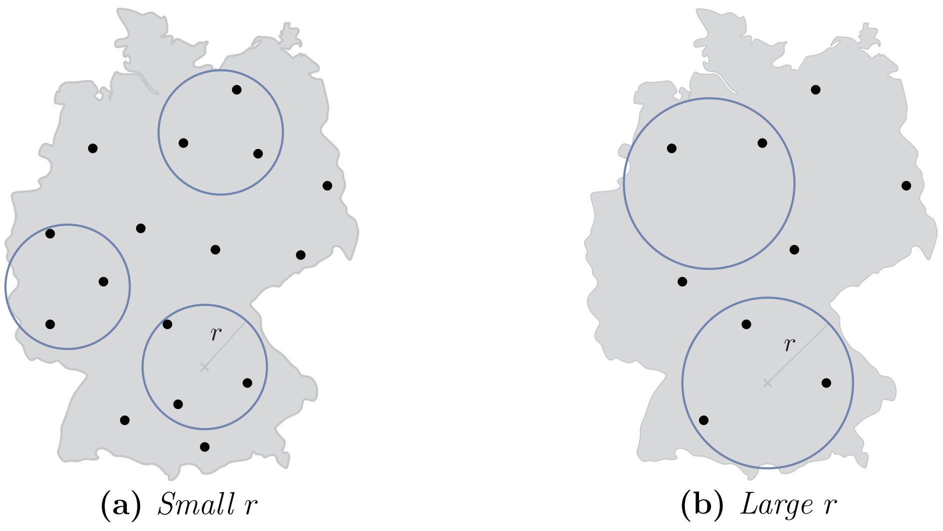


Figure 4.4: *Concept of highway dimension and the corresponding hitting set (black dots). The set covers all shortest paths of length at least r. The highway dimension for the depicted scenario would be three, depending, of course, on other radii and hitting sets.*

The connection allows for a simple application of nested dissections [LRT79] on planar graphs to find an order of precedence for a CH with $\mathcal{O}(n \log n)$ shortcuts. On graphs with treewidth w, the search space of a CH can be bounded by $\mathcal{O}(w \log n)$ [Mil12], on minor-closed graphs with $\mathcal{O}(\sqrt{n})$ separators it can be bounded by $\mathcal{O}(\sqrt{n})$ [Mil12]. Such separators of size $\mathcal{O}(\sqrt{n})$ can also be found in planar graphs [LRT79, LT79], a class of graphs closely related to road networks. Even though road networks themselves are not planar, mostly due to bridges and tunnels, small separators are known to exist [EG08, DGRW11]. The similarities between road networks and planar graphs or even grid graphs – a type of graph that can be found as a basic element of many cities, e.g. Manhatten – cannot be carried over completely; compare [Som14] for a study of various methods and their trade-offs. Most techniques show a stronger performance on actual road networks in comparison to other artificial graphs.

All these results are to be considered carefully though, as running time does not directly relate to the search space. Larger vertex degrees might result in worse running times than one would expect.

The principle of algorithm engineering asks for a theoretical justification for these discrepancies. One possible justification is due to Abraham et al. [AFGW10]:

Highway Dimension. Abraham et al. propose a measure for graphs called *highway dimension* [AFGW10, ADF$^+$11, ADF$^+$13]. Intuitively speaking, highway dimension describes the diversity of all shortest paths of a given minimal length. Similar to HNR, we try to cover all shortest paths with a *locally sparse* hitting set. Such a hitting set to a length r covers all shortest paths if every such path of length at least r contains a vertex from the set. Abraham et al. define the highway dimension (h) of a weighted graph $G(V, A)$ in relation to such a hitting set $S \subseteq V$: h_S defines the maximal number of vertices from S contained in any ball of radius r around a vertex from V. The highway dimension of G is given as the minimal h_S over all possible hitting sets and radii r. TNR [BFSS07] is directly related to this notion and gives an experimental justification to the authors' claim that road networks have a small highway dimension. The concept directly translates into non-trivial bounds for CH and a few other techniques: A polynomial time preprocessing can create a CH with $\mathcal{O}(h \log h \log D)$ shortcuts, supporting a CH query[3] in $\mathcal{O}\left((h \log h \log D)^2\right)$ time, for a graph of diameter D. The concept has been tested on artificial networks [BKMW10, AFGW10, ADF+13] with provable small highway dimension.

Similar results have been found for other techniques as well. Due to their limited relevance to our contributions, we only cover CH-related results at this point.

4.2 Customizable Route Planning

The second technique that requires a closer introduction is CRP, an extension of the classic multi-level separators [HSW08]. Unless otherwise specified, we base this section on [DGPW11, DGPW13].

CRP is a technique based on multi-level separators, designed to handle changes to its metric in an efficient way. The technique splits the preprocessing in two, performing a separate construction and customization step.

4.2.1 Construction

The construction of the basic CRP data structures follows the idea of natural bottlenecks distributed throughout a road network. Be it in form of a highway network or actual natural borders like rivers or mountains, road networks usually offer a vast amount of different natural separators. These structural properties are completely independent of any actual metric one could come up with.

The basic idea ha been used before (cf. Chapter 3). Only recent advancements in partitioning techniques [DGRW11, Sch13] have made the idea viable for road networks on a continental scale, though.

The first step of the preprocessing is used for all parts that are purely a structural matter: It creates a multi-level partition of the network as illustrated schematically in Figure 4.5. The lowest partition level defines a set of small graphs, while all cells themselves are used to generate a full clique – a fully connected subgraph (see Figure 4.6) – that can be used to bypass a cell (compare Figure 4.7). We store these cells in the form of distance matrix.

[3]the same holds true for a Reach query

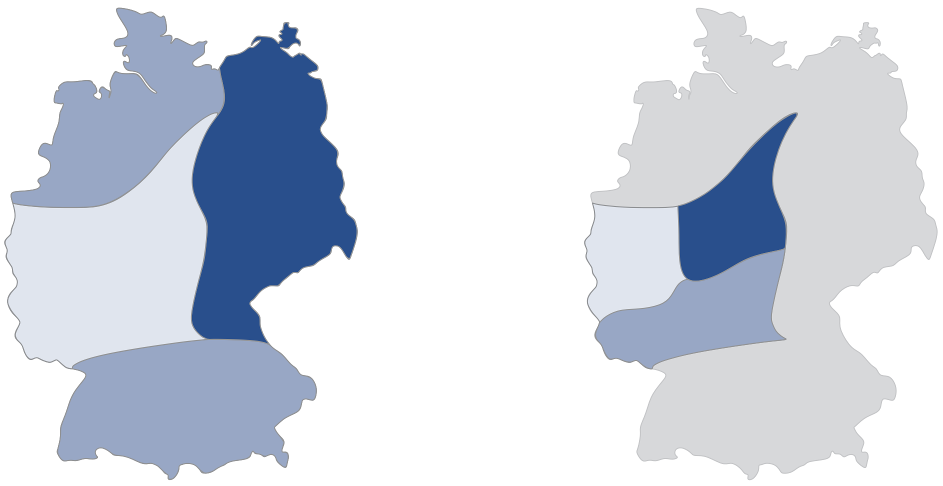

(a) *Top-Level Partition* **(b)** *Sub-partition*

Figure 4.5: *Schematic diagram of a multi-level partition of Germany with two levels. The second level is only shown for a single cell.*

When finished, the construction step provides the data structures that are required for the customization and the query of CRP. The main pieces of information we obtain are the multi-level partition, a vertex-mapping for the graph in accordance to this partition, and the layout of the cells that hold all the distance information. The vertex-mapping, which we describe next, offers some helpful optimizations during the query phase; the cells, at the current point, are purely handled in a structural fashion and do not contain valid distance information. This information is only computed during the customization.

Vertex Labelling. To simplify some processes down the line, CRP relabels vertices according to the following scheme: In a multi-level partition, border vertices on a given level are also border vertices on the level below. Each border vertex, however, has a topmost level for which it is a border vertex instead of an inner vertex. This level is considered the first sorting criterion and vertices that are border vertices on a higher level come first. Within a level, vertices are sorted by increasing cell id. The first vertex in this order is relabeled to the id zero and following vertices are assigned the respective next id. This labeling process allows to decide on the level that a vertex has to be processed on.

Shadow Levels. In an extension to the multi-level partition, we can utilize the power of graph-partitioning even further by creating some additional levels of partitions. These additional levels are referred to by the literature under many different names: shadow-levels, guidance-levels, or mezzanine-levels. These levels are only utilized during the metric-customization. At query-time, these levels are ignored. This approach does

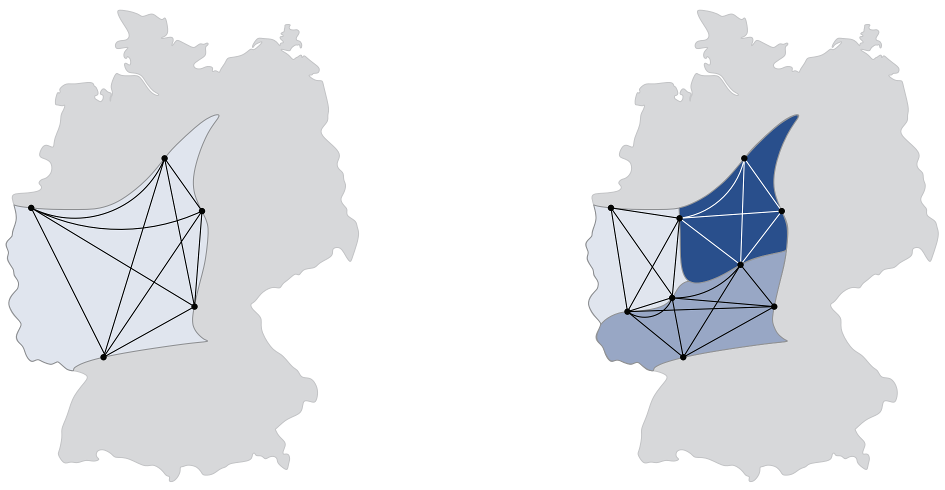

(a) *Top-Level Clique* **(b)** *Sub-Level Cliques*

Figure 4.6: *The resulting data structures after constructing the multi-level partition, shown for a single cell and its underlying partition.*

offer a decisive advantage during construction and only requires a rather small amount of additional information to be stored. The distance matrices for these guidance-levels are not stored permanently.

4.2.2 Customization

The second part of the preprocessing for CRP is a phase that is extremely fast in comparison to the construction. It can easily be executed for many possible metrics. In general terms, the customization is done in levels: Starting at the bottommost level, the clique-weights for the initial cell are computed, some of which might be of infinite weight. This first step operates on arcs of the original graph. The moment these cells are computed, only border arcs of the original graph contribute to the following steps. For the rest, the customization operates on cliques computed in the previous level. Delling et al. utilize two main constructs for customization:

Microcode. The first conception, microcode [DW13], follows the basic operations of CH preprocessing. Similar to the ideas developed independently in [BCRW13], microcode describes the process of building a CH, completely omitting all witness searches. For a transformation from Algorithm 4.1 to a so called micro-instruction, we realize that, if every shortcut is created, the actual operation does not depend on any information stored in the metric. Instead of keeping track of IDs or an explicit graph, we assume an arc, and by extension a shortcut, to be stored at a fixed memory position. Now, the full process can be interpreted as a simple stream of instructions that add up two values and store the result at a third memory position; this is, if the result improves the already stored value. These three values are exactly what comprises a micro-instruction. [DW13] also discusses the use of guidance-levels to create the microcode and to reduce the memory footprint by using macro-instructions, another set of instructions that lists $i + o$ input memory positions and $i \cdot o$ output locations. By mixing both types of instructions, the full process gets slowed down in comparison to pure micro-instructions due to additional branches and explicit loops. In total, however, these instructions allow for sub-second customization on multi-core servers.

Bellman-Ford. The second building block of CRP customization is the Bellman-Ford algorithm (compare Algorithm 2.3). Updating the distance matrix for a given cell, we require a solution to the all-to-all shortest path problem. Even though the algorithm of Floyd and Warshall (compare Algorithm 2.4) is an efficient algorithm for this problem, Delling et al. show far better performance for the Bellman-Ford algorithm; a reason for this is the very simple structure of the shortest paths we can expect during the customization. For the most parts, shortest path in our setting will only consist of a few hops, visiting each of the cells exactly once. The low number of iterations in combination with vector instructions makes for an efficient algorithm. Except for the bottommost level, which for the most time is handled using microcode, we only have to consider a small set of cliques connected by a few arcs in between. For the most part, shortest paths are expected to visit each clique once, due to the embedding in the plane. The expected low number of hops in a path – remember that the Bellman-Ford algorithm computes distances by the number of required hops – results in a very low number of iterations required to find the shortest path distances with respect to the current cell. In addition, the algorithm can be executed for multiple different sources in parallel using vector instructions – see [DW13] for a more detailed discussion.

4.2.3 Query Algorithm

The query algorithm for CRP is conceptually identical to other hierarchical techniques. Starting at the source and target, we search upward in the hierarchy defined by the multi-level partition. In simple terms, when we reach a border vertex v, the search switches to the highest level that contains v as a border vertex.

Following this procedure, the search can bridge entire regions (cells) of a graph using a single shortcut. Figure 4.7 illustrates this process for a short example path. Assuming both source and target to be inner vertices of a cell, the search first starts on arcs of the unprocessed graph. Reaching the border vertices of the associated cells, the search switches to the highest possible level and continues. In the example in Figure 4.7, the path only goes up a single level and bridges one cell via a shortcut. Vertices of bridged cells only become relevant during path unpacking.

In contrast to CH queries, as we already mentioned earlier, the search does not only go upwards in the hierarchy, though. In fact, it can be implemented as a purely

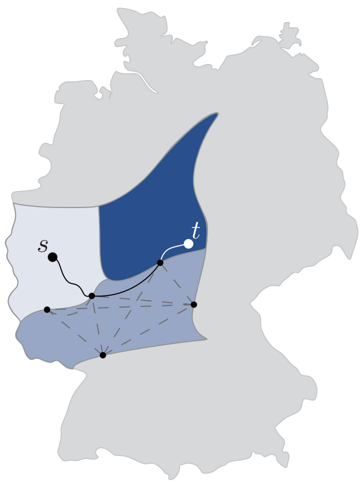
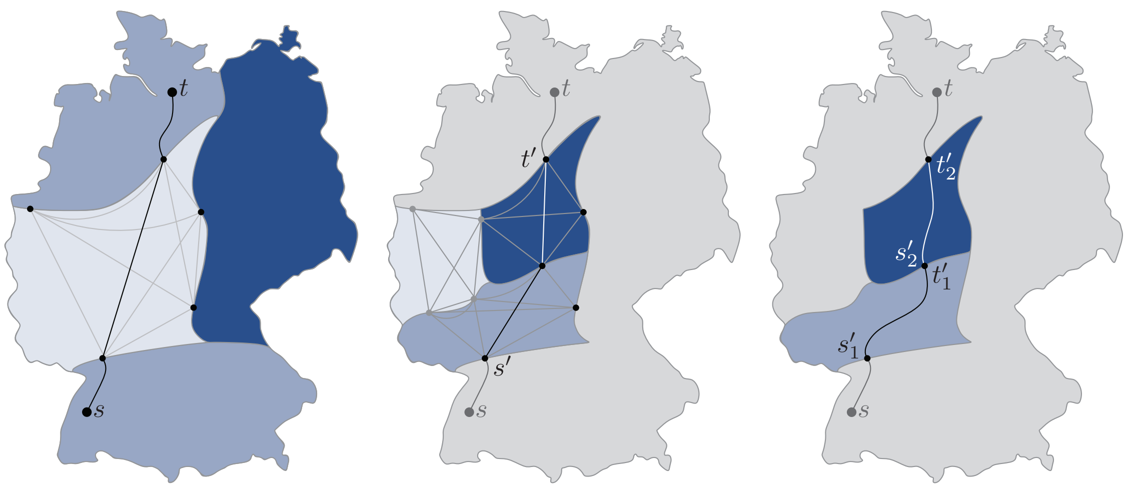


Figure 4.7: *Query in a multi-level partition. The path follows ordinary arcs until a border vertex is reached, takes a shortcut through a partition and follows ordinary graph arcs again in the partition containing the target.*

unidirectional search. For a CH, an unidirectional query would still require some way of knowing when to descend. In a CRP query, each vertex has a different set of arcs that might be valid. Which set is valid depends on the highest common level of a vertex in relation to the source and/or target. The highest common level can be determined purely based on its own id and the id of the target (or the source if searching backwards). The reason for this is the vertex labeling we described earlier. In simple terms, one can imagine the id of a vertex to be built out of different blocks. Each block describes the id of a cell that contains the vertex. Assuming c cells, the first $k := \log c$ bits of the vertex id label the cell on the topmost level. If two vertex ids differ within these first k bits, it is safe to search for the shortest path between them on the highest level. While this does not exactly describe the true label properties, it provides the general idea. The exact details can be found in [DGPW11] or [DGPW13].

Path Unpacking. Similar to a CH, a shortest path computed by a CRP query most likely contains shortcuts. To find the actual path, we have to unpack it. The shortcuts we encounter during a CRP query are different to those in a CH. Whereas each CH shortcut consists of exactly two arcs that might have to be extracted in a recursive manner, even a nested dissection [LRT79] does not guarantee a path structured as simply as in a CH. For CRP, shortcuts are unpacked using localized instances of Dijkstra's algorithm. These instances are induced by the cells on the level below of the shortcut bridging a higher level cell. The respective origin and destination of the shortcut are used as source and target vertices. As we illustrate in Figure 4.8, one might have to continue this process recursively as the path found in the cells on a given level might be comprised of shortcuts as well.



(a) *Top-Level* **(b)** *Sub-Level* **(c)** *Graph Level*

Figure 4.8: *Recursive unpacking process during a CRP query.*

The costly unpacking process, results in a far slower query than a CH query. This effect is even magnified by the larger search space of a CRP query in comparison to a CH query. Still, CRP allows for shortest path computations in the order of a few milliseconds, more than fast enough for any interactive service. The strength of CRP is in its simple structure and the fast customization. Additionally, storing multiple metrics is cheap. The trivial parallelization allows to integrate a new metric within less than a second and the required memory overhead in terms of shortcuts is less than a hundred megabytes. Supporting multiple metrics at once becomes cheap in comparison to a classic CH, for which each metric had to be processed independently. However, the recent customizable CH [DSW14] also offers this possibility for a CH.

4.2.4 Real Time Customization

One of our contributions is the utilization of GPGPUs to enable real time customization of CRP (see [DKW14]) for which I supplied the initial implementation. This contribution is a joined work with Daniel Delling and Renato Werneck.

While our approach does not allow for arbitrary customizations, we were still able to allow for a multitude of different encodings based on a classification and penalty system. A metric for this scenario is encoded in form of a few small lookup tables that contain multipliers. These tables can be adjusted at will, for example to make travel along a highway more costly or to penalize tunnels. The cost of an arc can be decoded from some base cost and references to these multipliers. By this method, we are able to store every bit of information required for customization on the GPGPU itself and only transfer a new table which is cheap compared to the gigabytes of data required for a whole graph. The transfer of the customized data back to the CPU, which is used to perform the queries, can be done asynchronously and finished nearly at the same time as the customization.

As this process of fast GPGPU-based customization plays no major role in the techniques we discuss later on, we will only dedicate a few small paragraphs to these results, even though they enable a whole new degree of personalization.

Contraction Customization. In Section 4.2.2, we already discussed methods of customizing CRP. One of these methods was the so-called microcode, introduced by Delling and Werneck in [DW13]. We modified microcode to operate efficiently on a GPGPU. Most important in these modifications is to make use of the shared memory[4] as much as possible. The much larger global memory[5] requires coalesced access patterns to work efficiently. This holds true for the shared-memory as well. The effects are much worse in global memory, though. We managed to reduce the memory requirements of the contraction process by reusing fully processed entires. The moment a memory position has been read for the last time, we allow for it to be reused. As a micro-instruction usually compares the computed value to the already stored value at the target position (initially set to infinity for all values), we have to make sure this result does not influence our contraction process. For that reason, we augment the instructions by an additional bit that, if set, forces a write access. This way, we perform a reinitialization of previously used entries in the shared memory at the cost of a single bit per instruction.

Second, we also require a high degree of parallelization. Even though the described compression enables us to pack multiple different cells greedily into blocks (as long as the required working set fits into the fast memory) that we can process simultaneously, the size of the shared memory severely limits the number of cells we can process at a time. On modern GPGPUs, we require hundreds of threads to work in parallel to hide latencies effectively [Cor14]. While the original microcode is processed as a stream of instructions, it is not inherently sequential and we can find some parallelism within a cell: any two instructions that do not write to the same memory location can be executed independently of each other. We compute levels of instructions by selecting them greedily from the latest to the earliest, assigning a instruction to the highest possible level.

Bellman-Ford Customization. On CPUs, microcode has proven less efficient on higher levels of the hierarchy [DW13] than a Bellman-Ford based customization. The same holds true for GPGPUs, for which the Bellman-Ford algorithm has been shown to work efficiently [DBGO14]. In our special case, we can make use of the special structure of the (turn-aware) structure of the CRP-cells that distinguish between entry

[4]Fast but small (48 KB) memory on the GPGPU. Similar to a cache on a CPU but explicitly addressable.

[5]Slower memory, comparable to RAM.

Table 4.1: *Customization cost (time and energy consumption) on various hardware settings. The western European road network is used for an input (see Chapter 6).*

			rate		memory	time	power	energy
setup	GPU	amount	core	mem	$[MiB]$	$[ms]$	$[W]$	$[J]$
1	Titan	1	1.0	6.0	3 791	150.4	248	37.7
2	Titan	1	1.2	6.0	3 791	129.3	280	36.2
3	780 Ti	2	1.2	7.0	3 800	67.3	574	38.6
4	780 Ti	4	1.2	7.0	3 821	35.8	1 045	37.4

and exit vertices. The arcs of an entry vertex only need to be relaxed once at the beginning of the computation. The arcs of an exit vertex only need to be relaxed once at the end of the computation. The Bellman-Ford algorithm itself only runs on inner vertices. We present both a global and a local implementation of this scheme, one operating in global, one in shared memory.

A Few Results. During the customization, we only transfer a minimal amount of data to the GPGPU, a set of small tables that enable the decoding of arcs. All further data, i.e. the micro-instructions, can be transferred prior to the actual customization as part of the metric independent preprocessing. The new GPGPU-based implementation allows for immensely fast customization. In Table 4.1, we present an overview of the customization results obtained when using our tuned GPGPU customization. The transfer of the customized cells back to main memory can be hidden almost completely in an asynchronous transfer. The NVIDIA Titan offers 6 144 MiB of DDR5 RAM (6 GHz), 14 multiprocessors of 192 cores each (2688 cores in total) and operates at a core clock rate of 1 GHz, as long as it stays cool enough (837 MHz else). The EVGA GTX 780 Ti offer 15 multiprocessors, offering 2880 CUDA cores in total, each of which operates at 1.2 GHz. It has 3 GiB of RAM clocked at 7 GHz.

Our results reduce the amount of time required for customization by an order of magnitude, compared to 12 cores of an Intel Xeon X5680 (3.33 GHz, 6×64 KB L1, 6×256 KB L2 and 12 MB shared L3 cache) using DDR3-1333 RAM. In addition to the reduced computation time, the GPGPU implementation also reduces the required energy footprint of the customization to about a third of the required energy of the twelve core CPU.

Chapter 5

5 Alternative Routes - a Snapshot

Our main topic is the consideration of alternative routes to a shortest path. At this point, we present the current state of the art.

So far, we have introduced the basic concepts and a wide range of different speed-up techniques. The main topic of this thesis, however, are alternative routes. Our reasoning behind this thorough introduction of modern speed-up techniques will become clear in the following sections as most of the methods we describe require a solid understanding of speed-up techniques to be able to grasp them.

An early mention dates back to the fifties, even though it does not completely describe an alternative route. Minty [Min57] proposed an analog computer for the SPP. He states:

> Build a string model of the travel network, where knots represent cities and string lengths represent distances (or costs). Seize the knot "Los Angeles" in your left hand and the knot "Boston" in your right hand and pull them apart. If the model becomes entangled, have an assistant untie and re-tie knots until the entanglement is resolved. Eventually, one or more paths will stretch tight – they then are alternative shortest routes.[1]

Next to this rather impractical algorithm for continental scale networks, initially the main focus on alternative routes was directed at [Jac55] – but not limited to [Tru52] – communication networks.

A variety of different concepts describe methods of finding alternative routes. In a road network, the most obvious one is to directly consider different metrics. Ben Akiva et al. [BABDR84] describe an approach considering metrics such as travel time, distance, traffic signals, aesthetics or even road signs that guide the path for their

[1]This method does, of course, only computes alternative paths of identical length.

algorithm. As they already state, many of the paths might be similar. At other times, routes might even be unreasonable. An example might be a route that uses a completely unfortified road in order to safe a few meters.

The literature discusses a selection of other techniques; the following paragraphs give an overview and a short summary of these techniques.

Different Objective Functions. The most basic approach to alternative routes would be to consider a set of different metrics [BABDR84]. For example: we could consider the best travel-time, minimal distance, road quality, scenery, and many more criteria. A problem in this approach, however, is that many of the potential criteria are not independent of each other. Differences between the resulting routes might be too small to justify the overhead of supporting each of these metrics independently. CRP, however, would be actually able to support a reasonable amount of metrics at low cost.

k-Shortest Path. A, at the first glance reasonable, approach to alternative routes is the k-shortest path problem [Yen71]. The basic notion behind the problem, which has been studied quite extensively (e.g. [Shi79, Epp94, Epp98, Rup97]), is that next to the shortest path itself, slightly suboptimal paths will offer good alternatives. While this idea seems valid for specific networks[2], it has been described as less effective [BDGS11] in the context of alternative routes in road networks. It is rather unlikely to find a good alternative route among the first few hundred paths. Jumping off the highway at a ramp and directly returning back onto it does not take much time compared to the full journey. Doing this at every possible combination of ramps might already contribute a large number of possible paths that are only slightly worse than the original shortest path. This directly implies that we might need a very large value for k before we can report a reasonable alternative route.

Eliminating arcs/vertices. Among the first considerations for alternative routes is the concept of blocked paths or blocked vertices. The studies are motivated by telephone connections for which operators only had a low number of potential connections they could establish. Automatically finding connections for long range calls in the U.S.A. was an early research topic according to Jacobitti [Jac55]:

> When a telephone customer makes a long-distance call, the major problem facing the operator is how to get the call to its destination. In some cases, each toll operator has two main routes by which the call can be started towards this destination. [...] If the first-choice route is busy, the second choice is made, followed by other available choices at the operator's discretion. [...] When operator or customer toll dialing is considered, the choice of routes has to be left to a machine.

[2]Possible examples could be Directed Acyclic Graphs(DAGs) with a low number of paths or social networks in which paths are of very short length.

The process of eliminating vertices or arcs from the graph can be handled with (near infinite) penalties. As such, the elimination process can be seen as a specialized form of penalization.

Penalties. In 2011, Bader et al. [BDGS11] reiterate and discuss a somewhat related approach to the blocked vertices that goes back to de la Barra and Perez [dlBPA93]. Instead of excluding a single vertex, they apply a penalty to the arcs of the shortest path and adjoining arcs[3]. Penalizing the shortest path alone could result in a large number of small detours as described in the former paragraph on k-shortest paths. The penalization of adjoining arcs avoids this scenario. The authors of [BDGS11] describe an iterative process to compute route candidates, the union of which forms the so-called alternative graph: each of the contributing paths is a shortest path in a dynamically adjusted graph. Variations of penalization schemes as well as the chosen paths that form the alternative graph give this method many degrees of freedom.

Equilibria. Another special form of penalization could be seen in the calculation of equilibria. The computation of equilibria usually focuses on a large set of source and target pairs, the so-called demand set [LS11], distributed over a network. For the computation of alternative routes, we could interpret a query as a demand set that includes the same source and target multiple times. The initial solution to this problem, which uses free-flow traffic, would result in a large amount of demand along the shortest path. In an iterative process, arc weights are adapted based on previous demand and new routes are calculated. The process of finding a good flow through the network for the desired demand is described by Frank and Wolfe [FW56]. It is unclear, however, how well these methods behave if only a single pair of source and target is selected. One way of dealing with this could be to select multiple pairs of vertices that are close to the desired source and target.

To the best of our knowledge, not many techniques exist that use speed-up techniques to compute such equilibria. The single effort known to us is that of Luxen and Sanders who offer a method based on CH [LS11]; their method implements the Frank-Wolfe algorithm. For a fast integration, they perform a non-recursive cumulative unpacking, called *hierarchy decomposition*, in which they process the shortcuts in a preset order. During the process, counters are updated from top to bottom of the hierarchy until a final set of arc-level counters is found.

In its current form, this seems too expensive for a real-time scenario due to repeated CH creation. The approach of Luxen and Sanders should be implementable far more efficiently using using the recent customizeable CH [DSW14].

Branching methods. The branching method, as a method to approximate k-shortest paths goes back to Bellman and Kalaba [BK60]. This method computes a concatenated

[3]The adjoining arcs form a variation to [dlBPA93] who only consider penalizing the path itself.

path by first computing the shortest path between s and t and then selecting some adjacent arc (u, v) with $u \in \Pi_{s,t}$ and $v \notin \Pi_{s,t}$. The concatenation of $\Pi_{s,u}$, the arc (u, v), and $\Pi_{v,t}$ provides the desired new path. To approximate the k-shortest path problem, the found paths are kept in a list and sorted by increasing length.

Via-Nodes/Plateaux. The plateau-method or the related via-vertex approach [Cam05, LS12, ADGW13] are the currently most used approaches to compute alternative routes. The basic idea is to identify a good intermediate destination, or via-vertex, to describe a concatenated path. Abraham et al. [ADGW13] define a set of criteria the alternative route has to fulfill to present a viable candidate. We discuss these criteria in Section 5.2. The main problem of this approach is how to identify a good vertex to act as a via-vertex. The plateau-method essentially describes a way of identifying such vertices. It defines a plateau as an overlapping segment of a forwards directed shortest path tree rooted at the source and a backwards directed shortest path tree rooted at the target. Due to this overlap between the two trees (compare Figure 5.1), the transition from the source-tree to the target-tree will, in some sense, look naturally – we will discuss later what exactly we mean by this.

Pareto-Optimal Routes. When offering a routing service, one of the arguments for alternative routes is the varying preferences of different users: while one might consider the fastest route to be best, others might prefer a slower but shorter path. A natural way of handling all possible variations is the concept of Pareto-optimal shortest path (compare Section 2.3). As given in Definition 2.6, in this scenario we compute all paths which are not definitely worse than any other path. The problem can be solved via a generalization of Dijkstra's algorithm [Han80]. Despite the extensive research on speed-up techniques, only few consider Pareto-optimal shortest paths for speed-up techniques. The main focus for Pareto-optimal-optimal routes is on goal directed search [SI91, MdlC05] or approximation due to highly similar routes [DW09]. Bader et al. [BDGS11] study Pareto-optimal shortest paths as a way to compute alternative routes without employing a speed-up technique and employ similar filtering ideas as Delling and Wagner. Sanders and Mandow [SM13] present theoretical work on the parallelization of multi-objective search. Next to the direct utilization of Pareto-optimal routes, we can artificially introduce a second criterion by counting the frequency of an arc appearing as part of a previously computed alternative route. In an iterative process, we can repeatedly modify the second criterion and compute another, now hopefully different, Pareto optimal path.

A paradigm related to this approach is presented in [GKS10]. Geisberger et al. compute shortest paths for a linear combination of two metrics. This linear combination allows the discovery of the subset of Pareto-optimal-optimal paths that lie on the convex hull of all Pareto-optimal-optimal paths. The authors present not only a way

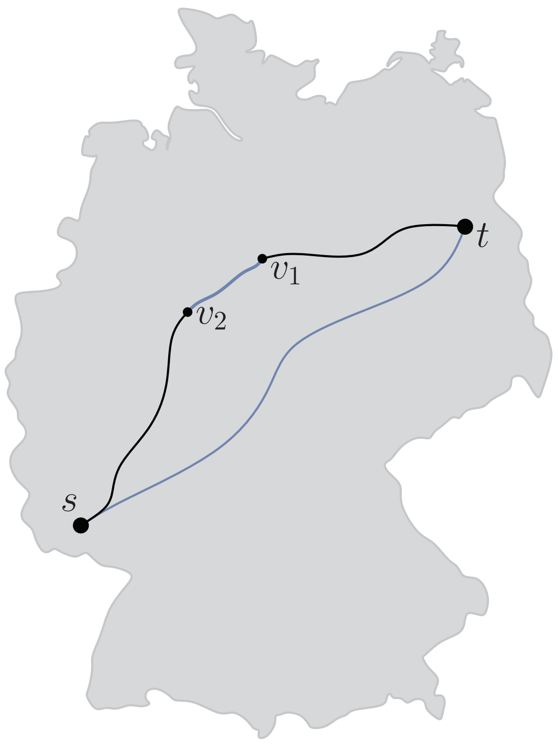


Figure 5.1: *Illustration of a plateau for the shortest paths Π_{s,v_1} and $\Pi_{v_2,t}$. The plateau is the common segment v_2, v_1 which, due to subpath optimality, is also a shortest path.*

to compute single paths for a fixed linear combination but also show how to efficiently compute the full range of distinct paths that exist.

Time-Dependence. A completely different approach could be to take varying travel time profiles (TTF) [BGSV13] into account. The notion behind this idea is that a route that is optimal at any given time of the day could also provide a general alternative path. Batz [Bat14] offers a detailed overview of current efforts to compute such time-dependent routes and even offers possibilities of computing full TTF-profiles between a given source and target. These profiles could be used to describe a set of alternative routes. Results in compression experiments indicate that the data currently available to the research community offers only little variation, though [BGL$^+$12].

Quality. The most important criterion, next to the quality[4] of the alternative routes calculated, is the time required for finding them. A technique suitable for usage in a modern navigation service has to operate on a speed-up technique in order to provide reasonable query times. In the following sections, we take a look at the more

[4]We will discuss the perception of quality in a moment

recent developments regarding alternative routes. Next to the approaches that try and approximate the k-shortest path problem, two main concepts were developed that strictly focus on alternative routes in navigational services. Both approaches discuss their own criteria on how to describe a high quality set of alternative routes. The first concept, which we discuss in Section 5.1, evaluates the full set of alternative routes as a whole. The second concept, discussed in Section 5.2, evaluates the quality of a single route in relation to previously found paths.

5.1 Alternative Graphs

As the first concept, we discuss alternative graphs, which were, to the best of our knowledge, introduced in [BDGS11]. An *alternative graph* essentially describes a compact representation of multiple paths as a whole in a human-readable form. This representation of a set of paths as a graph can encode even more choices as the routes on their own; e.g. Figure 5.2b can be calculated from two paths but contains four possible routes.

Next to the ease of use, the concept of alternative graphs allows for an evaluation of a set of paths as a whole rather than on their own, a concept that is not considered in this form elsewhere. The authors express these requirements in a set of measures that describe the graph as a whole. They do not make any assumptions on the quality of single paths that leads through the alternative graph.

Definition 5.1 (Alternative Graph Quality Criteria due to [BDGS11]). *Given an alternative graph $G^{st}\left(V^{st}, A^{st}\right)$, following indicators describe the quality of G^{st}.*

total distance $$\mathcal{D}_{tot} := \sum_{a=(u,v)\in A^{st}} \frac{\mu\left(a\right)}{\mathcal{D}_{G^{st}}\left(s,u\right) + \mu\left(a\right) + \mathcal{D}_{G^{st}}\left(v,t\right)}$$

average distance $\mathcal{D}_{avg} := \frac{\sum_{a\in A^{st}} \mu(a)}{\mathcal{D}_{G^{st}}(s,t)\cdot\mathcal{D}_{tot}}$

decision arcs $$m_\delta := \sum_{v\in V^{st}\setminus\{t\}} \delta_{out}\left(v\right) - 1$$

The total distance criterion describes in how far the different paths through G^{st} are non-overlapping. Average distance measures how much longer the average alternative path is in relation to the shortest path. Finally, the decision arcs denote the complexity of the graph presented to the user. The higher the amount of decision arcs, the easier a human might get confused when presented with G^{st}. As a target function, Bader et al. ask to maximize $\mathcal{D}_{tot} - \left(\mathcal{D}_{avg} - 1\right)$ under the constraints that $\mathcal{D}_{avg} \leq 1.1$ and $m_\delta \leq 10$.

The proposed criteria are not without problems, though. Figure 5.2 illustrates two alternative graphs that offer identical parameters for every one of the measures from Definition 5.1 yet are different enough that one might be preferable over the other.

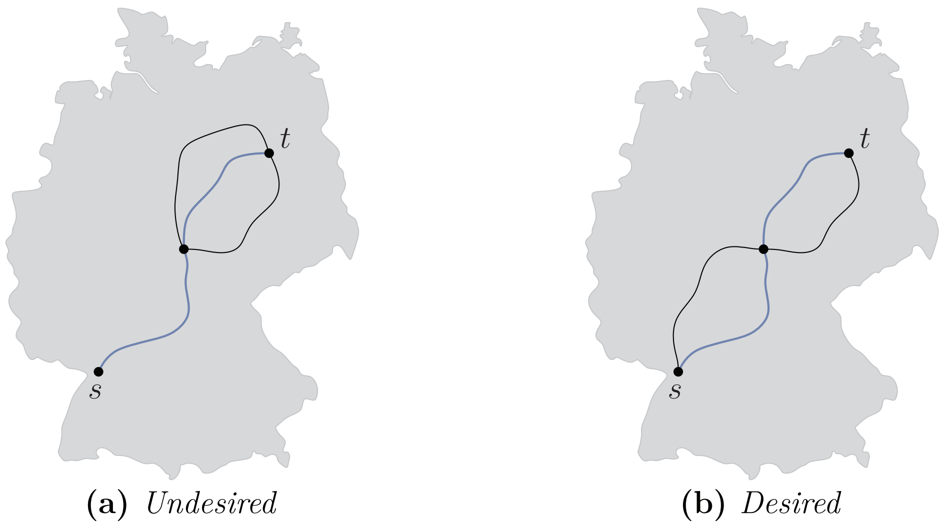


Figure 5.2: *Two graphs with identical $\mathcal{D}_{tot}$, $\mathcal{D}_{avg}$, and m_δ, of which the right one might be more desirable. Assume unit cost for all arcs.*

An additional problem of this approach is that the presence of many good paths enables the inclusion of a very bad path. Figure 5.3 gives an example of a setting in which near unlimited stretch could be achieved, of course only while ignoring the limits on the decision arcs (m_δ). We can, however, increase this effect while respecting the limits on m_δ by increasing the length of the shortest path we consider.

5.1.1 Computation

The main technique discussed by Bader et al. focuses on penalties. The general concept of penalization, which has already been studied in [CBB07], offers a wide range of possibilities due to a multitude of available implementation choices. The most drastic forms include the effective deletion of a set of arcs by applying a near infinite penalty or the deletion of single vertices, which essentially translates into the penalization of its adjoining arcs. In a far fetched translation, we can even get close to time-dependent routing by interpreting a snapshot of the time-dependent cost as a penalization with respect to the free flow cost. This interpretation, however, does not consider the changes in the TTFs that occur during the journey.

In terms of the penalty method, Bader et al. follow similar ideas to those already to be found in [dlBPA93]. [BDGS11] introduces two types of penalties: the first one is applied to the shortest path, the second one to the arcs that are directly connected to but not part of the shortest path[5]. In addition, they combine the approach with the notion of plateaux and propose a thinout mechanism. We discuss their contributions

[5]The shortest path might only be a shortest path when considering penalties that have been applied in a previous penalization step.

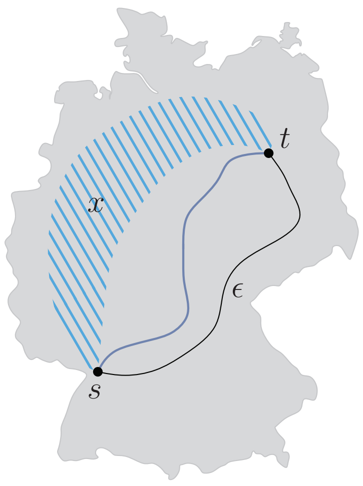


Figure 5.3: *Inclusion of a bad path into an alternative graph due to combined view. Assume unit weight for the shortest path and a set of x identically weighted paths schematically shown as the tiled area. In these conditions, ϵ can grow as large as $1.1 + \frac{x}{10}$ without violating the constraints of Definition 5.1. The number of paths for x is limited by the m_δ; the effect gets even worse if the distance between s and t gets larger, though.*

in the following paragraphs.

Penalization. As mentioned, Bader et al. consider two types of penalties. We present them in Definition 5.2.

Definition 5.2 (Penalties)**.** *Given a shortest path $\Pi_{s,t}$ in a weighted graph $G\left(V, A\right)$ between a source s and a target t, we define the following penalties:*

path-penalty $\psi := \varepsilon \cdot \mu\left(a\right),$
$\forall a \in \Pi_{s,t}, \varepsilon \in \mathbb{R}$

rejoin-penalty $\psi_r := \frac{\iota}{2} \cdot \mathcal{D}\left(s,t\right),$
$\forall a = (u,v) \in A, (u \in \Pi_{s,t} \wedge v \notin \Pi_{s,t}) \vee (u \notin \Pi_{s,t} \wedge v \in \Pi_{s,t}), \iota \in [0,\varepsilon]$

ε and ι are some penalty factors that can be used to tweak the penalties. Bader et al. also propose that limits on successive increases might be required as well. Furthermore, they say that it could be beneficial to choose ε per arc (u,v) in dependence of $\mathcal{D}\left(s,u\right)$ and $\mathcal{D}\left(v,t\right)$. This idea of varying penalties has also been discussed previously in [dlBPA93].

The main notions behind the penalties in Definition 5.2 are as follows: The path-penalty is designed to force a deviation from the shortest path. To be independent of the network representation (c.f. igure 5.4), Bader et al. propose a multiplicative factor. This factor increases every arc by a fraction of its original length. The factor should

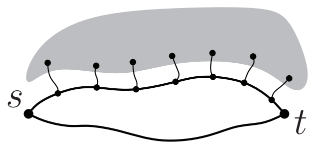


Figure 5.4: *Illustration of the effect that path geometry can have on the penalization if not done carefully. Both paths between s and t could be equivalent. If, however, we add a fixed penalty per arc, the upper path would be penalized far more strongly than the lower one.*

not be too large in order to not miss out on good paths, but large enough to avoid rediscovering the same path over and over again.

The rejoin-penalty is designed to reduce the number of short deviations along the route. Similar to the k-shortest path problem, we can consider a highway and the accompanying ramps as an example. When we penalize the highway, it might, at some point, become faster to get off the highway at a ramp and directly return to it. Such short hops are undesired and can be avoided using the rejoin-penalty.

From the sequence of resulting paths, [BDGS11] selects a set with the goal to maximize the objective function (see Definition 5.1). According to a personal conversation with the authors, we could verify that they usually compute the first twenty paths to construct the alternative graph from.

The combination of both penalties seems to provide reasonable alternative graphs already. Bader et al., however, achieve better results by combining the penalization approach with another technique.

Plateaux. The best results of [BDGS11] stem from the combination of the penalization scheme as described above and the plateau method [Cam05]. As the penalty method is based on a preexisting set of paths, it is possible to execute the classic plateau method in advance and apply penalties afterwards on the set of these routes.

Thinout. The iterative process of alternative graph construction with the help of single paths may lead to the inclusion of unnecessary information in the alternative graph. To reduce the complexity of the found graph as well as to increase its quality, Bader et al. propose two methods to thin out the graph and remove unwanted information. *Global Thinout* considers the full graph and identifies useless arcs by checking the stretch factor of the shortest path that includes the arc (for an arc (u, v) the path $\Pi_{s,u} \cdot (u, v) \cdot \Pi_{v,t}$). If the length of this concatenated path exceeds the distance between s and t by too much, the arc is removed from the alternative graph. In *Local Thinout*, only the arc (u, v) and the distance within the alternative graph between u and v is compared as to the respective difference in stretch. This way, most long and inefficient arcs can be identified.

Implementation Suggestions. All experimental data presented in [BDGS11] builds on Dijkstra's algorithm. The authors suggest that the presented algorithms might be efficiently implementable using techniques such as the dynamic variant of HNR. Similar to the observations in [LS11], we suspect that the amount of dynamic changes to the network might exceed the efficiency bounds of dynamic HNR or CH. Other dynamic approaches that came into focus lately (mostly based on Arc-Flags [DFV11, DDFV12]), also seem too inefficient for our demands. Even customizeable CH [DSW14] might be too costly for this endeavour.

Paraskevopoulos and Zaroliagis [PZ13] recently improved upon the method of Bader et al.. We discuss their changes in the following paragraphs. Our own take on an efficient implementation will be dealt with in Chapter 8.

5.1.2 Recent Improvements

Due to Paraskevopoulos and Zaroliagis, some improvements [PZ13] to the method as described by Bader et al. [BDGS11] were developed in parallel with our own contributions that we discuss in Chapter 8. The authors follow the general approach of [BDGS11] but add an additional pruning step, reducing the number of considered vertices immensely. Paraskevopoulos and Zaroliagis actually provide an implementation that offers reasonable query times, though their implementation only achieves query times of below a hundred milliseconds for rather small instances. In terms of quality, due to the concepts of Bader et al., [PZ13] manages to provide immense improvements by developing some sophisticated selection strategies that also answer some of the open questions that are not discussed in detail in [BDGS11]. As it turns out,the improvements are not directly comparable, though. The authors of [PZ13] use a severely different network model. We discuss this aspect further in Chapter 8.

Pruning according to [PZ13]. One of the benefits of the previously (compare Chapter 3) discussed ALT algorithm is that lower bounds remain valid even while applying changes to the metric, as long as the applied changes are non-decreasing. For pruning purposes, the authors restrict the search space to be used during the execution of the penalty/plateau method to vertices that will offer a reasonably short alternative route. To be exact, all vertices v with $\mathcal{D}(s,v) + \mathcal{D}(v,t) > (1+\epsilon) \cdot \mathcal{D}(s,t)$. Using the ALT algorithm with potential-function $\pi(\cdot)$, it is rather straightforward to compute all non-pruned vertices. The authors execute a normal ALT query, pruning all vertices v with $\mathcal{D}(s,v) + \pi(v,t) > (1+\epsilon) \cdot \mathcal{D}(s,t)$ in the forwards search and $\pi(s,t) + \mathcal{D}(v,t) > (1+\epsilon) \cdot \mathcal{D}(s,t)$ in the backwards search. Using the queue of the opposite direction for an exact distance estimate, only the correct vertices are kept for later processing.

An Improved Plateau Selection Strategy. For selecting good plateaux that improve the objective function according to Bader et al., [PZ13] suggests considering both the cost of the plateau path as well as the amount of sharing it has with other plateau-paths. The main difficulty in this optimization problem is that the selection of a plateau influences the quality criteria of other plateaux due to the potential overlapping sub-paths. As a result, the inclusion of a new path might reduce $\mathcal{D}_{tot}$ for other alternative candidates. [PZ13] ranks a plateau $P = (P_u, \ldots, P_v)$ by $\varrho := \frac{\mu(P)}{\mathcal{D}(s,P_u)+\mu(P)+\mathcal{D}(P_v,t)} - \frac{\mu(P)+\mathcal{D}(s,t)}{(1+\mathcal{D}_{tot})\cdot\mathcal{D}(s,t)}$. These ranks ($\varrho$) seem, according to the authors' experiments, to improve the plateau-based selection over the approach taken in [BDGS11].

Penalization Adjustments. The most important adjustment of [PZ13] seems to be the change proposed for the ψ_r. Rather then treating inbound and outbound arcs to the shortest path identically, [PZ13] proposes to penalize inbound arcs more if they are close to the source and less if they are close to the target. In an analogous fashion, outbound arcs are penalized more strongly if they are close to the target and less if they are close to the source. The authors do so by scaling the penalization linearly for every arc (u, v) by $\mathcal{D}(s, u) / \mathcal{D}(s, t)$ in case of an inbound arc and by $\mathcal{D}(v, t) / \mathcal{D}(s, t)$ for outbound arcs. This idea has already been mentioned in [BDGS11], though.

5.2 The Via-Node Paradigm

The via-vertex paradigm, or the related plateau-method, is a very intuitive form of computing alternative routes and follows the approach a lot of people would take themselves when searching for good alternatives on a map. The selection of a single vertex to describe the alternative routes is not only conceptually very simple, the method also lends itself for an implementation on top of a speed-up technique. We can argue that all necessary computations seamlessly integrate into speed-up techniques the moment a good intermediate vertex is found. Assuming one could easily name a good via-vertex v for any source and target pair, an alternative route can be described as the concatenated path $\Pi_{s,v,t}$. Not all possible concatenated paths form a reasonable alternative path, though. To the best of our knowledge, Abraham et al. [ADGW13] are the first to formally specify a set of criteria a viable alternative route has to fulfill:

5.2.1 The Concept of Viability

The notion of viability criteria is rather straightforward and can be easily motivated. Figure 5.5 gives an idea of what we desire from a good alternative to a shortest path by providing some negative examples.

According to the definitions of Abraham et al. [ADGW13], we formally describe the criteria in the following way:

Definition 5.3 (Global Viability). *Given a shortest path $\Pi_{s,t}$ in a graph $G(V, A)$ between a source s and a target t, as well as a via-vertex v: The concatenated path $P_{s,v,t}$ of $\Pi_{s,v}$ and $\Pi_{v,t}$ is a viable alternative if the following three conditions are fulfilled.*

limited sharing (LS) *The length of the common segments between $P_{s,v,t}$ and $\Pi_{s,t}$ is bounded in relation to the length of $\Pi_{s,t}$. Formally, we limit $\mathcal{L}(P_{s,v,t} \cap \Pi_{s,t}) \leq \alpha \cdot \mathcal{D}(s,t)$, for $\alpha \in [0,1]$.*

local optimality (LO) *In addition, we postulate that any segment of $P_{s,v,t}$ – not longer than a certain fraction of the shortest path – has to be optimal. We require for any $u, w \in P_{s,v,t}$ with $\mathcal{L}(\Pi_{u,w} \in P_{s,v,t}) \leq \beta \cdot \mathcal{D}(s,t)$ that $\Pi_{u,w} \in P_{s,v,t} = \Pi_{u,w}$, with $\beta \in [0,1]$. We refer to local optimality also by LO.*

uniformly bounded stretch (BS) *Finally, any segment of $P_{s,v,t}$ is bounded in relation to the distance between its first and last vertex. We limit for any $u, w \in P_{s,v,t}$ the segment $\Pi_{u,w} \in P_{s,v,t}$ between u and w on $P_{s,v,t}$ by: $\mathcal{L}(\Pi_{u,w} \in P_{s,v,t}) \leq (1+\epsilon) \cdot \mathcal{D}(u,w)$, for $\epsilon \in [0,\infty]$. We refer to bounded stretch also by BS.*

Any via-vertex that fulfills the criteria in Definition 5.3 is called an admissible or viable via-vertex.

While Definition 5.3 seems reasonable to us, it presents one major problem. The criteria themselves are costly to check. *Limited sharing* is rather easy to verify on extracted paths. The other two criteria require a shortest path query for any pair of vertices from $\Pi_{s,v} \times \Pi_{v,t}$[6].

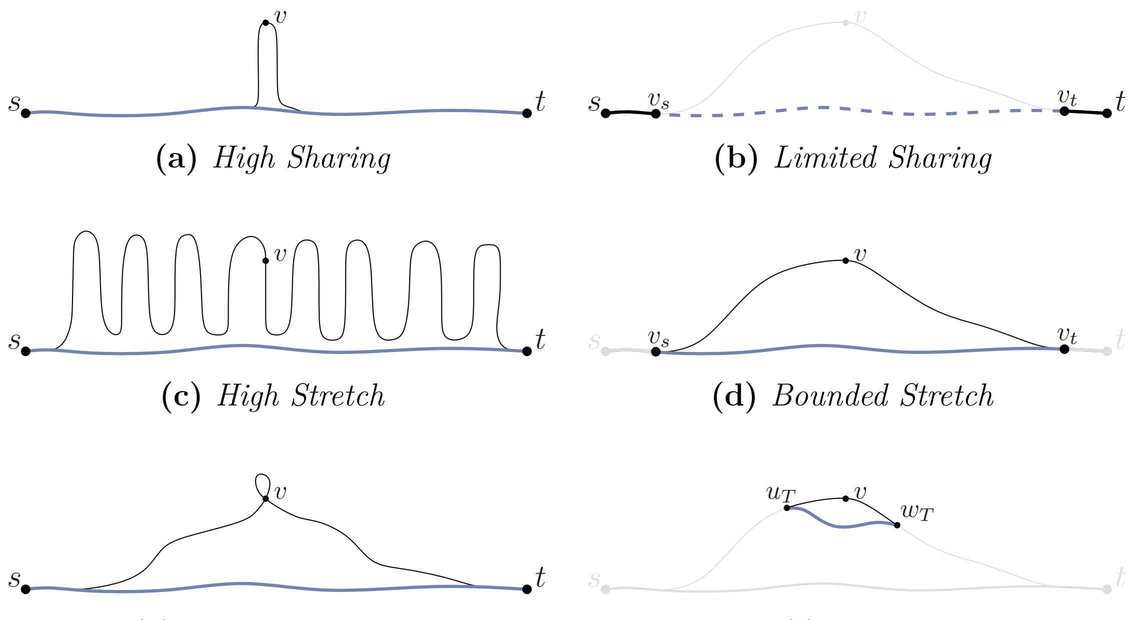
It remains unclear whether it is possible to calculate local optimality (LO) and bounded stretch (BS) in sub-quadratic time. Abraham et al. describe a way of approximating both criteria, though, thus allowing for a reasonable execution time by only requiring three shortest path queries for any potential via-vertex.

Definition 5.4 (Approximated Local Viability). *Given a shortest path $\Pi_{s,t}$ in a graph $G(V, A)$ between a source s and a target t, as well as a via-vertex v and an approximation parameter T: The concatenated path $P_{s,v,t}$ of $\Pi_{s,v}$ and $\Pi_{v,t}$ is a viable alternative if the following three conditions are fulfilled.*

limited sharing (LS) *Remains the same as in Definition 5.3: The length of the common segments between $P_{s,v,t}$ and $\Pi_{s,t}$ is bounded in relation to the length of $\Pi_{s,t}$. Formally, we limit $\mathcal{L}(P_{s,v,t} \cap \Pi_{s,t}) \leq \alpha \cdot \mathcal{D}(s,t)$, for $\alpha \in [0,1]$.*

local optimality (LO) *Let u_T be the vertex that is closest to v of all vertices on $\Pi_{s,v}$ with $\mathcal{D}(u_T, v) \geq T$. In a similar way, let w_T be the vertex that is closest to v of all vertices on $\Pi_{v,t}$ with $\mathcal{D}(v, w_T) \geq T$. We call $P_{s,v,t}$ T-LO, or in short LO, if $\mathcal{L}(\Pi_{u_T,w_T} \in P_{s,v,t}) = \mathcal{D}(u_T, w_T)$.. A viable path has to be T-LO for $T = \beta \cdot \mathcal{D}(s,t)$ for some $\beta \in [0,1]$.*

[6]Due to the concatenation of two shortest paths, the criteria are fulfilled automatically for every other pair of vertices.



(a) *High Sharing* **(b)** *Limited Sharing*

(c) *High Stretch* **(d)** *Bounded Stretch*

(e) *Localized Detour* **(f)** *T-Test*

Figure 5.5: *Motivation behind the viability criteria (a,c,e). The path should be sufficiently different (top), not too long in comparison with the shortest path (middle), and reasonable during every sufficiently short segment (bottom). The (approximated) viability (see Definition 5.4) tests that operate against these unwanted conditions are illustrated in the respective images b,d, and f.*

bounded stretch (BS) *The distinct segment of the alternative route may not be much longer than the bridged segment of the shortest path. For the vertex $u \in \Pi_{s,v} \cap \Pi_{s,t}$ with $\mathcal{D}(s,u)$ maximal and the vertex $w \in \Pi_{v,t} \cap \Pi_{s,t}$ with $\mathcal{D}(w,t)$ maximal, we limit the segment $\Pi_{u,w} \in P_{s,v,t}$ between u and w on $P_{s,v,t}$ by: $\mathcal{L}(\Pi_{u,w} \in P_{s,v,t}) \leq (1+\epsilon) \cdot \mathcal{D}(u,w)$, for $\epsilon \in [0,\infty]$.*

The test for LO with regard to T is called T-test.

From this point forward, we refer to the approximated local criteria whenever we talk of limited sharing (LS), bounded stretch (BS), and local optimality (LO). The ideas of Definition 5.4 are additionally visualized in Figure 5.5.

Abraham et al. show the T-test to offer a 2-approximation of the global local optimality criterion as well as a relation of approximated local optimality and bounded stretch to uniformly bounded stretch. According to [ADGW13], any path that passes the T-test for $T = \beta \cdot \mathcal{D}(s,t)$ and has bounded stretch (according to Definition 5.4) of $(1+\epsilon)$ is uniformly bounded in its stretch by $\frac{\beta}{\beta-\epsilon}$.[7]

[7]Compare Definition 5.3

Plateaux and the T-test. As we mentioned previously, plateaux are strongly related to the concept of via-vertexs. This relation stems from the fact that a large enough plateau induces an admissible via-vertex somewhere along the plateau. Remember that a plateau is an overlapping segment of two shortest path trees. As such, if we assume a vertex v in the center of a plateau of length $2T$, we can guarantee for a successful T-test at v. In the case of a plateau, we can introduce this vertex as an artificial via-vertex and omit the additional shortest path query required for the T-test. An implementation to compute alternative routes using plateaux is available by Cambridge Vehicle Information Technology [Cam05][8] under the name of *Choice Routing*.

5.2.2 Current Implementations

Starting with the concept of viability, Abraham et al. [ADGW13] introduce a series of different implementations that can be used to find alternative routes based on via-vertexs.

Bi-directional Dijkstra The most basic implementation discussed in [ADGW13] builds on two unidirectional executions of Dijkstra's algorithm. In the first search, starting at the source, the algorithm searches towards the target in the forward graph. The second search starts at the target and searches backwards towards the source. The searches, however, do not stop when reaching the source or target. Instead the settle all vertices within a ball of $(1+\epsilon)\mathcal{D}\left(s,t\right)$.

The resulting overlap of vertices contains all possible via-vertexs, as no viable alternative route can be longer. In their algorithm, named X-BDV, Abraham et al. check all candidates for the resulting stretch and for their performance in the T-test. Both checks can be done in constant time per vertex when first executing a single additional instance of Dijkstra's algorithm. They report the best vertex in accordance to some objective function.

As it is very expensive, the algorithm is mostly used as a reference. The query time is too large for practical use in our interactive setting, but the algorithm can be seen as a gold standard in terms of the number of alternative routes found.

Reach. A natural extension to the slow X-BDV algorithm is to prune the searches in accordance with some speed-up technique. The theoretical results [AFGW10] found in relation to these techniques suggest that speed-up techniques will rarely prune good via-vertexs. This is partly motivated in LO, which indicates the existence of a shortest path that shares little with the shortest path. Abraham et al. propose two variations, the first of which uses Reach and is referred to by X-REV: Similar to X-BDV, an algorithm based on Reach builds (now pruned) shortest path trees from s and t. The pruning has an unwanted side effect, however. Most speed-up techniques do not offer

[8]It operates using Dijkstra's algorithm, however, and is too slow for large-scale networks as a result.

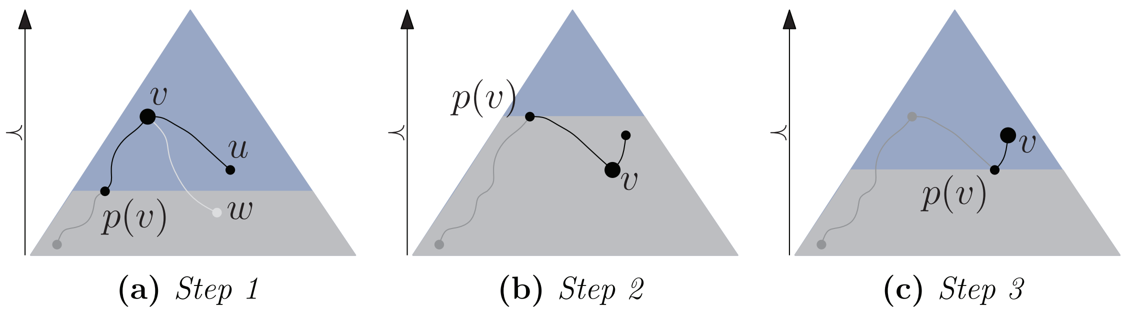


Figure 5.6: *Relaxed search variant (first order) of the CH forward search in $G^\uparrow$ at the vertex v. The vertex u (see 5.6a) is included in the search, even though $u \prec v$. w on the other hand is pruned as $w \prec p(v)$. During multiple steps of the algorithm, the level used for pruning can be raised (5.6b) and lowered (5.6c).*

any guarantees besides the fact that the pruned search tree does contain the shortest path between s and t. As a result, the path from s to any vertex contained in both shortest path trees (in the same way as the way to t from said vertex) might not represent an actual shortest path. This induces the necessity to first query $\Pi_{s,v}$ and $\Pi_{v,t}$ for any via-vertex v. In combination with the T-test, each via-vertex candidate requires three pruned shortest path queries to check its admissibility. Furthermore, the addition of shortcuts to the graph introduces a partial unpacking problem. To correctly compute the amount of sharing and to find the necessary vertices for the T-test, we have to unpack the paths. Without information about the vertices where the via-path deviates from/rejoins the shortest path and the vertices as described in Definition 5.4, we cannot check the admissibility. In a balanced hierarchy, as is usually the case in practice, these vertices can be calculated with by binary search in $\mathcal{O}(\log n)$. Again, as in X-BDV, Abraham et al. report the best via-vertex among the tested candidates, according to some rating function that considers the criteria of Definition 5.4.

Contraction Hierarchies. In its most basic form, the CH implementation (X-CHV) is again identical to the approach of X-BDV. We perform the normal forward and backward search in $G^\uparrow$ and $G^\downarrow$. Similar to X-BDV and X-REV, all common vertices of both searches form candidates for the respective via-vertex selection. Even more than with Reach, the information found in the respective CH queries does not fully reflect actual shortest path information. Again, we first have to query for $\Pi_{s,v}$ and $\Pi_{v,t}$ for any via-vertex v. In the same way as for X-REV, we have to employ partial unpacking of shortcuts to find out about the required vertices for the T-test and the deviation/rejoin vertices to the shortest path.

5.2.3 Relaxation

Pruning the search trees is a necessity to allow for queries fast enough to support any interactive service. Modern speed-up techniques have become so good in pruning vertices from the search trees, though, that we might miss out on a lot of potential via-vertexs. Abraham et al. introduce the concept of *relaxation* to alleviate the resulting problems of strong search tree pruning. To relax the search of a speed-up technique, they lessen the amount of pruned vertices. For Reach, Abraham et al. propose to multiply the reaches of all vertices by an appropriate constant. The resulting values increase the amount of unpruned vertices. For any $\delta \geq 1$, they specify the algorithm δ-REV that operates identically to X-REV but uses reaches multiplied by δ. In addition, the authors show the existence of a non-trivial δ_{BDV} that guarantees δ_{BDV}-REV to find the same solutions as X-BDV in a more efficient way.

For X-CHV, the relaxation is far more complicated than for X-REV and lacks its theoretical justification. To relax the search of a CH query, Abraham et al. propose for the algorithm to be able to look down in the hierarchy. Naturally, the amount of looking down has to be limited or we would end up executing X-BDV. The process, illustrated in Figure 5.6, operates as follows: Let $p_i(v)$ be the i-th ancestor of v in the CH search tree. In the k-relaxed variant of a CH search, an arc (v, w) is pruned if $w \prec \{v, p_1(v), \ldots, p_k(v)\}$. In the case that v has less than k ancestors, the arc (v, w) is not pruned. While the process is somewhat intuitive, it remains unclear to what effects this relaxation influences the shortest path tree in $G^{\uparrow}$or $G^{\downarrow}$. The method has proven itself experimentally, though. Still, it gets expensive very fast and the trade-off between relaxation and query performance offers diminishing returns for larger than three to four times relaxed CH searches. We refer to a k relaxation of the X-CHV by k-CHV.

A further approach to relaxation has recently been patented by Google [Gei14]. Their new approach artificially reduces the number of levels in the CH. Instead of looking at the set of k parents to decide which arcs to relax, they simply relaxation all the arcs within a level in addition to the upwards directed arcs. While this approach should find roughly the same vertices for via candidates as the previous approach by Abraham et al. [ADGW13], the process is simpler and a bit more intuitive.

Candidate Sets for Alternative Routes. A separate implementation of the CH based via-vertex paradigm is the algorithm of Luxen and Schieferdecker [LS12]. The authors operate on the conjecture that the number of viable alternative routes between two sufficiently far apart regions of a road network is limited. Therefore, they propose to compute reasonable via-vertexs in advance. [LS12] uses a multi-level partition in addition to a third partition that supports the execution of an arc-flags enhanced CH. Between pairs of cells on a given level[9] the authors compute a set of alternative routes

[9]On the coarsest level between all pairs, on the finer levels only between close pairs of cells.

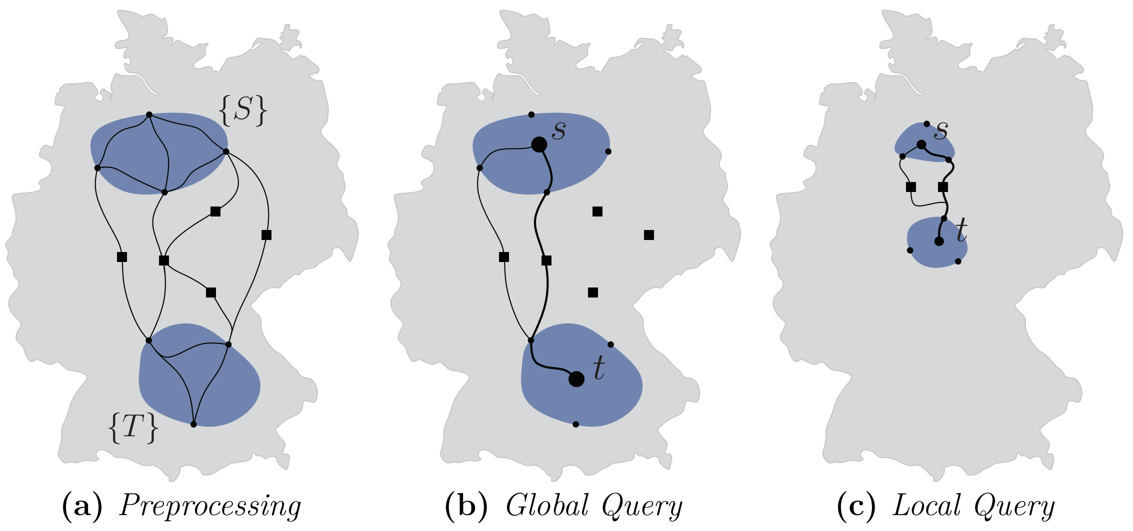


Figure 5.7: *Candidate sets for alternative routes: Between pairs of cells the set of reasonable alternative paths is calculated between their border vertices. The required via-vertexs form the candidate set (5.7a). The query checks these candidates and reports the first admissible alternative route (5.7b). Closer s,t-pairs use a fine-grained additional partition (5.7c).*

between the respective border vertices. These routes are covered with a small number of via-vertex candidates. As a result, these via-vertexs need not be in the actual search trees of the source or target.

During query time, all appropriate via-vertex candidates are checked for their approximated admissibility (see Definition 5.4). In contrast to X-CHV or the other variants from [ADGW13], Luxen and Schieferdecker simply return the first admissible path. In the case of no precomputed via-vertex candidates for a given pair of source and target, they fall back to X-CHASEV, a variant of X-CHV that uses arc-flags to further speed up the query algorithm. The main benefit of this work is an improved query performance in terms of query time. The authors manage to provide the, currently, fastest method to compute alternative routes comparable to the quality of a relaxed X-CHV variant (see Figure 5.7 for an illustration of the technique). This performance comes at a price of additional preprocessing, memory consumption and complex algorithms as the authors require an additional partitioning step to the CH preprocessing, storage for the via-vertex candidates and the respective partitions. Finally, the overhead to calculate arc-flags for the CH adds its footprint to both the preprocessing time and memory overhead. In addition, it is unclear to us how to explain the gap to X-BDV.

Chapter 6

6 Experimental Methodology

Any experiment can only become meaningful in the right setting.

This thesis focuses, as explained earlier, on experimental algorithms. A big part of this field is the evaluation of new algorithms using real-world instances. This is in contrast to the worst-case considerations of algorithm theory. In this chapter, we present our experimental setup that is used throughout the main experiments of this thesis.

6.1 Hardware and Environment

We evaluate our algorithms on various machines, each supporting a Linux based environment. The following paragraphs describe our experimental environment.

Machines. The hardware setup for our evaluation is characterized in Table 6.1. Most of our algorithms, except for the algorithms in Chapter 7 and 8, operate sequentially. These algorithms only use a single core of the machines. In parallel algorithms we specify the number of cores used in the respective section.

Environment. The compute servers used in our experiments operate on a Linux environment, running Ubuntu LTS on a 3.5 kernel. We compile our code with GCC 4.8. For flags we use optimization level three (`-O3`) and architecture specific tuning via `-mtune=native` and `-march=native`. For algorithms that use vector instructions, we utilize SSE in version 4.2 (`-msse4.2`).

The parallel execution of our algorithms is done in either of two models. The first one, a thread parallel model, is implemented in OpenMP. We use OpenMP in version 3.1 (`-fopenmp`). For the work presented in Chapter 7, we operate on a task parallel model and utilize Intel® Threading Building Blocks[1] in version 4.0.

[1] `www.threadingbuildingblocks.org`

Table 6.1: *Compute machines used in the experimental evaluation throughout this thesis. The benchmark value (bm) is measured on using the benchmark provided for [BDG+15]. Cache sizes are given in KB. The benchmark indicates the average query time of an instantiation of Dijkstra's algorithm on the road network of the United States.*

name	CPU						RAM	BM
	Model	Cores	[GHz]	L1	L2	L3		[s]
M1	X5550	2×4	2.67	8×64	4×512	8 192	DDR3-PC1333	4.7
M2	E5-2670	2×12	2.30	24×64	24×512	30 720	DDR4-PC2133	3.4

6.2 Instances

The main sources of instances used in our evaluations are twofold.

DIMACS Challenge. The most widely used instance for the evaluation of shortest path algorithms is the Western European road network. It has been provided by the PTV AG during the DIMACS implementation challenge [BFM09]. The network consists of 18 million vertices and 42.5 million arcs. Except for ferries, for which the traversal time of an arc is directly specified, the graph provides 13 road categories. Category i is assigned an average speed of $10 \cdot i \left[\frac{km}{h}\right]$. According to recent publications [DGPW13], this assignment, even though at first glance simplistic, seems to resemble commercially available data quite well.

OpenStreetMap. Even though the graph form the DIMACS challenge offer a publicly available and realistic source, the size of the graph is limited. Sadly, larger graphs from the challenge are missing important road segments [ADGW13].

In recent years, another source for road networks came into existence. The community project OpenStreetMap[2] is building a crowd-sourced database of geographic information. By uploading their own Global Positioning System (GPS)-tracks, anyone can contribute to this source and improve it. The database offers a wide range of different classifications for road segments; as a results, it seems to provide a far more detailed picture than the DIMACS graphs in some regions. We build a graph from the database extracts provided by GeoFabrik[3] as of August 11th, 2014. The files contain all OSM items up to the 10th of August 2014, 20:02 UTC+0.

To enable others to extract the same input graph that we used, we describe our extraction process in detail: For this process, we extract all arcs that are part of the road network, discarding those only available for pedestrians. We calculate the length

[2]`www.openstreetmap.org`
[3]`download.geofabrik.de`

Table 6.2: *Assumed average travel speed values for the different speed-categories used in our conversion from the OpenStreetMap format. The assumed average speed values were taken from ProjectOSRM.*

road type	avg. speed	road type	avg. speed	road type	avg. speed
motorway	90	primary_link	60	unclassified	25
motorway link	75	secondary	55	residential	25
trunk	85	secondary_link	50	road	25
trunk_link	75	tertiary	40	living_street	10
primary	65	tertiary_link	30	service	10

Table 6.3: *Overview of the used instances in our experiments. All numbers specified are multiples of a thousand items. The Karlsruhe instance does not only cover the city of Karlsruhe but the whole administrative district.*

source	ID	name	$\|V\|$	$\|A\|$	Dijkstra		
					$\|V\|$[%]	decr.[%]	$\|A\|$[%]
PTV	1	Western Europe	18010.2	42560.3	50.7	3.8	51.1
PTV	2	Germany-DIMACS	4375.4	10736.2	49.3	4.2	49.4
OSM	3	Europe	30110.7	72309.2	49.6	3.7	49.2
OSM	4	Germany	4925.5	11639.9	50.1	3.6	50.1
OSM	5	Netherlands	823.5	1986.8	50.5	4.1	50.5
OSM	6	Baden-Wuertt.	677.7	1617.7	49.9	3.7	49.9
OSM	7	Adm. Karlsruhe	160.2	382.2	48.8	3.7	48.9
OSM	8	Berlin	60.7	148.3	50.8	4.6	50.6
OSM	9	Luxembourg	30.5	71.5	50.5	3.6	50.5

of every arc before we combine paths of degree-two vertices into a single arc, stripping the network of its exact geographical representation. From the induced graph, we take the largest SCC and estimate the required time to traverse an arc via assumed average speed values based on the provided road classes. We use the average traveling speeds specified in Table 6.2 for our calculation of the expected traveling time along a road segment. The values are chosen with respect to the values used in ProjectOSRM[4]; ProjectOSRM is a community supported navigation service for OSM data.

In combination, we get a reasonable set of test instances, covering a wide spread of different graph sizes. Table 6.3 summarizes the instances and provides an overview of their most important features.

[4]`project-osrm.org`

Model. All algorithms presented here are based on a simple model of the road network without any turn cost or restrictions. This model has been, for the most part, the standard model considered by algorithms for the SPP. For this reason, we do not extract any information on turns from the OpenStreetMap files. This decision is also justified in the fact that turn-restrictions modeled in OpenStreetMap have, for a long time, not been in the focus of the contributors. Mostly aimed at visualization, only the recent availability of services like ProjectOSRM made the lack of restrictions apparent to the contributors of OpenStreetMap and they only just begun adding the restrictions to the dataset.

6.3 Queries

For our experiments, we concern ourselves with a range of different types of queries. Each one of these query types offers a special kind of insight into the quality of our algorithm; the different sets consist of a thousand different requests per type. Depending on the goal of the respective table/figure, we may choose to present the average value, the mean, minimum, maximum, or specific quartiles with respect to these queries.

Random Queries. Random source and target pairs represent one of the most used type of queries. For this type of query, we select a source vertex s of a graph $G\left(V, A\right)$ uniformly at random. The target for the query is, again, selected uniformly at random from $V \setminus s$. A major drawback of the random queries is that they mostly describe longer range queries. In general, we expect far shorter queries to be the typical load of a route planning service. The numbers based on random queries can be interpreted as a kind of worst case. We mostly present results on random queries to supply a valid comparison with the wide range of available publications.

The complexity of such a random query can also be seen in the performance of Dijkstra's algorithm as presented in Figure 6.3. In general, a random query settles around 50 % of the vertices in a graph. The `decreaseKey` operations only play a minor role

Random Short-Range Queries. In addition to the general random queries, we also present some numbers on a set of short-ranged random queries. For these queries, the targets is not more than three hundred kilometers away from the source. These queries are mostly used in Chapter 8 to compare to [PZ13] in which such queries are used for their measurements. For comparison with our usual Dijkstra rank queries, we give the average Dijkstra rank of this input set in Table 6.5. The limit, as can be seen in comparison with Table 6.4, affects mostly queries on the European network. A random query on all other networks used in this thesis is usually within or close to the bound of three hundred kilometers.

Table 6.4: *Characterization of random input sets used on the different graphs.*

source	ID	name	Path		
			hops	time [min]	km
PTV	1	Western Europe	1 348	680	1 253
	2	Germany-DIMACS	464	207	387
OSM	3	Europe	1 554	1 180	1 793
	4	Germany	429	268	390
	5	Netherlands	159	79	134
	6	Baden-Wuertt.	258	99	132
	7	Adm. Karlsruhe	173	53	66
	8	Berlin	124	21	18
	9	Luxembourg	100	27	33

Table 6.5: *Average Dijkstra rank of random input sets, limited to a maximum of three hundred kilometers between source and target (distance metric).*

source	ID	name	log avg rank
PTV	1	Western Europe	18.59
	2	Germany-DIMACS	19.28
OSM	3	Europe	18.34
	4	Germany	19.39
	5	Netherlands	18.18
	6	Baden-Wuertt.	18.01
	7	Adm. Karlsruhe	15.86
	8	Berlin	14.46
	9	Luxembourg	13.40

Dijkstra Rank. The Dijkstra rank tries to overcome the drawbacks of the random queries. First used by Sanders and Schultes [SS05], these queries express the difficulty of a shortest-path query in relation do Dijkstras' algorithm. Given a random source vertex s, the Dijkstra rank of a vertex v is specified as the iteration of Dijkstra's algorithm that settles it, starting at s. For our Dijkstra rank plots, we again choose 1 000 different vertices uniformly at random. For each of these, we execute an instance of Dijkstra's algorithm and remember the vertices of rank 2^i for all i such that $2^i \leq n$. This approach gives us a thousand different source and target pairs per Dijkstra rank. The Dijkstra rank itself is only limited by the number of vertices in the graph. It is not to be confused with the diameter.

Comparing Dijkstra Ranks. The Dijkstra ranks have to be treated carefully, though, when comparing different graphs. Figure 6.1 illustrates a large relative difference

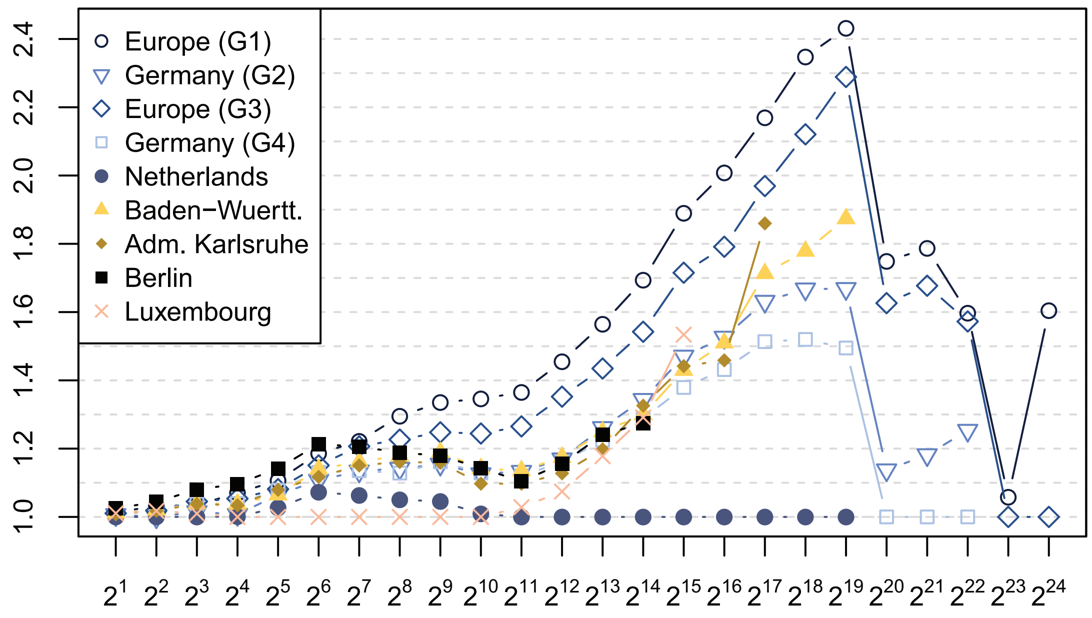


Figure 6.1: *Relative variation in length (hops) on different input graphs.*

between the median hop count and the average hop-count found in the specific settings. On a fixed graph, the complexity of different Dijkstra ranks gives a good impression of the behavior of different ranges. When comparing two Dijkstra ranks on different graphs, we might deal with different complexities. The plot justifies the use of Dijkstra ranks with respect to both the complexity of the query and the length of shortest paths, though.

6.4 Specialized Inputs

Next to the classic alternative routes on road networks, we also consider problems that require their own specific inputs. We describe these inputs in the following sections.

6.4.1 Bi-Criteria Inputs

In Chapter 7, we present an algorithm that speeds up the general problem of Pareto-optimal paths. Our solutions to this problem are of a more general relevance than the pure alternative routes considerations. To compare ourselves to available publications, we evaluate our implementation on input data provided by other researchers. We use three sources of inputs:

Table 6.6: *Characterization of the road instances provided by Ehrgott, Machuca, Mandow, and Raith; we specify the size of the graph in its number of vertices and arcs as well as the complexity of the bi-criterion problem in the form of the Pareto-optimal labels.*

instance	state	$\|V\|$	$\|A\|$	labels	
				avg.	max
DC1-DC9	Washington, DC	9 559	39 377	3.33	7
RI1-RI9	Rhode Island	53 658	192 084	9.44	22
NJ1-NJ9	New Jersey	330 386	1 202 458	10.44	21
NY1-NY20	New York City	264 346	730 100	2 082.60	7 397

Road Networks. The set of road networks used in our studies was provided by Enrique Machuca as well as Andrea Raith and Matthias Ehrgott. Raith and Ehrgott [RE09] studied a set of three US road networks with highly correlated objectives (Pearson's correlation coefficient: 0.99) in the form of travel time and distance. For each of the three graphs (Washington DC (DC), Rhode Island (RI), and New Jersey (NJ)) they select nine different source vertices to generate specific instances.

Machuca and Mandow [MM12] operate on an uncorrelated (Pearson's coefficient: 0.16) version of the New York City (NY) road map. Next to the travel time, they consider a model of the economic cost. In this model, they try and approximate the actual economic cost of traversing an arc in the form of fuel consumption and highway tolls. Similar to Raith and Ehrgott, Machuca and Mandow select a series of source vertices to generate a set of twenty instances.

Grid Graphs. A further kind of graph often used as input in considerations of Pareto-optimal paths are grid graphs. Grid graphs offer distinct advantages for testing algorithms to solve the multi-criteria search problem.

The correlation is a good indicator of the complexity of the multi-criteria-search. A feature most interesting for our evaluations is that we can easily generate metrics for any kind of correlation coefficient; this enables us to test a wide range of of problems. We can easily create anything from between small and easy and large and difficult instances. In addition, grid graphs do not differ too much from road networks. Road networks are also mostly planar and crossings with four connected arcs are their most common feature.

The method we use to achieve a specific kind of correlation between two objectives has been described in [MMO91]. We can generate a metric μ_q for any correlation coefficient q, starting at a random metric: For a grid graph with assigned metric $\mu(a) := (x, y), a \in A$, both x and y chosen uniformly at random from an interval $[1, c_{\max}]$, we generate a new μ_q, via the following assignment:

$q \geq 0$: $x' := x$ and $y' := q \cdot x + (1 - q) \cdot y$

$q < 0$: $x' := x$ and $y' := 1 + c_{\max} - (|q| \cdot x + (1 - |q|) \cdot y)$

This variation allows for any correlation coefficient $q \in [-1, 1]$ on arbitrarily large grid graphs. In the literature, $c_{\max}$ is usually chosen as ten. To study the effect of larger label counts, we also provide some experiments using $c_{\max} = 1000$.

Sensor Networks. As a final kind of input used in Chapter 7, we generated a series of artificial sensor networks. The intent behind the inclusion of this additional kind of artificial networks, next to the grid graphs, is to study influences of higher variation in vertex degrees on our algorithm. Dennis Schieferdecker kindly provided us with a series of sensor networks based on a variation of unit disc graphs. The exact generation process can be found in his PhD thesis [Sch14]. To put it simply, the generation procedure randomly places vertices in a plane with a pattern indicating some holes, applying a Unit Disc Graph (UDG) model to create the necessary communication links, or arcs. We studied a series of graphs with 100 000 vertices each, varying degree between five and fifty, and hop distance as well as an uncorrelated integer of the range $[1, 10\,000]$ as metric.

6.4.2 Stochastic Travel Times

In Chapter 13 we discuss yet a different problem. Before, we were only concerned with a static metric or the combination of a few static metrics. For our final contribution, we take a look at probability distributions. For the exact problem definition turn to Chapter 13. At this point, it shall suffice to get a general idea of the problem.

In the previous settings, all metrics are expressed as a constant. This view is, of course, a simplification of the real conditions found in a road network. We know of two different approaches to lessen this simplification to find potentially better shortest paths. One approach is the previously discussed method of time-dependent shortest paths. The second approach is a probabilistic setting in which the experience cost at an arc is subject to a probability distribution. This probabilistic setting provides a series of challenges like finding a robust path, a path that maximizes the chance to arrive on time, or a full driving strategy. In this thesis, we discuss a new approach for the latter and evaluate it on the following input sets.

To test our approach, we consider two different types of inputs, each with its own style of probability distributions. Despite many studies arguing for log-normal or gamma distributions (e.g. [HL74, RESAD06, CYGW07], many approaches choose normal distributions (e.g. [LSTM03]) for their simplicity.

Mobile Millenium Framework. The university of Berkeley provides data due to the Mobile Millenium Project (http://traffic.berkeley.edu). The project collects probe data

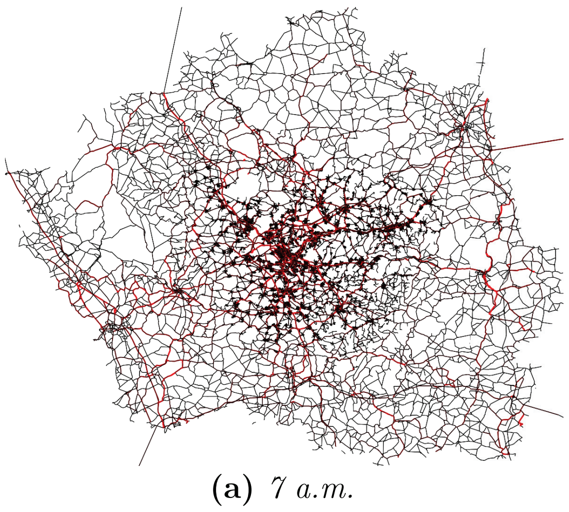

(a) *7 a.m.* (b) *7 p.m.*

Figure 6.2: *Congestions during certain times of the day in the city of Stuttgart, according to our traffic simulation data.*

from various GPS sources and processes the resulting traces via filtering and estimation techniques [HAB12] to obtain travel time distributions. These distributions are kept at a granularity of road segments and are represented in the form of Gaussian mixtures. Even though Gaussian mixtures have been argued to poorly fit actual travel times [HL74], we perform experiments on the given data; we apply filters to remove some extreme values that seem to be the result of measuring errors. These experiments not only prove some kind of robustness to the distribution type, we also are directly comparable to other publications that use said data set.

Traffic Simulation. For the generation of Gamma distributions, we consider an input based on traffic simulation [KMVP14]. The simulation is based on a detailed model of Stuttgart, including public transportation as well as a detailed demand model. The model was used to generate travel times for a full day in intervals of an hour, based on the road capacities and the expected demands. From the resulting travel times, we constructed Gamma distributions for road penalties. We assign the minimum possible travel time to an arc and add a probability distribution to result in the same mean and variation we find in the simulated travel time; e.g. for 24 values $t_1, \ldots, t_{24}$ we create an arc from $t_{\min} = \min(t_1, \ldots, t_{24})$ and assign a gamma distribution that has a mean and variation that resembles $t_i - t_{\min}, i \in [1, 24]$. In Figure 6.2 we give an impression of the amount of congestion during two different times along the day.

6.4.3 Arc Expanded Graphs

Finally, we also created a set of arc-based representations of our inputs. The reasoning behind this is given in the algorithms presented in Chapter 11. In this chapter, we focus on a set of algorithms that could benefit severely from simple u-turns. To combat any concerns, we provide some experiments on the arc-based graphs for which we set a u-turn penalty of 100 seconds, essentially forbidding u-turns. Remember that vertex-based techniques can, in some sense, directly function on arc-based graphs. The interpretation of the graph differs, however. See Section 2.1 for a description.

The arc-expanded graph of Western-Europe (graph number one) that we use in our tests has 42.5 million vertices and 191.1 million arcs. We refer to it as graph $1e$.

Acknowledgements

The used input instances within this thesis are the result of the work of many people. At this point, we would like to thank all those that provided the input data for this thesis and the accompanying papers. We would like to give thanks to the PTV AG for supplying the network of Western Europe for the DIMACS challenge. Also, we like to thank the many contributors to the OpenStreetMap project. Finally, we owe gratitude to all the individuals that supported us with specific graphs: Dennis Schieferdecker for the generation of the sensor networks, Enrique Machuca, Andrea Raith, and Matthias Ehrgott for supplying their multi-criteria networks as well as Samitha Samaranayake for supplying us with the Mobile Millenium graphs and Peter Vortisch as well as Nicolai Mallig for generating the time-dependent graphs used in Chapter 13.

6.5 Comparing Alternative Route Quality

The approaches presented in the following chapters differ greatly and even aim to solve completely different problems. Sadly, we cannot offer a definite decision on what the best method to compute alternative routes is. The quality of an alternative route, other than that of a shortest path, cannot be specified easily and is, for the most part, purely subjective. A wide range of different influences can make many different routes optimal in some sense.

Nevertheless, some criteria can be quantified for all techniques. The most important is the success rate; it specifies in how many cases a viable alternative route can be found on average. Most publications calculate the success rate for the first, second, and third alternative routes [ADGW13, LS12].

For the sake of comparison to other authors [ADGW13, LS12, BDGS11], we present typically used measures in the form of success rates for first, second, and third alternatives, their average quality in terms of LO, limited sharing, BS, and values for $\mathcal{D}_{tot}$ and $\mathcal{D}_{avg}$.

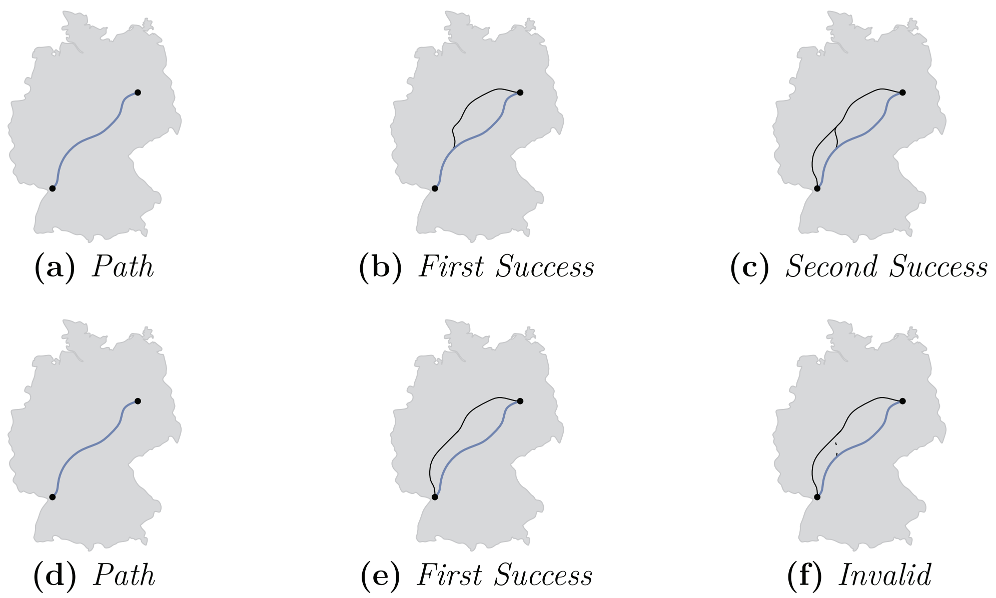


Figure 6.3: *Influence of the alternative route discovery order on the success rate for higher degree alternatives: Both sequences show the same three paths. The first sequence offers successfully discovered first and second alternative routes. The second sequence shows a higher quality first alternative route but also a failed attempt for the discovery of a second alternative route, due to high overlap.*

The usually taken approach, however, allows to sacrifice alternative route quality for better success rates. It can be beneficial to select routes that have a higher initial overlap to still be able later on to find enough new information for a second alternative. This setting is visualized in Figure 6.3. This strong influence of the overlap by previous routes in the discovery of higher degree alternatives brings us to also offering some additional insight into the techniques presented here. We present the number of first alternatives we can find (# *alt*) in our plots to show how many different alternative routes can be found. The measures are described in full in the upcoming section.

6.6 Quality Measures

The following paragraphs give a short insight into the measures we use and their potential evidence.

6.6.1 Classic Measures

For many of the techniques presented in Chapter 5, there are currently no statements on the quality of the calculated routes to be found. What springs to mind, though, is that different approaches come with different quality criteria. In [BDGS11], the authors focus on an objective function that combines the values $\mathcal{D}_{tot}$ and $\mathcal{D}_{avg}$ into a single value. The other extreme is found in [ADGW13] and [LS12]: here, the authors focus on the quality of a single route and the accompanying success rates during the calculation but barely consider the interaction between different routes.

Alternative Graph Measures. The objective function described by Bader et al. ($\mathcal{D}_{tot} - \alpha\,(\mathcal{D}_{avg} - 1)$) does provide some insight into the distinctness of different paths through the alternative graph. The influence of the average distance, however, is minimal. Due to the limit of $1.0 < \mathcal{D}_{avg} < 1.1$, the main objective function basically only considers $\mathcal{D}_{tot}$, usually setting $\alpha := 1$. Previously, we have already shown that two graphs of identical values for $\mathcal{D}_{tot}$ can represent alternative graphs of entirely different quality. Taken on their own, both measures still give some insight into the quality of an entire set of routes.

Alternative Route Measures. The alternative route measures compare a single alternative route to the shortest path as to stretch and its local quality. Limited sharing is compared to all previously found paths. Both stretch and local optimality offer a good insight into the quality of a single path, at least with respect to the still important feature of travel time. The amount of overlap to the previous alternative graph has to be treated carefully. While obviously reasonable for the first alternative route, second or third alternative routes may only exist in cases where initially a route with high overlap has been found. The result would be a direct interaction between the success rates and the general quality of alternative routes of higher degree. The success rate, as a result, is mostly meaningful for the first alternative path. Second or even third alternative routes might suffer severely from prior alternative routes that are mostly distinct from the previously found set of paths.

6.6.2 An Additional Measure

Next to these usually discussed measures, we also focus on two additional measures for alternative routes. We also take a look at the number of potentially viable alternative routes that can be chosen for the first alternative route. This number has to be treated carefully, though, as different viable routes might only differ very little. For the comparison, we only take a look at routes that offer at least a single variation to any other route found. Among all the routes, we report the best possible values for local optimality (LO), limited sharing, and bounded stretch (BS).

In addition to the parameters of Definition 5.3, we consider the relative length of the segment with the highest BS. In our opinion, the local or global character of the BS is highly relevant to its conspicuousness during the traversal of a route. This influence is not equivalent to the LO. While a minimally increased length for a segment may result in the loss of LO, the difference might not even be noticeable for a driver and mainly be due to an overly accurate model. To indicate the difference between a localized value for BS and the global character, we additionally present values for the relative length of the segment that provides the highest BS. We refer to this measure as ϵ-base to indicate the length of the segment that the values stems from.

Part II

Contributions

Chapter 7

7 Parallel Pareto Optimal Routes

The Multi-criteria shortest path problem is not only relevant to alternative routes but also a component to many approaches in multi-modal routing scenarios (e.g. [DDP+13]). This chapter presents required data structures – which can be of use on their own – and an efficient implementation of a parallel algorithm to solve the bi-criteria shortest path problem. This chapter is the result of collaboration with Stephan Erb during his masters thesis. We augmented the B-tree code of Timo Bingmann to support bulk operations. The implementation used in our experiments was created by Stephan Erb under my supervision and in an extensive collaboration. The presented numbers are new measures based re-evaluated and extended experiments. In addition to the code of Stephan Erb, Valentin Buchhold implemented the code to enable single-dimension Δ-stepping under my supervision.

For our first contribution, we take a closer look at the computation of Pareto-optimal shortest paths. Remember Definition 2.6: A Pareto-optimal path is characterized by a non-dominated label, a label to which no other path exists that is better or equal to it in every aspect. In this chapter, we consider the bi-criteria shortest path problem, in which two metrics are considered at the same time, e.g. travel time and economic cost. Our main contribution to this field is an efficient implementation of an algorithm presented by Sanders and Mandow [SM13] that, to the best of our knowledge, first solved the Pareto-optimal shortest path problem precisely and in parallel. Prior to their approach, we only know of heuristic parallel approaches based on weighted sums [Son06, SB10]. In the upcoming sections, we first introduce the algorithm of Sanders and Mandow in detail before we go on and introduce our contributions that enable an efficient parallel algorithm for the bi-criteria shortest path problem.

7.1 Multi-Criteria Search

In simplified terms, the algorithm of Sanders and Mandow, named *paPaSearch*, can be seen as an extension to Dijkstra's algorithm. In the case of a single considered metric, both algorithms are identical to each other. The main observation of Sanders and Mandow is that one might not only consider a single Pareto-optimal label (compare Algorithm 2.2) at a time but rather all Pareto-optimal labels simultaneously; this observation also marks the main difference to Dijkstra's algorithm in higher dimensions (considering multiple metrics). Remember the basic principle behind Dijkstras' algorithm: the idea that at any point during the execution of the algorithm, due to positive arc cost the currently minimal distance label cannot be improved upon. Similar to this concept, a Pareto-optimal label, optimal among all labels, cannot be dominated if all of the metric's components are of positive cost. We consider globally optimal labels at this point, not only locally optimal labels.

Not only for *paPaSearch* but in general, multi-criteria search is an extension to Dijkstra's algorithm in which we consider vertex-labels rather than vertices themselves. To be exact, Dijkstra's algorithm can be easily re-formulated in a way that we also consider the minimal label instead of the minimal vertex. Classical algorithms to the multi-criteria shortest path problem follow this approach by defining a total order on the labels [Han80, Mar84, TTLC92]. The order has to provide the guarantee that a minimal label with respect to the order is also globally Pareto-optimal. In that case, the algorithm is work-optimal, i.e. it only operates on labels that are part of the output. Nevertheless, temporarily generated labels – labels that will be dominated later on in the execution of the algorithm – differ dependent on the chosen order and orders may vary in their computational overhead as a result. Typical orders include positive linear combinations ($x^T \cdot \ell$, with $x_i > 0\,,\forall i \in [0, d-1]$) or lexicographic orderings. Reducing the order of the labels to a single-dimensional ordering destroys valuable information in comparison to full-fledged labels: the projection guarantees that the minimal label in the order is also Pareto-optimal among all remaining labels. However, we can only guarantee this for a minimal label. Sanders and Mandow essentially combine all possible orders in their algorithm [SM13] and process all possible labels simultaneously.

Goal Directed Search. Next to the different ordering strategies [Mar84, TTLC92, PS13], goal direction has been considered to speed up the one-to-one Pareto-optimal SPP [MdlC10]. The consistent vertex potentials haven proven to be efficient in the recent study of Machuca et al. [MMdlCRS12]. The reconsideration of the multi-objective variant of A* due to Stewart and White [SI91] brings multi-objective search closer to the properties of A*; in particular, the authors claim that better informed consistent heuristics always result in an equally or more efficient search. [SM13] presents a simple way of parallelizing NAMOA*, the algorithm described in [MdlC10] as an extension of [MdlC05]. In the following parts of this thesis, we only cover the one-to-all case of the bi-criterion SPP, though.

7.2 paPaSearch

The most basic form of papaSearch is nearly identical to Algorithm 2.2. We specify it in Algorithm 7.1. On a high level, only little modification is required to obtain papaSearch from Dijkstra's algorithm. The details, however, require far more attention when describing their algorithm to the fullest. In the following paragraphs, we take a closer look at Sanders and Mandow's algorithm for parallel multi-criteria search.

Algorithm 7.1 Parallel Pareto Search

Input: A weighted graph $G(V, A)$, a d-dimensional metric μ, a source vertex (s)
Output: The shortest path between s and all other vertices in G

```
1: $\mathcal{L}[] = \{\emptyset\}^n$                                    ▷ Initialize tentative distance
2: $\mathcal{L}[s] = 0^d$
3: $\mathcal{Q} = \left\{\left(s, 0^d\right)\right\}$                   ▷ Initialize the Pareto-queue
4: while $\mathcal{Q}.size() > 0$ do
5:     $L = \mathcal{Q}.extractAllParetoOptima()$                        ▷ Settle Pareto-optimal labels
6:     for all $(v, \ell) \in L$ and arcs $a := (v, w) \in A$ do
7:         $\ell' := \ell + \mu(a)$
8:         if $\ell'$ not dominated by any label of $\mathcal{L}[w]$ then
9:             $\mathcal{L}[w] = \mathcal{L}[w] \cup \ell'$              ▷ compare unreached vertices
10:            $\mathcal{Q}.push(w, \ell')$
11:        end if
12:        for all $\widehat{\ell} \in \mathcal{L}[w]$ with $\ell'$ dominates $\widehat{\ell}$ do   ▷ compare decreaseKey
13:            $\mathcal{L}[w] = \mathcal{L}[w] \setminus \widehat{\ell}$
14:            $\mathcal{Q}.remove(w, \widehat{\ell})$
15:        end for
16:    end for
17: end while
18: return $\mathcal{L}[\cdot]$
```

The complexity of the algorithm is hidden in the details: Sanders and Mandow describe 6 steps that make up the actual algorithm. In general, the algorithm maintains two kinds of data structures: a Pareto-queue [1] and for each vertex a label-set, holding all the currently known Pareto-optimal labels of the vertex. The different steps of the algorithm are:

Definition 7.1.

[1] A variation of a priority queue which we will describe later on.

1 - Pareto Queue: The set L of Pareto-optimal labels has to be identified and removed from the Pareto-queue $\mathcal{Q}$.

2 - Generating Candidates: For every label-vertex-pair $(v, \ell) \in L$ we have to scan all arcs adjacent to v. For every arc (v, w) we generate a new label-vertex-pair $(w, \ell + \mu((v, w))$.

3 - Grouping Candidates: Due to the relaxation of multiple labels at once, we might generate multiple candidate labels for a single vertex. To group the candidate labels, we sort the labels by their respective vertex-id.

4 - Pareto Optima: The different candidate labels need not be Pareto-optimal labels, even when considered on their own. As we expect a small number of candidates and a large number of labels at the vertices, [SM13] proposes to filter the label candidates prior to any interaction with the label-sets of the respective vertices.

5 - Merging Labels: The labels in the label-set of a vertex and the new candidates for that vertex are merged. Again, dominated labels have to be removed and newly found Pareto-optimal labels are stored in the label-set.

6 - Bulk Update: To complete the process of a single round of paPaSearch, the changes to the different label-sets have to be reflected in $\mathcal{Q}$ as well. Newly dominated labels are removed and surviving label-candidates are inserted into the Pareto-queue.

The workload of paPaSearch is distributed over two general classes of operations. We consider *global* operations and operations that are *local* to vertices. The global operations are performed on a newly introduced data structure: the Pareto-queue. The local operations only consider labels or label-candidates at a specific vertex. First, we present a closer look at the global operations before describing the local operations in detail.

7.2.1 A Parallel Pareto Queue

For a sequential multi-criteria search, we already talked about how we can order the different labels to achieve a label-setting algorithm. These algorithms can use the order to operate purely on a classical priority queue to obtain the next label to process. For the Pareto-queue, Sanders and Mandow propose a *balanced* binary search tree.

A binary search tree is a graph that consists of vertices called nodes. At the top of the tree is a special node called the root of the search tree, the only node that is no descendant to any other node. The nodes at the bottom of the search tree do not have any further descendants and are called leaves. The number of descendants of every node depends on the kind of tree we use. In a binary search tree, every inner node (all nodes but the leaves) has two descendants. If the search tree is balanced, the difference in height of the sub-trees to the left and to the right of a given node cannot exceed some predefined bounds. In a search tree, we also require an order to be defined on the contained elements. All elements to the left of a given node are smaller or equal to a key (also referenced as splitter key)[2] associated with it, the elements to the right of it are larger. An example for such a tree is the red-black tree introduced in [GS78]. A balanced search tree has a height of $\lceil \log n \rceil$, meaning that the longest path from the root to a leaf contains at most $\mathcal{O}(\log n)$ vertices. For the Pareto-queue, [SM13] proposes to store labels in the leaves of a balanced binary search tree that orders the different labels lexicographically by increasing x, y-coordinate and finally by vertex-id.

Locate Pareto Minima. Only very little modification is required to utilize a red-black tree structure for the purpose we require. For every inner node v, we store the leaf with the minimal y coordinate that occurs in the subtree rooted at v (the leftmost one is chosen to break ties). This slightly modified tree already allows for the parallel extraction process of paPaSearch; we describe it in Algorithm 7.2. Due to the lexicographic order, in combination with knowledge on the minimal y coordinate in every subtree, we can efficiently search the tree and only descend into the direction of non-dominated labels.

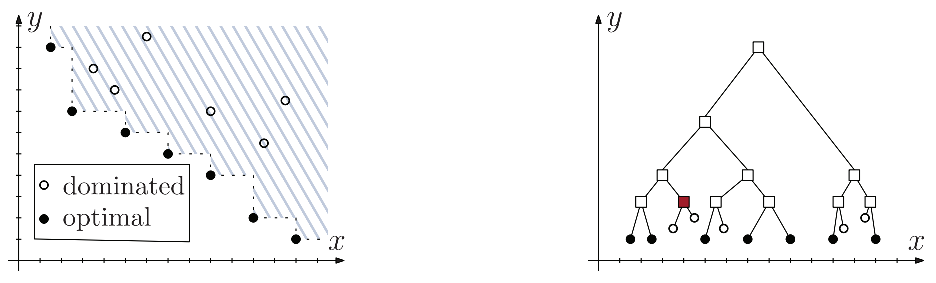
Algorithm 7.2 findParetoMinima

Input: A search tree node v, a domination value u
Output: All Pareto-optimal labels stored in the subtree rooted at v

```
1:  if $v$ is a leaf then
2:      return label stored at $v$
3:  else
4:      if $\min_y(v.left) < \min_y(v.right)$ then
5:          return findParetoMinima($v.left, u$)
6:      else
7:          if $u < \min_y(v.left)$ then
8:              return findParetoMinima($v.right, u$)
9:          else
10:             return findParetoMinima($v.left, u$)          ▷ Do in parallel
11:             ∪ findParetoMinima($v.right, \min_y(v.left)$)
12:         end if
13:     end if
14: end if
```

Bulk Updates: The bulk updates required for paPaSearch are far more complicated than the localization of Pareto-optimal labels. They require the introduction of a `split` and `concat` operation on trees as defined in [MS08]. For both operations, we rely on the interpretation of the tree as a sorted sequence.

[2]Under the assumption of unique keys.



(a) *Pareto Queue* **(b)** *Search Tree*

Figure 7.1: *Illustration of the content of a Pareto-queue (7.1a) and a simplified view of its search tree representation (7.1b). The dashed line marks the current Pareto-front. In the search tree representation, the red node does not need to be scanned as its minimal y value is larger than the minimal y value in the subtree to its left. The search tree representation only shows a difference in relative y coordinates for dominated entries.*

Definition 7.2.

`concat`: *The* **`concat`** *operation is defined on two sorted sequences $s_0 = \langle l_0, \ldots, l_{i-1} \rangle$ and $s_1 = \langle l_i, \ldots, l_n \rangle$ if $l_{i-1} < l_i$. The required running time is in the order of $\mathcal{O}(\log \max(i, n - i + 1))$; in words it is logarithmic in the length of the longer sequence. Let s_0 be the larger sequence. First we descend into the tree of s_0 for height(s_1) − height(s_2) levels by following the pointers to the right. At this point, we add the right sequence to the tree and propagate the required updates upwards through the tree as we would have done during a ordinary insertion. See [MS08] for a closer description.*

`split`: *Consider a sequence $s = \langle l_0, \ldots, l_n \rangle$ that we want to split at l_i. The result of this split is given in two sorted sequences $s_0 = \langle l_0, \ldots, l_{i-1} \rangle$ and $s_1 = \langle l_i, \ldots, l_n \rangle$. From every node v on the path from the root to l_i we create two new nodes v_l and v_r. To v_l we append the children of v that are to the left of the path (or none if the path descends to the left) and to v_r the children that are to the right of the path. Exploiting the fact that the left trees are of strictly decreasing height and the right trees of strictly increasing height, we can concatenate the left trees into s_0 and l_i in combination with the right trees to s_1 in $\mathcal{O}(\log n)$ concatenations and $\mathcal{O}(\log n)$ total time. For further details, refer to [MS08].*

Based on these operations, we can describe a recursive parallel version of Algorithm 7.3. The reasoning behind the removal of ranges will become apparent later on when we describe the vertex local operations.

The bulk update operation can be implemented analogously to `bulkRangeRemove`. A major problem with the operations discussed is the potentially too finely grained parallelism. Viewing the nodes of a binary search tree as atomic units seems a bit

Algorithm 7.3 bulkRangeRemove, $|T|$ denotes the number of elements in T

Input: A search tree T, a list of ranges $\langle R_1, \ldots, R_k \rangle$, operating processor $i, \ldots, j$
Output: The search tree T without the elements in the ranges $R_1, \ldots, R_k$

```
1: if $k = 1 \lor i = j$ then
2:     sequentially remove $R_1, \ldots, R_k$ from $T$    ▷ by processor $i$, $\mathcal{O}(k \log |T|)$
3:     return $T$
4: else
5:     $(T_1, T_2) := \mathrm{splitAt}(T, R_{\lfloor k/2 \rfloor + 1})$    ▷ by processor $i$,$\mathcal{O}(\log |T|)$
6:     bulkRangeRemove( $T_1, \left\langle R_1, \ldots, R_{\lfloor k/2 \rfloor} \right\rangle, i, \ldots \lfloor (i+j)/2 \rfloor$ )    ▷ in parallel with
7:     bulkRangeRemove( $T_2, \left\langle R_{\lfloor k/2 \rfloor + 1}, \ldots, R_k \right\rangle, \lfloor (i+j)/2 \rfloor + 1, \ldots, j$ )
8:     return concat$(T_1, T_2)$    ▷ processor $i$, $\mathcal{O}(\log |T|)$
9: end if
```

wasteful. In the database community, B-tree variants have proven to be more efficient, especially when it comes to cache usage (compare [LLSL10] for example).

The operations on the Pareto-queue are connected to the vertex local operations at two points. The first point is the generation and the grouping of candidate labels. The second is the collection of update request to the Pareto-queue after the merge of new label candidates in combination with the removal of dominated labels from their respective label-sets. Both operations require a parallel sort, creating a lexicographic order by vertex-id, x and y-coordinate of the label candidates during the former operation, a lexicographic order by x and y-coordinate and the vertex-id as a tie-breaker during the latter.

7.2.2 Vertex Local Operations

The vertex local operations mostly describe a filtering step. This step initially operates on the new candidate labels alone to filter out already dominated labels. During the merge operation, this process continues to detect previously existing labels that are now dominated.

The connection between the Pareto-queue and the local operations was previously described as step two (generating candidates) and step three (grouping candidates). In this two step approach, paPaSearch first creates new labels by relaxing all arcs adjacent to the label's vertex. To avoid load balancing issues due to varying vertex degrees, [SM13] propose a prefix-sum over all degrees to distribute the different arcs to the processors in a fair way. The different labels are sorted lexicographically to group them by their respective vertex-id and to prepare the data for later interaction with the label-sets. The sorted labels are distributed to the available processors by means of another prefix-sum, potentially splitting the spanning multiple processors with the labels of a single vertex; this directly implies the need for synchronization and concurrent access to the different label-sets.

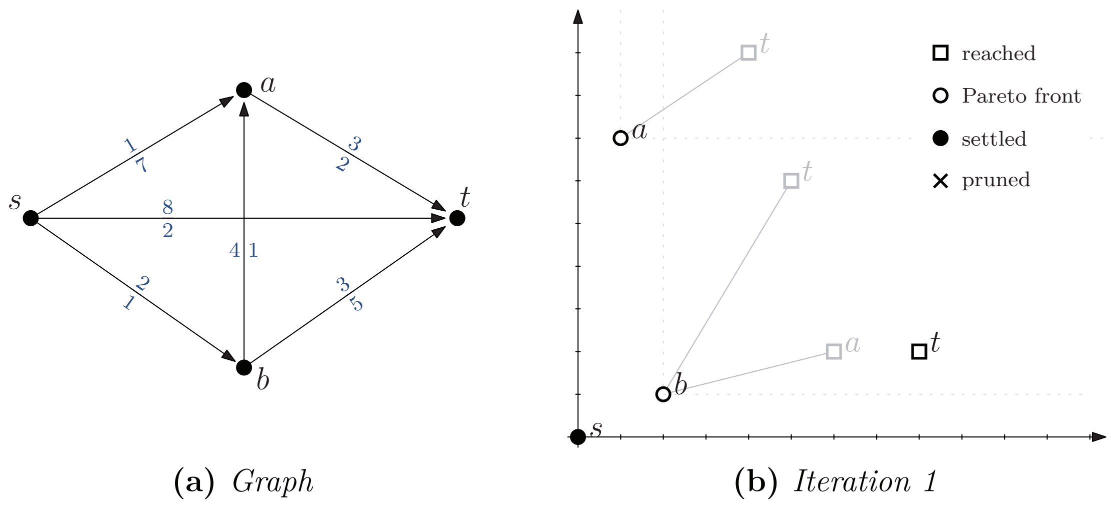
Pareto-Optima Among Candidates. Usually we expect a low number of new label candidates and a large number of labels in the local label-sets. Filtering the candidates amongst themselves promises a reduction of overhead when operating with large label-sets. As just stated, a parallel algorithm is required to implement the filtering, as each set of new labels might be assigned to an arbitrary number of processors. [SM13] describes a method based on prefix-minima to compute the Pareto-optimal labels. Due to the lexicographic ordering by the x-coordinate as first and the y-coordinate as a second criterion, every new minimal y coordinate marks a new Pareto-optimal label (compare Figure 7.1). Prefix-minima can be obtained in parallel in an identical manner to prefix-sums.

Merging Label Candidates and Label-Sets. According to one of the authors of [MMdlCRS12], the most common structure for storing label-sets is an unbounded array. We were unable to verify this for ourselves, though. Inserting labels into an unbounded array at arbitrary positions requires an expected linear running time in the size of the array. In addition, parallel execution of operations seem impossible due to the high degree of dependence between the different operations over the full range of the array. For our implementation of paPaSearch, we therefore follow the general proposal of Sanders and Mandow to store label-sets in a search tree. Rather than using a binary search tree, however, we utilize a B-tree implementation; we describe this approach in Section 7.3.3.

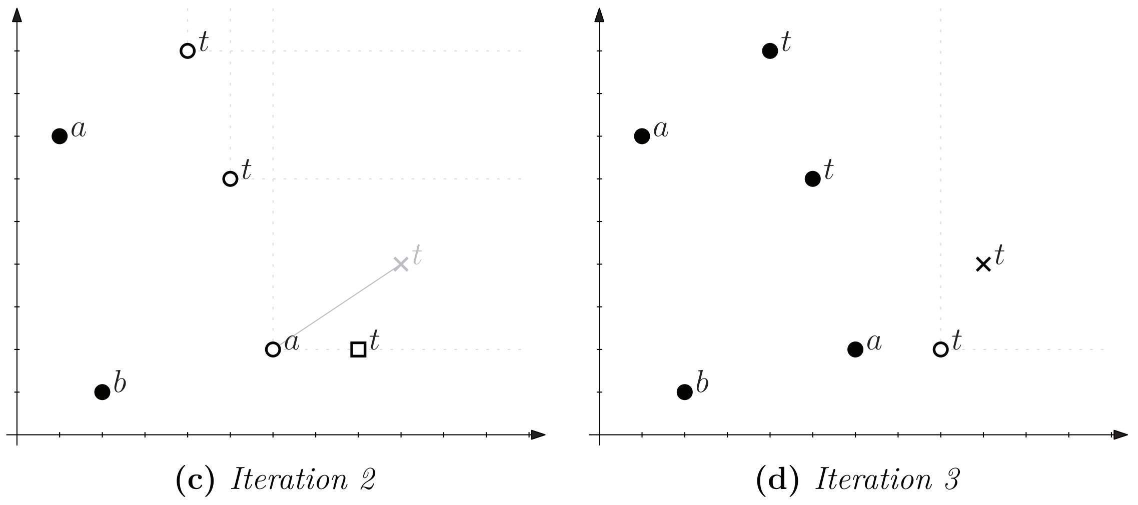
In the bi-criteria case, this representation allows for the localization of a label with respect to both its x-coordinate and for its y-coordinate in time of $\mathcal{O}(\log |L[v]|)$; this is due to the increasing x-coordinate and decreasing y-coordinate in the lexicographic order of non-dominated labels. The range of dominated old labels to a new candidate label, in case it is not dominated by an old label itself, can be obtained by locating it both with respect to its x coordinate (the location marks the start of a sequence of potentially dominated labels) and its y-coordinate (which defines the end of the sequence of dominated labels). This sequence of ranges may contain overlapping ranges. By filtering and pruning, we can construct a list of non-overlapping ranges that works well with Algorithm 7.3. For insertion, the same approach as in Algorithm 7.3 has already been proposed in [FS07]. The implementation provided by Frias and Singler, currently part of the GCC STL, operates as the frame of reference for our own considerations.

7.3 An Efficient Implementation

In the conclusions to [SM13] Sanders and Mandow state:



(a) *Graph* **(b)** *Iteration 1*



(c) *Iteration 2* **(d)** *Iteration 3*

Figure 7.2: *An example execution of the parallel Pareto-optimal search*

> As paPaSearch is explained here, it might be too complicated to be practical. We still decided to go to this level of sophistication because this yields elegant and easy to interpret asymptotical results. However we believe that a simplified version of the algorithm might be useful in practice. In particular, we may get away with implementations that work sequentially at each single node in the node local operations. The batched nature of all the operations might also help with cache efficiency.

In this chapter, we discuss some modifications and simplifications to the previously presented version of paPaSearch. The results presented here are based on our own publication [EKS14] and are twofold. First, we present an efficient adaptation of weight-balanced B-trees, a data structure introduced by Arge and Vitter [AV03]. Additionally, we present some simplifications and adaptations to paPaSearch that result in an efficient parallel implementation of a bi-criteria search.

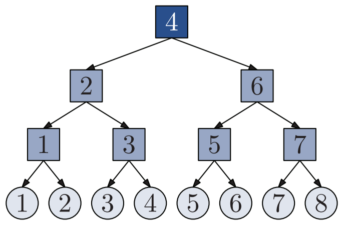
7.3.1 B-Trees

One parameter not considered in the asymptotic analysis of paPaSearch is the effect of modern memory hierarchies. The simplified model of constant time memory access does not reflect the reality. For the most part, memory access is subject to a full hierarchy of caches – usually referred to by L1, L2, and L3 – all data passes through on its way from the Random Access Memory (RAM) to the registers used by the CPU. Efficient data structures are often evaluated for their cache behavior, leading to research on both cache-aware and cache-oblivious algorithms. For example the well-known van Emde Boas layout [vEBKZ77] describes a cache-efficient layout for a binary search tree. The cache paradigm is closely related to considerations on hard disc access for databases, a field for which the B-tree has become one of the standard data structures.

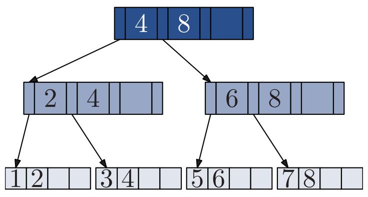
Basic Concept. First introduced by Bayer and McCreight ([BM72, HM82, BM02]), today the B-tree has become a standard data structure for systems operating on memory that focuses on blocks for memory transfers. B-trees are, in their core essence, a variation of the classical binary search tree. The main difference is in the number of splitter keys per node. In a binary search tree, every node distinguishes between elements that are designated for the left sub-tree and those that are put into the right sub-tree via a single comparison to the node's splitter key. In a B-tree, conceptually we unite a set of different search tree nodes into a single B-tree node. Instead of a single splitter element, a B-tree node consists of a sequence of splitters, each of which behaves exactly like the splitter in a binary search tree. All elements smaller or equal to the splitter are located to the left of it, all larger elements are located to the right. Comparisons to determine into which sub-tree an element belongs can now be made via binary search or by linear scanning, utilizing multiple elements of a cache-line rather than a single element. The number of these splitter elements in a node itself is a tuning parameter. For illustration purposes, we present a juxtaposition of a binary search tree and an according B^+-tree[3] in Figure 7.3. Balancing in a B-tree requires the possibility of splitting and fusing different nodes. At first glance it is hard to see the reasoning behind this extension. The benefits, however, will become apparent in the next paragraph.

Caching Effects. In the presence of caches or even more when considering spinning discs, the B-tree provides some advantages; non-uniform cost for memory access lets the B-tree shine in comparison to a classical binary search tree. Linear access patterns to a continuous range of elements (compare [DGNW13] or [ADGW11]) benefit the

[3] A B-tree that stores elements only in the leaves.



(a) *Binary Search Tree*



(b) *B^+-tree*

Figure 7.3: *A binary search tree (left) and a B^+-tree (right) representation of the same set of keys. The root found at the top, leaves on the bottom and inner nodes in between.*

most from current hardware features such as *prefetching*[4] or caching. Whereas we are most likely to generate a cache-fault[5] for close to every comparison at a node when traversing a binary search tree[6], we utilize every cache-line for multiple comparisons when traversing a B-tree. The literature presents a wide range of different material that concerns B-trees. At this point, we only mention a small selection. In terms of their cache efficiency, even cache-efficient layouts for binary search trees (e.g. [CHL99]) cannot compete with the B-tree [Sai09]. The original concept has received quite a bit of attention. Most prominent are versions that consider *cache-aware* [CGM01, RR00] or *cache-oblivious* [BDFC00, BFGK05] variants of the B-tree.

Weight Balanced B-tree. An extension to the balancing constraints of B-trees is due to Arge and Vitter [AV03]: they introduce the so-called weight-balanced B-tree, an extension to the weak B-tree or (a,b)-tree that allows a wider range of children for inner nodes[7]. Arge and Vitter abandon the restrictions on the number of children for a new constraint based on the size of a (sub)tree. The size, or weight, of a tree is defined as the number of elements stored within it. Their B-tree variant is fully characterized by Definition 7.3.

Definition 7.3 (Weight-balance Constraints). *Let $\mathcal{T}$ be a weight-balanced B-tree with* branching parameter $b > 4$ *and* leaf parameter $k > 0$. *The following conditions hold:*

1. *all leaves are on the same level and have a weight between k and $2k - 1$*
2. *the weight of an internal node on level l is limited by $2b^l k$*

[4]A process in which the processor heuristically transfers likely to be accessed data from memory to the cache.

[5]An access to a un-cached region of memory, triggering the transfer of data from the main memory to the cache.

[6]Cache aware layouts reduce this effect.

[7]In the standard B-tree, all leaves are on the same level (usually referred to by level zero) and each inner node has between a and $2a - 1$ children for some constant a.

3. an internal node other than the root has its weight bounded from below by $\frac{1}{2}b^l k$

4. the root has more than a single child

As a result from these restrictions [AV03], inner nodes offer between $\frac{b}{4}$ and $4b$ children and the root node has between 2 and $4b$ children.

Further Studies. Different efforts have been made to further increase the performance of B-trees. Rao and Ross [RR00] propose a variant, the CSB^+-tree that increases the branching factor of inner nodes by eliminating the need to store a pointer for every child. Instead, the different children are kept in a continuous block of memory, making them addressable via a single pointer to this memory block; Chen et al. [CGM01] propose a similar idea and additionally treat inner nodes different from leaves and the root by increasing their branching factor. Additional overhead to fetch these larger nodes can be hidden in prefetching. The positive effect of larger nodes has also been studied in [HP03]. Larger nodes seem to keep a good balance between cache misses, Transaction Look-aside Buffer (TLB) misses and number of executed instructions. Even more techniques to optimize the cache performance of B-trees have been presented by Graefe and Larson [GL01].

Parallel B-trees. The B-tree is also a prominent data structure in the database community. A natural way of improving performance on a lot of data structures is to parallelize them. In general, in accordance to Sitchinava and Zeh [SZ12], we distinguish between *concurrent data structures* – a type of data structure that allows multiple threads of execution to interact with it simultaneously – and *parallel data structures* – a type of data structure that executes single operations in parallel in order to speed up queries or updates. Many different locking-based parallel B-trees haven been proposed [PMSSY96, PMSS04, LSSSS07]; recent results indicate better performance for lock-free implementations [SCK$^+$11], though.

7.3.2 Further Parallel Queues

Next to the work in the database community, other approaches have studied in how far priority queues or search trees can be augmented to support parallel processing. Here, we only present a selection of these approaches to provide ideas for further reading.

Parallel Buffer Trees. Sitchinava and Zeh [SZ12] present a parallel buffer tree for the Parallel External Memory (PEM) model. The tree is able to process a sequence of `insert`, `delete`, `find`, and `RangeQuery` operations in an optimal number of parallel Input/Output (I/O) operations. The implementation heavily uses sorting and distribution/merging to achieve these optimal bounds. The practicability of the presented data structure is, however, not investigated in the paper. Similar to the proposed structure of [SM13], the implementation looks forbiddingly expensive.

Parallel Priority Queues. In the efforts to parallelize Dijkstra's algorithm, some research considered the parallelization of the priority queue. The abstract data type has been mentioned as early as 91 by Pinotti and Pucci [PP91] and has received quite a bit of attention over the years (e.g. [DP92, CH94, BDMR96, HAP12]). We know of three kinds of approaches to parallelize operations on a priority queue. The first type of parallelization speeds up individual queue operations using a small number of processors [Bro99], the second type operates in bulk updates of $\mathcal{O}(k)$ items utilizing $\mathcal{O}(k)$ processors (e.g. [DP92]), or allowing concurrent operations [RK88]. Some of these queues require the number of elements to be large in comparison to the number of cores used. However, Sanders has shown [San98] that also a small number of elements may be enough to gain performance boosts by parallel execution. How well the results transfer to current hardware remains an open question.

7.3.3 A Practical Parallel Pareto-queue

For our own implementation of the Pareto-queue, we chose a weight-balanced B-tree [AV03]. Our main reasoning behind this relies on two arguments. First, the cache efficiency of B-trees offers big advantages over classical search tree implementations. As previously argued, this has been observed in many studies. Our second criterion is founded in the simplicity we gain from the weight-balance constraints (see Definition 7.3) for re-balancing efforts.

The data structure we propose is a standard weight-balanced B-tree with a minor augmentation: just as Sanders and Mandow propose, we store the minimal y-coordinate of the label contained in the (sub-)tree that is induced by a given node within the node itself. Additionally, we opt for nodes that are several cache lines in length (cf. Hankel and Patel [HP03]). Aside from these modifications, the general layout of our data structure remains the same as proposed by Arge and Vitter. In addition, we introduce methods to operate on this Pareto-queue in a bulk update process. All of our proposed methods operate in a task-parallel model, in particular we use the Intel® Threading Building Blocks[8] for our purposes. In the following paragraphs, we describe the different tasks that fully define our own implementation of the Pareto-queue.

Retrieval of Pareto-optimal Labels. The retrieval follows the same design that has already been proposed for paPaSearch. Recursively and in parallel we descend down the tree until we reach a leaf. In the leaves, we scan the labels from left to right to report all Pareto-optimal labels. Due to the lexicographic order of the labels, every new minimum for the y coordinate marks a new Pareto-optimal label (compare Figure 7.1). We give our modified Algorithm 7.4 as a comparison to Algorithm 7.2.

One can see that our retrieval algorithm and the original Algorithm 7.2 behave identically in their concepts. Some conceptual additions, however, set it apart from the

[8] `www.threadingbuildingblocks.org`

Algorithm 7.4 findParetoMinima

Input: A B-tree node v, a domination value u
Output: All Pareto-optimal labels stored in the subtree rooted at v

```
if $v$ is a leaf then
    return all Pareto-optimal label stored at $v$        ▷ by scanning
else
    for all children $c$ of $v$ do
        if $\min_y(c) < u$ then
            spawn parallel task: findParetoMinima($c, u$)
            $u = \min_y(c)$        ▷ simulates the decision of Algorithm 7.2
        end if
    end for
    return collectSubtaskResults()        ▷ dummy method, actual implementation differs
end if
```

algorithm given in [SM13]: Next to our cache-efficient data structure, we also introduce a different method for the aggregation of the extracted labels. We keep a large region of memory ready to hold the extracted labels. Whenever a task processes a leaf, it allocates memory from that region via a simple `fetch_and_add`[9] to a global pointer that points to this memory region. To avoid unnecessary conflicts between caches, we allocate memory in sizes that correspond to cache lines.

Every thread uses its allocated memory, or buffer, until it runs full. When the retrieval is finished, some of the buffers might not have been filled. Luckily for us, there is a way around this problem that comes at barely any extra cost: As a further optimization to paPaSearch, we integrate the generation of new candidate labels directly into the retrieval step. The generated label candidates have to be sorted with respect to their vertex-id and lexicographically with respect to their coordinates anyhow.

Threads utilize this sorting mechanism and simply fill their (partly empty) buffer with a few additional dummy elements. These dummy elements are assigned to a nonexistent vertex (Identification (ID) set to infinity) and are collected at the end of the label-set by the sorting algorithm. By keeping track of the number of generated labels, we can discard these dummy elements after the labels have been sorted. The number of these additionally sorted labels is bounded by the number of threads used – small in comparison to the number of considered labels – and the number of cache lines we allocate as a buffer. As long as we assume the branching parameter and the leaf parameter as a constant, our modifications regarding the pure retrieval of Pareto-optimal labels have no influence on the asymptotic analysis of paPaSearch.

[9]An atomic method that retrieves the value of a variable and adds an offset to it without the possibility of interference by another thread.

As discussed here, generating the candidate labels during the extraction, however, could introduce a bottleneck. In our experiments, we did not encounter any problems with this on typical inputs. For graphs with highly nonuniform distribution of vertex degrees, especially when single vertices with a very high degree are part of the graph, we could introduce a threshold for a maximal vertex degree to process sequentially and spawn multiple tasks to handle vertices with a high degree in parallel. This is currently not implemented in our code, though.

Weight Deltas. The main ingredient for our Pareto-queue during updates is a sequence of `insert` and `remove` requests, sorted lexicographically by the affected label. Our implementation relies on detecting imbalances within the B-tree early on. We do this by computing a single prefix sum in advance: Each update is assigned an integer value of ± 1, indicating an insertion (+1) or a removal (-1) of a label. A prefix sum over these assigned values allows for a direct calculation of something that we call weight-delta: the weight-delta δ_{ij} marks the change to the weight of the B-tree between two requests u_i and u_j from the sequence of updates[10] $\mathcal{U} = \langle u_1, \ldots, u_k \rangle$. The difference between the prefix-sum entries at position i and j mark the change to the portion of the tree that is bounded to the left by the label of u_i and to the right by the label of u_j .

Updating the Pareto-queue. Our Pareto-queue operates on a sequence of updates $\mathcal{U} = \langle u_1, \ldots, u_k \rangle$. Each update consists of a combination of a label and a vertex-id, a unique combination in the Pareto-queue. Additionally, we distinguish between two types of updates; the `insert` requests store a new label in the Pareto-queue, the `remove` requests delete an existing label from the Pareto-queue. A removal is triggered either because the label is dominated or because it has been one of the Pareto-optimal labels in the last iteration.

We combine the removal and the addition of new labels into a single step to avoid unnecessary balancing operations. Due to the large amount of labels that are removed from the Pareto-queue at a time, we are likely to empty parts of the B-tree to the point that it violates the minimum weight constraints. Newly generated labels, however, are likely to fill the void and restore the balance.

Our insertion algorithm works in a way similar to the algorithm proposed in [FS07] and is given in pseudo-code in Algorithm 7.5. At every node, we split the sequence into multiple parts and apply it to the B-tree in parallel; for an illustration see Figure 7.4. Due to the large branching factor of a B-tree in comparison to the binary search tree, we generate a lot of tasks early on.

The way the algorithm is currently described, it works almost without the requirement of additional memory (in place). The layout of the tree itself is not changed, as long as the balance constraints are not violated. In case of violated balance constraints, we

[10]The exact description of an update follows shortly.

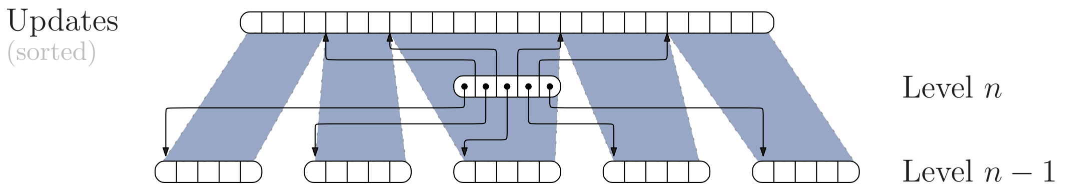


Figure 7.4: *Splitting an update sequence at level n to apply it to level $n-1$ in parallel.*

start a new task that is described in the following paragraph. If a task reaches a leaf, we apply the labels sequentially to that leaf by merging the updates and the requests. The mix of updates to the leaf requires a small trick, though, to be efficient. As our leaves store labels in an array, removal and insertion of labels can be costly if done on a per label basis. We avoid the problem of shifting labels back and forth by merging the old leaf and the updates into a spare leaf that is associated with every thread. This brings the merge process to a simple linear scan of both labels and updates. On completion, we simply swap out the spare leaf for the modified leaf, which now becomes the spare leaf for further updates done by that thread.

Touching all labels during the insertion of a few new candidates sounds expensive at first. The distribution of all update requests we found in typical problem instances, however, indicates that we touch close to every label anyhow (compare Figure 7.8).

Intuitively, it looks like this process is prone to load imbalances if a large number of updates target a small part of the B-tree. One can see, though, that the number of elements that can be stored in a sub-tree is limited. As a result, the number of remove requests is limited as well. In the same way we can put a limit on the maximal number of insertions and in total can claim that severe imbalances in the update sequence imply an imbalance in the B-tree as well. These imbalances are handled in their own way, as we explain in the next paragraph.

Balancing the Pareto-queue. To balance the Pareto-queue, we utilize a technique already described by Overmars [Ove83]: partial rebuilding. As one can see in Algorithm 7.5, we can detect imbalanced sub-trees, purely relying on the information of the weight deltas and the weight function of the B-tree, at the root (of a sub-tree) in need of balancing.

For the partial reconstruction of our B-tree based Pareto-queue, we initiate a balancing task for every consecutive range of sub-trees that is about to become unbalanced. We also allocate an appropriate number of leaves for the balanced (sub)-tree. From this point forward, the update procedure and the balancing procedure operate in a very similar manner. We recursively – and in parallel – distribute the updates by descending down the B-tree until we reach a leaf. Contrary to the previously described update procedure, we do not update the leaves in place. Instead, we directly store the merged sequences into the newly created set of leaves. From these leaves,

Algorithm 7.5 updateParetoQueue

Input: A B-tree node v, an update sequence $\mathcal{U}$, a prefixsum $\mathcal{P}$ over U
Output: a modified node v that represents the sub-tree after $\mathcal{U}$ has been applied

```
if $v$ is a leaf then
    merge $U$ and $v$                                    ▷ can be done in place
    return $v$
else
    bounds $b[]$
    for all splitters $s$ of $v$ do                      ▷ locate the splitters in the sequence
        $b[s]$ = binarySearch($s$,$\mathcal{U}$)
    end for
    balanced = true
    for all children $c$ of $v$ do
        if $weight(c) + \delta_{b[c]b[c+1]}$ violates weight constraints then    ▷ from $\mathcal{P}$
            balanced = false
        end if
    end for
    if balanced then
        for all children $c$ of $v$ do
            if $b[c+1] - b[c] > 0$ then                  ▷ parallel tasks for nonempty sequences
                $c$ = updateParetoQueue( $c, \mathcal{U}[b[c] : b[c+1]], \mathcal{P}[b[c] : b[c+1]]$ )
            end if
        end for
    else
        balanceTree($v$,$\mathcal{U}$,$\mathcal{P}$)                          ▷ details follow
    end if
    return $v$
end if
```

we construct a fully balanced tree to replace the unbalanced sub-tree(s). During the update process, we can again make use of the weight-deltas and the weight function to determine the desired location of every label in the final leaves. As we construct a fully balanced tree, the number of elements in the leaves is well defined; especially because of the known size of the updated tree due to its weight and the information from the weight-delta. For every leaf l, the sum of all preceding sibling nodes (restricted to the unbalanced part) in combination with the weight-delta provides the exact number of surviving labels that are lexicographically smaller than the labels stored in l.

The main difference to Algorithm 7.5 is how we handle large update sequences that fall into a single leaf. In the update process, we were hiding the imbalances in our balancing procedure. Now, we have to deal with the fact that there might be a large

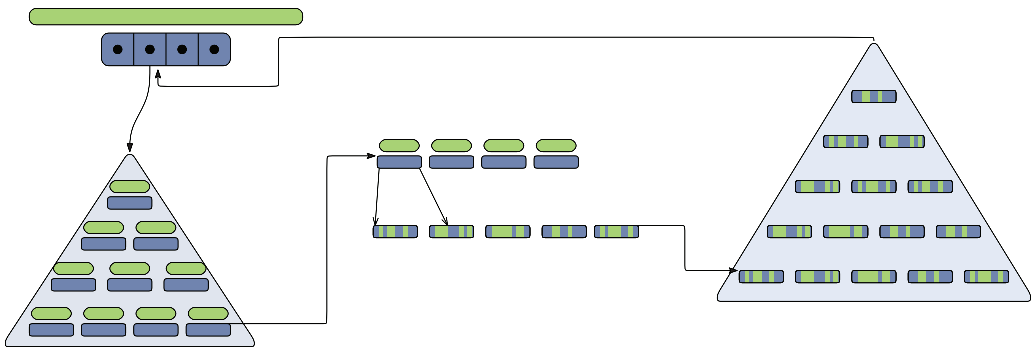


Figure 7.5: *Illustration of the merging process. New leaves are created from the (distributed via the tree) updates and pre-existing content. The result is used to create a new tree that can be swapped in for the unbalanced once.*

sequence targeting the content of a single leaf. In case of such a sequence, we know that it contains a lot of new labels, due to the limited amount of labels that can be deleted from a leaf. We handle this sequence by splitting the sequence into appropriate chunks and locate the affected subranges within the leaf via binary search. If we choose large enough chunks, the amount of cache conflicts between two processes writing to the same destination in the cache should be negligible.

From the now perfectly created leaves, we can construct a tree in a recursive parallel process. This new tree can be integrated into the node from which we started our balancing operation as the amount of space for children is guaranteed to be large enough. We give a schematic illustration to this process in Figure 7.5.

7.3.4 Engineering paPaSearch

Conceptually very simple, the paPaSearch algorithm does not leave much room for engineering efforts. The main overhead is hidden in the Pareto-queue, for which we already offered a solution. We demonstrate the efficiency of our solution in the upcoming Section 7.4. Our main goal in the implementation of the paPaSearch algorithm was to minimize typical bottlenecks such as poor cache usage or overly fine-grained parallelism. By integrating the label generation into the process of detecting Pareto-optimal elements, we already eliminated a source of unnecessary overhead due to additional load-balancing. While not currently implemented, our algorithm could be augmented to avoid load balancing issues due to high degree vertices. Handling leaves sequentially avoids overly fine-grained parallelism, compared to a task that is associated with the much higher count of nodes in a binary search tree. In addition, the sequentially processed leaves allow for a different form of label filtering that has proven to be far more efficient than the filtering method proposed in [SM13]. In our implementation, the sequential update of a single node looks like a bottleneck. However, later Sections present arguments that indicate a benefit to use our own Pareto-queue also for storing label-sets, in case the workload on label-sets gets larger[11]. As a result, it would pose no difficulties to extend our algorithm to handle large update sequences to a single vertex in parallel at only minimal additional cost.

Interleaved Filtering. In [SM13] the authors propose to filter candidate labels among themselves prior to any interaction with the label-set of the respective vertex. The reasoning behind this step is that we expect the number of candidates per vertex to be small in comparison to the label-set and aim at minimizing the interaction. This process, however, ignores some of the filtering capabilities that stem from a closer interaction between the candidate labels and the label-set. In our engineered version of paPaSearch, we opt for an interleaved filtering step that does not require any additional operations over the previously proposed step. In a linear scan among the candidate labels we search for the next non-dominated label[12]. Due to the rather small size of the label-sets we experienced in practice, a binary search seems excessive. In scenarios that exhibit rather larger candidate sets, such a variation could be integrated.

This undominated label has to be checked in relation to the label-set. Instead of first filtering all label candidates, we turn directly to the label-set and locate its position among the previously existing labels. If we find the label to be Pareto-optimal, we simply add it to the label-set and search for dominated labels in the label-set. In case it is dominated by another label, however, we can utilize the dominating label (or to be exact: its y-coordinate) for filtering further candidate labels. While the change seems insignificant, we found this modification to work much better than the originally proposed filtering step.

7.3.5 Analysis

The asymptotic analysis of our implementation only needs to be discussed in terms of the changes we applied to the theoretical work of Sanders and Mandow [SM13]. For the original asymptotic analysis, the authors provide a bound of $\mathcal{O}\left((N+M)\log M \cdot p^{-1} + n \log p \log M\right)$ for a set of N label scans and M arc relaxations on p processors. In general, our algorithm follows the design of Sanders and Mandow with a different data structure as the main variation. As a result, most of their analysis transfers to our implementation as well. While we did not implement some optimizations which would be required for a fully equivalent analysis, the interleaved operations performed by our algorithm can be adjusted to handle the currently not-considered corner cases as well.

For example, our current implementation uses the extraction process to directly generate candidate labels. Theoretically, vertices of very high degree could result in load imbalances as we currently perform the relaxation process sequentially. The respective

[11]In case of few labels, the maintenance overhead of the B-tree is larger than its benefit.

[12]Initially, this is always the first of the lexicographically sorted labels.

task could, however, be executed in parallel too, given our task parallel setting. On our evaluated networks, this additional step did not offer reasonable benefits to concern ourselves with its implementation.

Some adjustments, as the interleaving of some computations have no influence on the analysis. Others require some small changes to comply with the asymptotic running time. An example would be our consideration to treat a vertex or a leaf as an atomic unit. These can, however, be fixed by performing operations in parallel using constant size chunks distributed dynamically via work stealing. On the tested networks, this approach was unnecessary. In our analysis, we focus on the tree operations as they offer the most drastic change in comparison to the theoretic description in [SM13].

Extraction of Pareto-optimal-labels. The extraction process is equivalent in its analysis to the procedure discussed by Sanders and Mandow, only differing in the size of a leaf. Whereas the original discussion ends up with a single label, our implementation scans a full leaf for Pareto-optimal labels. Under the assumption of a constant maximum size for a leaf and a constant size for the inner nodes, this does not affect the asymptotic analysis.

Bulk Update By its definition, a weight-balanced B-tree of size N is of height $\mathcal{O}(\log N)$. Assuming an update sequence of size s that does not introduce any imbalances, the sequential update cost of the tree is bounded by[13] $\mathcal{O}(s \log N)$. In the parallel case, however, we need to consider our distribution mechanism. The height of the B-tree remains at $\mathcal{O}(\log N)$. In every level of the B-tree, we consider the sizes of the locally relevant update sequence and the amount of the splitters. Due to the sorted nature of both sequences, we can use binary search to localize the splitters within the update sequence or vice versa.

As long as the updates do not result in any imbalances, the amount of labels that can be added to a given sub-tree is limited. The weight of a sub-tree of height h is limited to be in $\left[\frac{b^h a}{4}, b^h a\right]$. Given the additional removal requests, the maximum s on every level ℓ is limited by $b^\ell a$. This consideration results in a critical path of length $\mathcal{O}\left(\sum_{\ell \in [1, \log N]} b \log b^\ell a\right)$ which equals $\mathcal{O}(b \log N(\log a + \log b \log N))$. Assuming a and b as constant, we end up with a critical path of length $\mathcal{O}\left(\log^2 N\right)$.

The sequential workload is given as $\mathcal{O}(sb \log N)$, where we compare each of the b splitters to every single label on all of the $\mathcal{O}(\log N)$ levels of the B-tree. In total, we get an asymptotic running time of $\mathcal{O}\left(\log N(\frac{s}{p} + \log N)\right)$. The critical path is longer than the one using a classical binary search tree. The process described by Frias and Singler [FS07] requires $\mathcal{O}\left(p \log N + \frac{s}{p} \log N + \log p \log N\right)$ time for splitting, inserting, and concatenating respectively. In our algorithm, however, we expect $s \in \Theta(N)$ and as a

[13]Localization of a given element in a leaf can be done in constant time ($\mathcal{O}(\log b)$).

result[14], the running time is still dominated by $\mathcal{O}\left(\frac{s}{p}\log N\right)$. $s > p\log N$ is sufficient for this observation. In the interesting cases, our algorithm offers the same performance as the implementation of Frias and Singler. Assuming a constant size for b, our asymptotic performance is equal to the one offered in [FS07].

Partial Reconstruction. The construction of a B-tree of size N can be achieved in $\mathcal{O}(N)$, due to the geometrically decreasing work per level. In parallel, we can achieve this construction in $\mathcal{O}\left(\frac{N}{p} + \log N\right)$, given that we use a recursive mechanism for the construction and the tree is of height $\mathcal{O}(\log N)$. In the combination with the construction of perfectly balanced subtrees, we can use an amortized argument to account for the cost of rebalancing as the critical path length only decreases[15]. The construction of *half-full* B-trees guarantees that we can insert or remove an equal amount of labels from the B-tree. In our case, we can use this argument to charge the workload of the construction to the $\Theta(N)$ operations that result in the imbalance of the respective tree.

7.3.6 Increasing Parallelism

For Dijkstra's algorithm, as already mentioned, Δ-stepping [MS03] can be used to achieve parallel execution. The notion behind this is that we can operate on many of the currently reached, but unsettled, vertices at a time without performing unnecessary work. Remember that Dijkstra's algorithm operates approximately in a cyclic manner around the source of the query (compare Figure 3.1). Obviously, since Dijkstra's algorithm requires $\mu(a) \geq 0, \forall a \in A$, we can settle all vertices of minimum weight at the same time. Δ-stepping takes this approach a step further and selects all vertices whose weight is within Δ of the current minimum weight in the priority queue. While this, of course, does not guarantee a final label for every processed vertex, the benefit in parallelism is larger than the drawback of the unnecessarily performed work.

The same approach can be used to increase the parallelism available during an instantiation of the paPaSearch algorithm. However, only single dimension relaxation integrates seamlessly into our B-tree based Pareto-queue. We can integrate a y-coordinate based Δ-stepping by simply modifying the extraction process by adding the Δ value to the minimum y value associated with the left sibling before descending. In the same way, while scanning a leaf, we adjust the minimal y value found by Δ for further pruning. As a result, the extraction process finds all labels that are within Δ with respect to the y-coordinate of the Pareto-front.

Whereas the integration of a y-coordinate based Δ-stepping can be done with ease, the same does not hold true for the x-coordinate. Our experiments show (see Section 7.4.4) a large benefit for the y-coordinate already. New techniques to enable

[14]In our experiments, we measured $0.1 \cdot N \lesssim s \lesssim 0.2 \cdot N$.
[15]The splitters can be selected perfectly without additional work from the sorted sequences.

two-dimensional (or even higher dimensional) Δ-stepping pose an interesting topic for further research.

7.4 Experimental Evaluation

In our experimental evaluation, we focus on two different aspects. First we discuss the modified weight-balanced B-trees and their behavior regarding the bulk operations. Afterwards, we present a close look at the performance of the paPaSearch algorithm in our engineered implementation. To allow better scaling observations, we use M2 in this chapter.

7.4.1 Pareto Queue

For our experiments that study the performance of our B-tree implementation, we follow the general setup of the only competitor known to us that operates in the same setting as we do: the `mcstl::set` of Frias and Singler [FS07]. Their experiments evaluate the insertion of a range of elements into a previously constructed search tree, both for a uniform distribution of the labels as well as for a skewed set of labels. For the skewed input set, we only generate labels that fall within the first few percent of the already inserted labels. Our Pareto-queue implementation itself can be configured with two parameters, the branching factor b and the leaf parameter l.

Parameter Tuning. To test the performance of our data structure, we performed a series of experiments to study the impact of both configuration parameters on the performance of our B-tree. For a range of different choices, we performed experiments for pure bulk insertion and bulk removal requests as well as some experiments combining update requests that add as well as remove some labels.

We follow the general approach of Frias and Singler [FS07] for our first experiments: Our tests operate on different batch sizes β. Every batch is applied to a tree of size n with $n := \beta \cdot \rho$ for some ratio ρ. After the initial construction of the tree, we first invalidate all caches before applying the batch operations. The elements to `insert`/`remove` are selected to be among the first $\sigma \cdot n$, $\sigma \in [0,1]$ of the already contained elements to show the behavior of our tree on skewed input instances.

Our experiments consider all variations of $\beta \in \{10\,000, 20\,000 \ldots, 100\,000\}$ for ratios $\rho \in \{10, 20, \ldots, 100\}$ and $\sigma \in \{0.1, 0.2, \ldots, 1.0\}$. For all of these combinations, we check different smaller leaf parameters $k \in \{8, 16, 24, 32, 40, 48, 56, 64, 128\}$ and larger leaf parameters $k \in [256, 384, 512, 640, 768, 896, 1\,024]$ as well as a range of different branching parameters $b \in \{8, 16, 32, 64, 128, 192, 256\}$ for a different number of threads performing the work (compare Definition 7.3 for the definition of k and b).

Due to the large amount of resulting data, we can only discuss a selection at this point. Figure 7.6 depicts three heatmaps; the first averaged over all our measurements, the second focusing on skewed input sequences, and the third depicting sparse updates. The data indicates that in general it seems best to operate in lower branching factors and with rather large values for the leaf parameter. Skewed update sequences seem to have only a minor effect on the selection; this is evidence of the good integration of skewed sequence handling into our algorithm. Very sparse updates, however, show a different behavior. The sparser the access to the tree gets, the less we benefit from the larger leaf sizes. For sparse updates, it seems best to both increase the branch factor slightly as well as to decrease the leaf parameter (see Figure 7.6c). Still, our implementation performs well in the general configuration of larger leaf size coupled with a low branching factor, even for sparse updates.

The measurements presented in Figure 7.6 only consider insertions and were performed on M1[16]. Additional measurements for mixed updates (both insert- and delete-operations) as well as pure deletion updates show near identical performance characteristics.

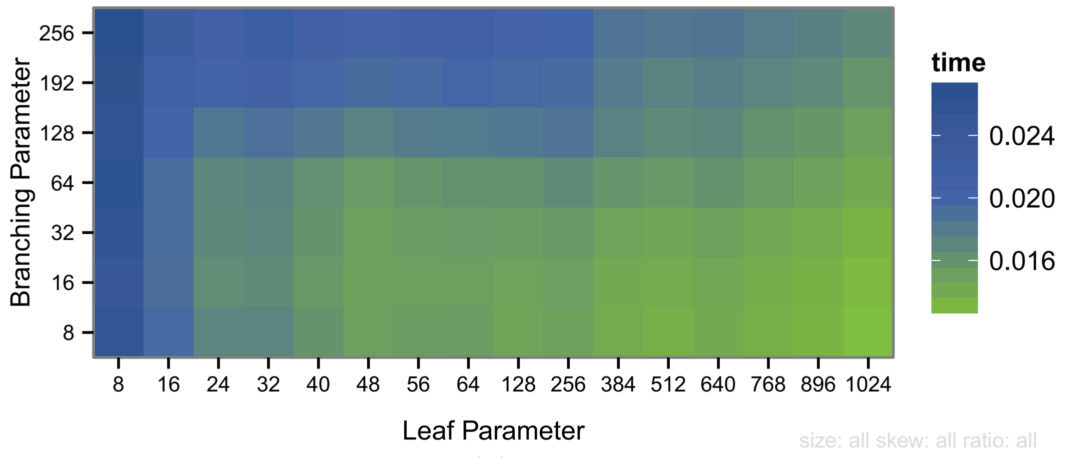
A Comparison to the MCSTL. The implementation of Frias and Singler [FS07] is available via the MCSTL project website[17]. This implementation, however, is not compatible with current implementations of the GCC compiler suite. For a comparison to their implementation, we adjusted their implementation by mixing current `libstdc++` implementations with some old functions that are not included anymore. Furthermore, we had to provide some missing `macros` and specify some additional `includes`. Even though it is near impossible to oversee whether the adjustments break other components of the strongly interconnected `libstdc++`, according to a simple checker, the implementation of the `mcstl::set` seems to work fine. This is confirmed even more in the fact that our measurements show consistent behaviour.

For our experiments, we compared this implementation of the `mcstl::set` to our previously specified B-tree configurations (see Table 7.1). The comparison is executed on the same test framework that we used for the parameter selection; to be precise, we test the performance of the `mcstl::set` for the same set of batch sizes, ratios, and skew values. All tests are performed using one through sixteen cores. At this point, we only present values for the *general* setup.

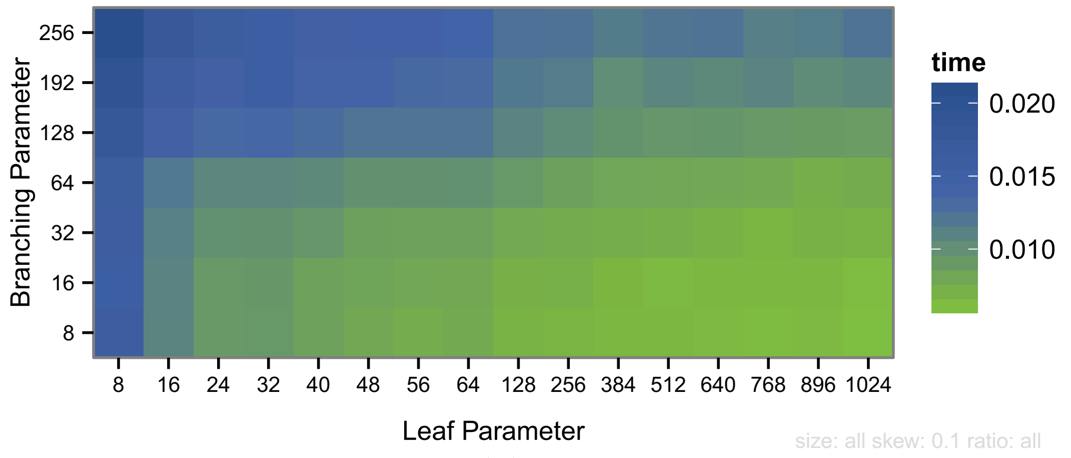
Figure 7.7 compares our B-tree in its *general* setting to the `mcstl::set`. The experiments show our tree to outperform the `mcstl::set` by far. For uniform insertions, our single-threaded variant already has the same performance as the 16 times parallel execution of the `mcstl::set`. In parallel execution, we gain additional speed-up over the compared implementation.

Figure 7.7 also shows the benefit of our method to handle skewed input sequences. Whereas we experience minor performance decreases for some versions for the `mcstl::set`, probably due to some additional load balancing operations, our B-tree can handle

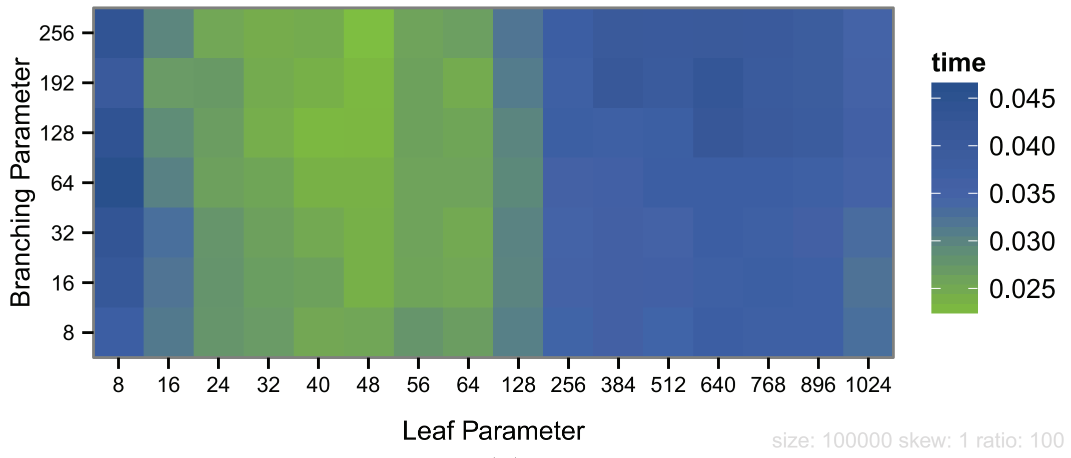
[16]The experiments were not repeated on M2, as they required an extreme amount of processing time.
[17]`http://algo2.iti.kit.edu/singler/mcstl/`



(a) *All Data*



(b) *Skewed*



(c) *Sparse*

Figure 7.6: *Selection of different heatmaps, showing the B-tree performance for a varying set of parameters (leaf/branching). Each plot depicts a different input characterization. The experiments were performed on M1.*

Table 7.1: *Proposed configurations of our B-tree in different scenarios.*

name	skew	density	branching	leaf
Sparse	low	very low	128	48
Dense	all	medium-high	16	512
General	all	all	32	256

skewed sequences more efficiently. The efficient bulk construction helps to handle these usually more difficult sequences of operations, resulting in a slightly faster performance. Only for rather small instances we suffer a small performance impact as our tree fails to exploit enough parallelism for an efficient work distribution.

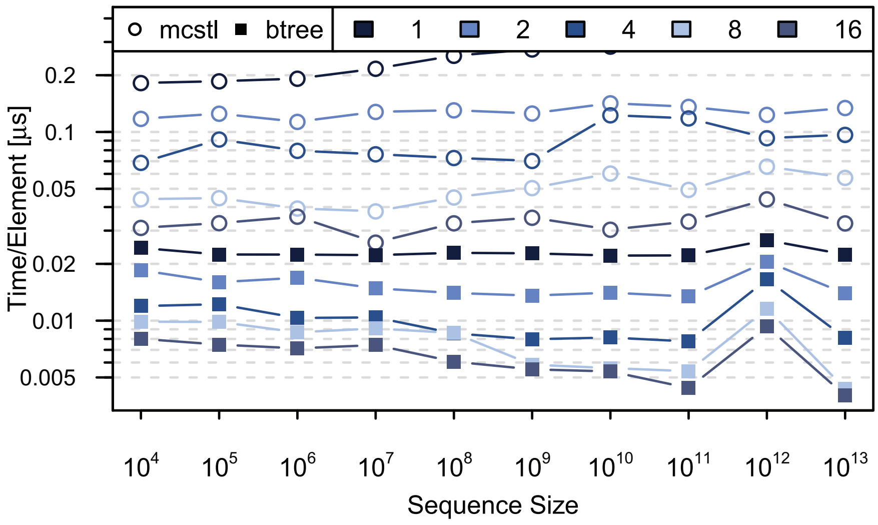
The comparison to the `mcstl::set` validates our implementation of the Pareto-queue and underlines its necessity for an efficient implementation. Bulk operations, as required for the implementation of the paPaSearch algorithm, can be performed efficiently in parallel, far more efficiently than expected from the available implementation of the `mcstl::set`. Building on top of this Pareto-queue implementation, we go on to evaluate our implementation of the paPaSearch algorithm for its viability in the next Section.

7.4.2 Label Sets

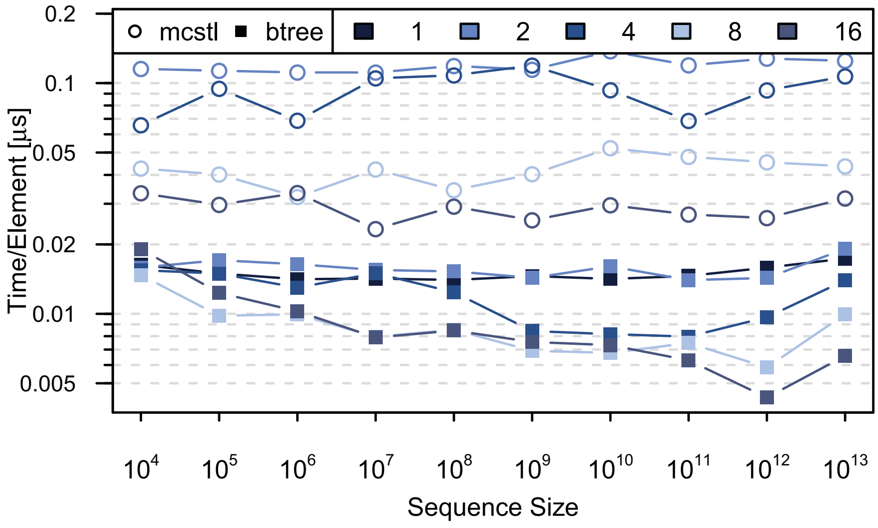
To show the viability of our implementation, we evaluate two major aspects. First, and most important of all, we study the viability of our implementation against a tuned implementation of a sequential algorithm. We validated this implementation by comparing it to other state of the art algorithms. This comparison is published in [EKS14] and achieves better query times then our competitor, even though we operate on a slower CPU (1.83 GHz Intel Core 2 T5600 over a 2.4 GHz Core 2[18]). We found our implementation, referred to by the name `LexClassic`, to be competitive to previous implementations and we use it as a baseline algorithm to compare our new parallel implementation to. The implementation operates in a label-setting fashion by using a lexicographic order for the selection of labels to process. This lexicographic order allows for a quick pruning of new labels by comparing every new label to the lexicographically largest label known for the respective vertex. For the most part, this comparison can be used to filter out dominated labels directly without locating them among the full label-set with respect to their x-coordinate first. This effect can be seen in Figure 7.8. The figure gives strong evidence that we operate at the end of the label-set anyhow. We can see that close to 90 % of all work is done on labels that are among the largest labels with respect to their x-coordinate. This access pattern potentially contributes to the success of the lexicographic approach, as it does not suffer from the usage of a simple array-type label-set[19]. For the labels generated during an instance of the

[18]An exact CPU number is not specified in the paper of Raith and Ehrgott

[19]So far, we are not aware of other label-set types used in the literature.



(a) *Bulk Insertion: Uniform*



(b) *Skewed Bulk Insertion: Insertions among the first 10 percent.*

Figure 7.7: *A direct comparison of the* `mcstl::set` *and our B-tree Pareto-queue for their performance. The plots depict operation times per element for up to 16 threads during the insertion of k elements (sequence size) into a tree of size $10 \cdot k$. The results for skewed insertion (on M2) suffer from our tuning on M1. For M1, we found a more uniform behaviour, showing that our data structure has to be adjusted for cache sizes. See Figure B.5 for the respective plot on M1.*

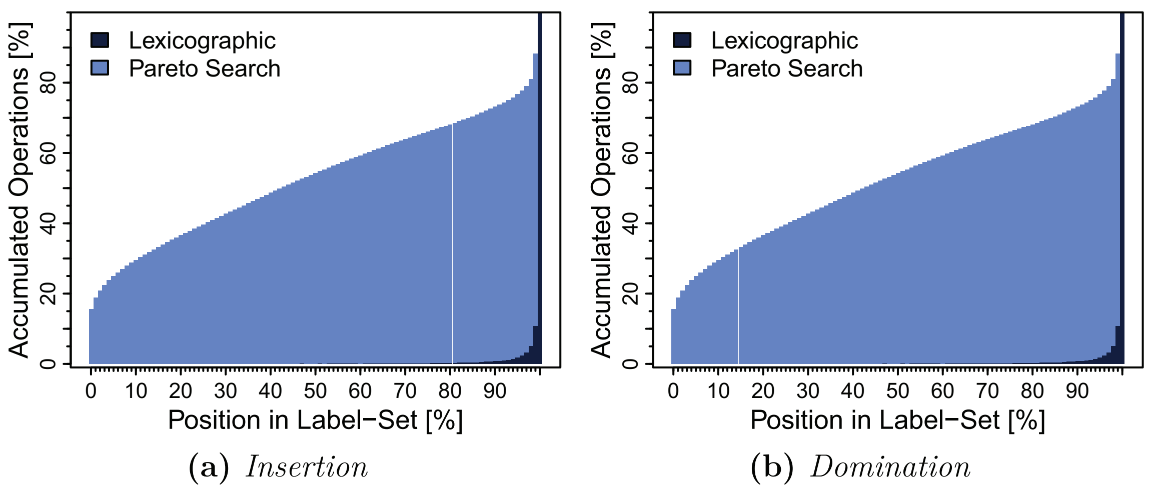


Figure 7.8: *Relative position of operations with respect to the entire label-sets, accumulated averages over all instances of the NY dataset.*

paPaSearch algorithm, we cannot report a similar effect. Rather than operating at the end of the label-sets, the operations are spread out over the entire label-set.

Due to the linear cost for inserting elements randomly into an unbounded array, we propose the use of our B-tree not only for the Pareto-queue but also for the label-sets.

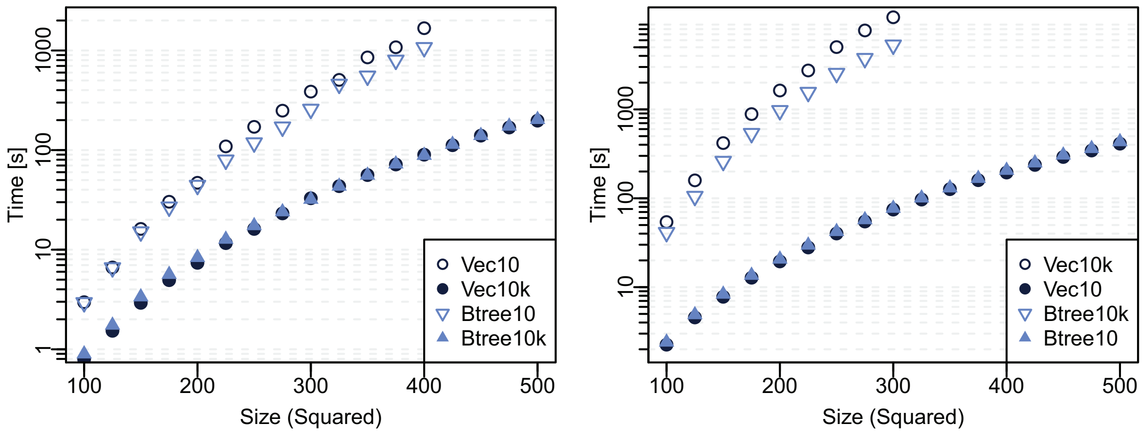
We test the hypothesis on grid graphs. The results are shown in Figure 7.9. The figure shows that our B-tree acts well as label-set, the benefit depends on the setting, though.

What can be seen is a negligible difference for the set of easier problems. We expect this small difference to result from us having to check close to the entire label-set anyway. In more difficult instances that contain a lot of labels, we can see a benefit from switching to our B-tree based label-sets. We therefore propose them as the more general solution that can also be augmented to support parallel operations on the label-sets as well.

7.4.3 Performance

We evaluate the performance of our parallel implementation of the Pareto-optimal shortest path algorithm on a set of different instances. The instances cover both real road networks, provided to us by our competitors, and a set of commonly used grid graphs. In addition, we also tested on some graphs of higher average vertex degree. For our experiments, we configured our implementation to use a the sparse configuration for the label-sets and the dense configuration for the Pareto-queue.

Correlation Influence. Compared to an algorithm operating in lexicographic order, our algorithm has to handle some synchronization, task management and other



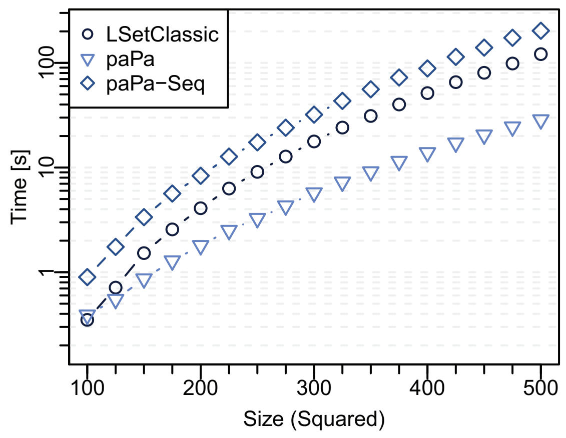
(a) *Uncorrelated (q=0)* **(b)** *Negatively Correlated (q=-0.9)*

Figure 7.9: *Running Times on grid graphs with fixed correlation coefficient and varying size. Juxtaposition of unbounded array (Vec) and B-tree (BTree) as label-sets. Values plotted for maximal cost of* 10 *and* 10 000 *(10k). Missing values not specified due to large memory consumption.*

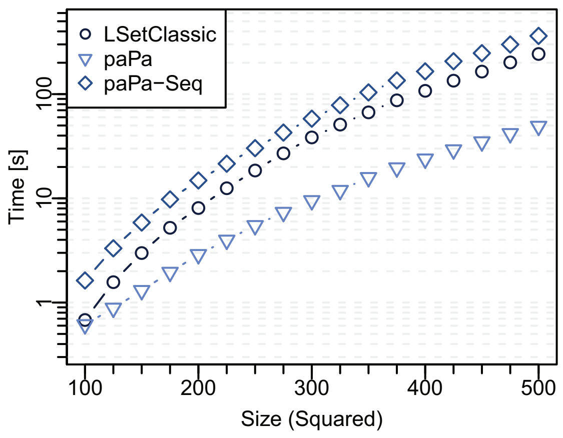
overheads. It is not surprising to find that we cannot outperform the label-setting algorithm in cases where next to no parallelism is available. This situation can be expected in small graphs with highly correlated arc costs. Grid graphs are ideally suited to investigate the influence of vertex count and correlation as we can generate graphs of arbitrary size with preselected correlation coefficients (see Section 6.4.1).

In the direct comparison to the label-setting algorithm, Figure 7.10, the comparison of both strongly depends on the correlation between the two metric components. The figure indicates a minor slowdown introduced over the tuned label-setting algorithm if we execute our parallel implementation on a single thread. When using a series of cores, the speed-up given by the parallel execution results in a large overall improvement. For larger instances and for less correlated arc cost, our algorithm works best. Still, only rather small graphs with not completely correlated arc cost suffice for our algorithm to outperform the label-setting algorithm (compare Figure 7.10c and Figure 7.10c). Already a second of running time is enough for our implementation to outperform the sequential algorithm.

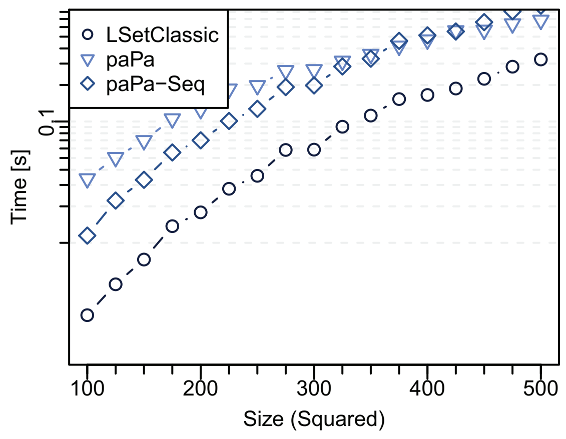
Further instances. One might argue that grid graphs might not hold enough significance. Our experiments on further instances, however, indicate that the results hold true on other instances as well. Especially the road networks provided by Machuca correspond exactly to what we learned to expect from an uncorrelated to negatively correlated grid graph of the respective size. Therefore, we omit the accompanying plots at this point. Turn to Appendix C for further information. A similar thing can be said for sensor networks. The increase in maximal vertex degree by an order of magnitude



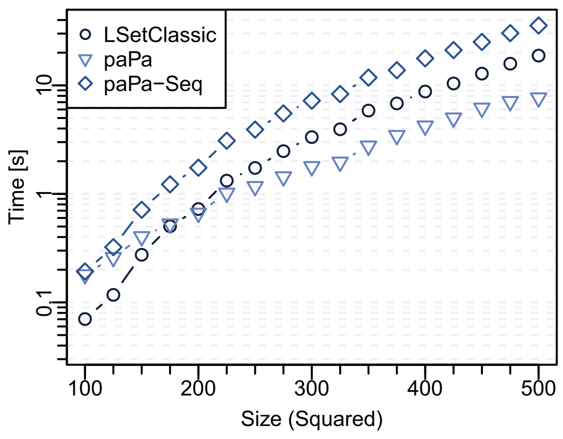
(a) *Negatively Correlated (q=-0.9)*



(b) *Uncorrelated (q=0)*



(c) *Correlated (q=0.9)*

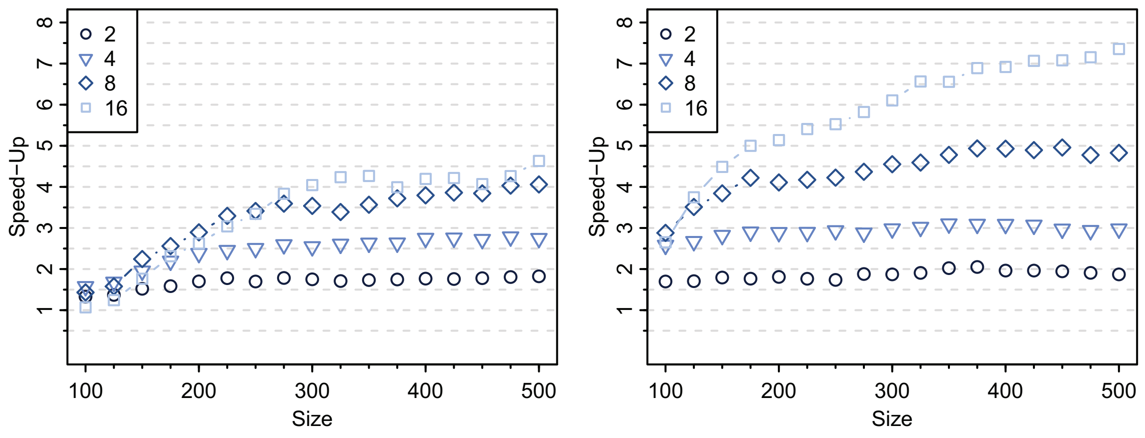


(d) *Correlated (q=0.5)*

Figure 7.10: *Influence of correlation on the speed-up experienced in an execution of the paPaSearch algorithm. Maximal arc cost of ten, parallel execution using 16 threads on M2.*

over the degree in a classical grid graphs, even with the addition of holes that break the grid structure, does not provide a noticeable challenge for our implementation. Again, additional measurements can be found in Appendix C.

Scaling. We have already seen that our B-tree shows a promising scaling behavior, not only with respect to the input size but also with respect to the number of cores used in the operations; remember Figure 7.7. Our full algorithm does not only require our B-tree but relies on further pre-built components as well. To get a better understanding of the scaling potential of our algorithm, we take a closer look at the different steps and their contribution to the overall performance. As previously described, we handle multiple steps during a search: localization of Pareto-optimal labels with candidate



(a) *Correlated (q=0.5)* **(b)** *Negatively Correlated (q=-0.5)*

Figure 7.11: *Speed-up of parallel Pareto-optimal shortest path search over the sequential version on M2. The maximal cost of an arc is set to ten.*

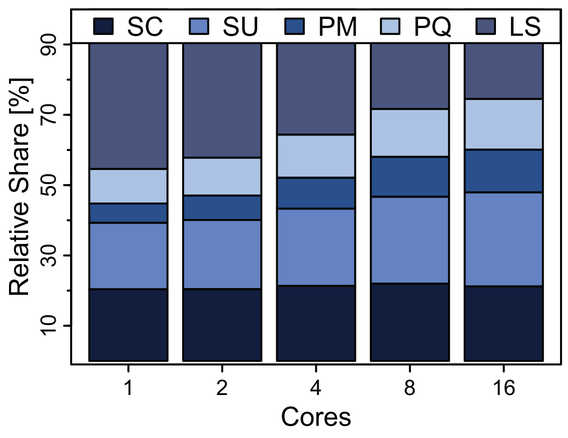
generation, update of label-sets and Pareto-queue, as well as two sorting runs to organize label candidates and update requests. We discuss two different viewpoints of scaling. The first is the general scaling behavior of our full algorithm, presented in Figure 7.11.

While the presented speed-up for the full algorithm is a bit less than the previously seen behavior of our standalone B-tree (compare Figure 7.7), our algorithm still scales surprisingly well. In dependence of the problem size and complexity, we can reach a good efficiency, speeding up the execution by a factor of five on eight cores. On sixteen cores, we can reach close to a factor of eight. Again, our implementation offers a better performance on the machine used for tuning. See Figure C.2 for a comparison.

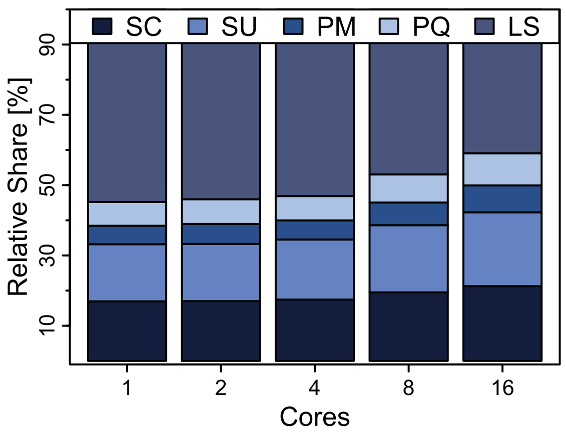
Two aspects of our implementation lend themselves for future optimizations. Taking a look at Figure 7.12, we can identify the pre-built sorting mechanisms as a problem, taking a relatively large share in the overall running time and even increasing their share in the computation when operating on a larger number of cores. The work on the label-sets scales surprisingly well while the share of the Pareto-queue operation increases slightly. Nevertheless, the overall share in the workload is very large. Other storages for label-sets should be investigated.

7.4.4 Delta-Stepping

The most prominent argument against our implementation is the high amount of available parallelism required to achieve the full potential. A direct possibility to increase the amount of available parallelism, as we previously discussed, is to introduce Δ-stepping. In the course of this thesis, we implemented an initial version of this approach that only considers the y-coordinate; this variant integrates well into our



(a) *Small Grid (100)*



(b) *Large Grid (400)*

Figure 7.12: *Relative share of the different components of our paPaSearch implementation of the total running time, measured on uncorrelated grid graphs. We present the relative share to the total running time for sorting candidates (SC) and updates(SU), updating the Pareto-queue (PQ) and label-set (LS), and for discovery of the Pareto-optimal labels (PM). The graphs are uncorrelated grid graphs with a maximum cost of ten.*

chosen data structures and presents a good starting point for initial studies.

We illustrate the potential of Δ-stepping in Figure 7.13 and Figure 7.14. The Figures show a high degree of adjustability with respect to the chosen values for Δ. The number of required iterations needed until the algorithm concludes is reduced very much in relation to the increasing values of Δ. The loss of the label-setting property is also clearly visible, though; in Figure 7.13 we can see that already a small relaxation in relation to the maximum arc cost can increase the number of processed values. A value of ten chosen for Δ nearly doubles the total number of processed labels. Larger values of Δ increase the number of labels even more. This effect translates over to the experienced speed-up achievable via Δ-stepping. We can already infer from Figure 7.13 that increasingly high values of Δ are not beneficial to the overall execution time of our algorithm as the required number of iterations does not decrease in a similar rate as the number of additionally processed labels rises. The amount of available parallelism can only benefit our algorithm as long as it is not completely negated by the additional workload due to unnecessarily relaxed labels.

The results shown in Figure 7.14 indicate that Δ-stepping offers promising results. Compared to the algorithm without Δ-stepping enabled ($\Delta_y = 0$), our modified variant manages to speed-up the query very effectively. The highly correlated inputs gain the most from using Δ-stepping. The same cannot be said for negatively correlated data, though. For a correlation coefficient of -0.9, we barely see any effect. Nevertheless, Δ-stepping offers great potential for further experiments, especially on instances that

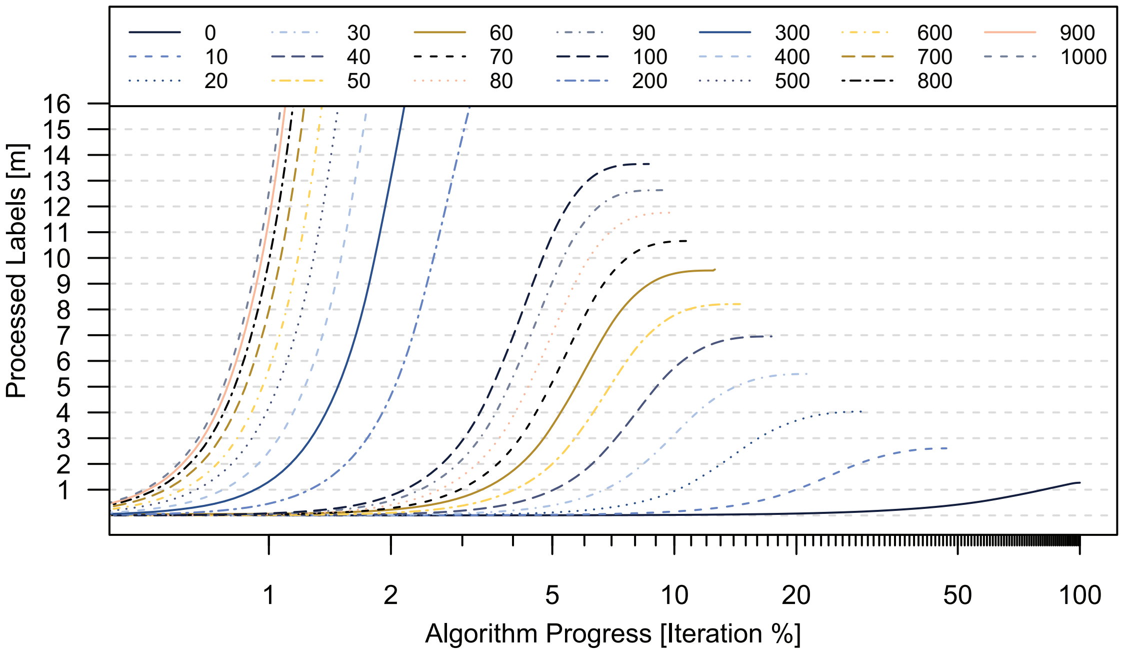
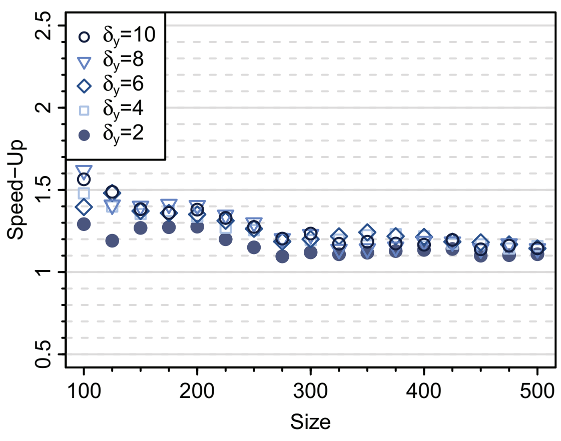


Figure 7.13: *Cumulative count of Pareto-optimal labels processed with and without Δ-stepping enabled over the course of the algorithm's execution for a range of different values of Δ. X-axis scaled to 100 % of the iterations required when running the algorithm without Δ-stepping enabled ($\Delta = 0$). The figure presents data based on a 400 by 400 grid with maximal cost of 1 000 and an uncorrelated metric.*

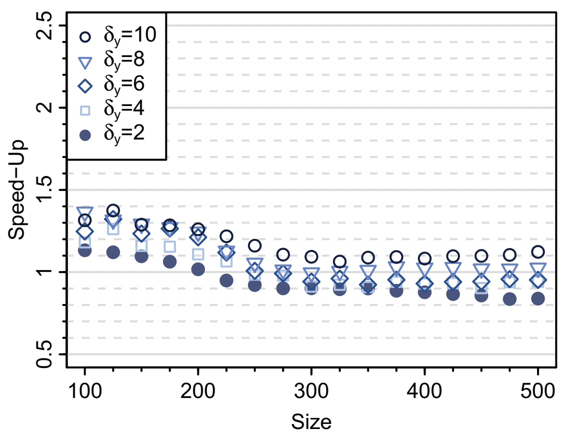
exhibit fewer labels. Especially the question whether it is reasonable to consider the x-coordinate as well needs to be studied.

7.5 Conclusion

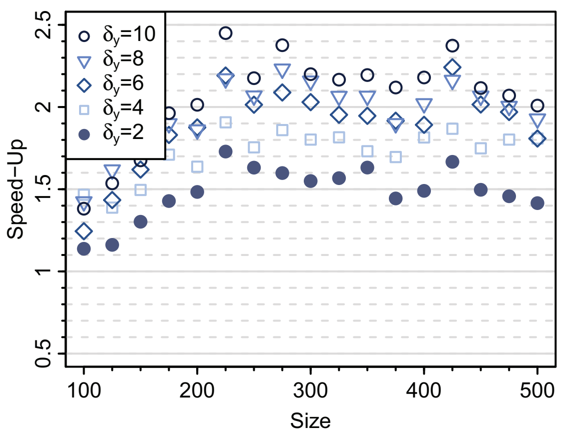
The presented data structures as well as our engineered implementation of paPaSearch have shown that the computation of Pareto-optimal shortest paths can benefit immensely from parallel implementations. We managed to achieve good speed-up with our implementation and are positive that further research into Δ-stepping has the potential for even better results. As an important subproblem of many algorithms (e.g. [DDP+13]), the algorithm has validity of its own. We hope that the Pareto-queue or even the full paPaSearch algorithm can be used to fasten the solutions to bi-criteria problems of different variation as can be found in multi-modal routing for example. Any technique that requires the computation of Pareto-optimal shortest path should be able to benefit from our algorithm. Not only the full algorithm but also our B-tree



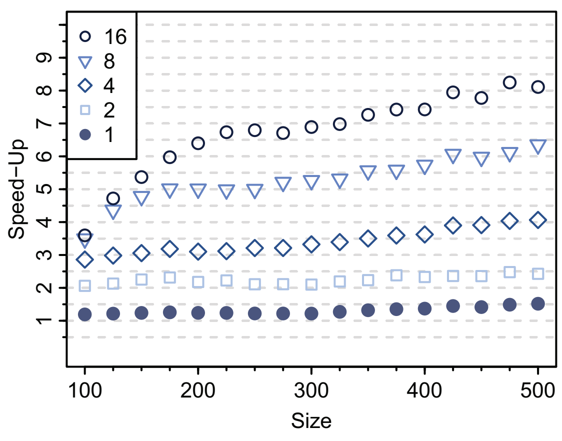
(a) $p = 16, q = 0$



(b) $p = 16, q = -0.9$



(c) $p = 16, q = 0.9$



(d) $\Delta_y = 10, q = 0$

Figure 7.14: *Speed up over our baseline implementation for the Δ-stepping variant using only the y-coordinate on grid graphs.*

can be used as a general purpose data structure. Our comparison to the `mcstl::set` shows its high efficiency and its validity, research into Δ-stepping promises interesting results and even further improvement.

As it stands, however, Pareto-optimal shortest path search is currently still too costly for providing a competitive tool for the computation of alternative routes. Even on rather small graphs, we can experience running times in orders hundreds of seconds, fully disqualifying it for our purposes of supplying a real-time system for alternative routes. The results of Bader et al. [BDGS11] contraindicate the use even further, as the quality of the resulting routes seems to be insufficient.

Chapter 8

8 Engineering the Penalty Method

Penalization is highly configurable, allowing to block vertices or arcs or allows for applying different more sophisticated penalties. However, in a speed-up-technique, it requires some form of dynamic adjustment to handle these penalties efficiently. The implementation requires dynamic changes while still offering query times below two hundred milliseconds. The chapter is based on our own publication "Evolution and Evaluation of the Penalty Method for Alternative Graphs" [KRS13]. Initial ideas were developed in Marcel Radermachers bachelors thesis, supervised by Dennis Schieferdecker and myself. This chapter uses my own implementation, though, which greatly extends and improves upon the publication.

A natural way of finding alternative routes to a shortest path is to look for other paths with a similar length. One possible way to approach this is to assume a delay along a road or path and ask for a route that is optimal under these modified conditions. This can be reasonable if for example we know a certain highway to be prone to traffic jams and we want to avoid it. This simple example describes the basic idea of how the penalty method operates. Next to us, it has been considered by Bader et al. [BDGS11] as well as Paraskevopoulos and Zaroliagis [PZ13].

In general, the method operates in an iterative fashion. Each iteration computes a shortest path P and adds it to a collection. Along this path P, the method applies penalties to road segments and to road segments that are directly connected to P. After the penalization, the method is continues by computing the next shortest path in a now updated model of the road network.

Usually, speed-up techniques assume a static road network (c.f. Chapter 3). As already suggested in [BDGS11], using dynamic adaptations might still be possible. Some techniques have been implemented to support dynamic changes to the network. One of the first was [DW07] that introduces dynamic routing using the ALT algorithm. Their results show large slowdowns depending on the number of changes. The reduction is the greatest if motorways are affected, which is what we expect to happen in our scenario, though.

[PZ13] employ pruning based on the ALT algorithm. Their results validate our concerns regarding the problematic performance: it seems that their approach only works up to medium range queries, as they limit their maximum covered distance for a path to three hundred kilometers. For continental scale networks, they report query times in the order of close to a second, even though they consider only short-range queries.

Recent extensive research into Dynamic Arc-flags [DFV11, DDFV12, DDF14] offers a lot of promising results; turn-around times in the order of multiple seconds prohibit their use in our setting, though.

Similar effects as for ALT have also been reported for HNR [SS07]; the results confirm that variations on important roads are significantly harder to handle than for unimportant roads. Updating a single arc can take around forty milliseconds. For many arcs updated at a time, the amortized cost per arc decreases, though, as some of the workload is shared. Updating a thousand random arcs takes eight seconds, which would be too high. It is unclear, however, how the performance would behave for a series of connected arcs. We would expect the update times to be much lower, but suspect that they might still be too high.

Dynamic Contraction Hierarchies. Instead of using HNR, we could improve the update times by using the dynamic adaptation of CH. The comparisons between Customizeable Contraction Hierarchies (CCH) and CRP, however, make us think that CRP might be the better choice if we consider full customization.

However, updating the data structures is not the only way CH can handle dynamic queries. An alternative method is to modify the query itself to operate on the updated graph. Adding downwards facing arcs around modified road segments on demand, the query can handle a surprising amount of changes to the graph without a significant impact.

This addition of arcs has some negative impact on our approach, though. In our case, we would expect each iteration to traverse the full alternative graph. From all vertices in this graph, we would have to perform an upward search. Even though these upwards searches show a large overlap, as we also see in Chpater 11, we still expect the performance to possibly degrade too much.

These results led us to the decision to not consider the approaches any further. A full exploration of its performance could prove interesting, though, especially in combination with our considerations in Chapter 10.

In this chapter, we present an approach for the penalty method in taking advantage of CRP, which is one of the three-phase techniques; it uses three phases in the sense that the preprocessing phase is itself split into two parts. The first part, c.f. Section 4.2, is completely independent of the metric and only operates on the structure of the graph. While in itself time-consuming, this phase is executed only once and is not affected by the penalty method.

In the second part, CRP computes the metric-dependent information by calculating cost-values for all arcs in the overlay graph. The so-called customization step can be done very quickly and in parallel for many of the affected regions.

8.1 Adjusting the Penalty Method

According to a conversation with one of the authors of [BDGS11], the original implementation computes a set of twenty paths. On this set, they employ a greedy algorithm to select fitting paths for the alternative graph. In an interactive setting, this would allow for a maximal turnaround time of five to ten milliseconds per path.

The techniques presented in [PZ13], while impressive in their achieved quality[1], also raise a few questions as to what extend we can apply them for our scenario. Some of their modifications seem to also be specifically tailored to their interpretation of a road network. The difference to our view does not only prevent a reasonable comparison but also renders some of their techniques inefficient for us. In addition, their algorithm seems to require a high number of computations as they require close to a second of running time for rather short range queries of up to 300 km. Assuming the usual running times of the ALT algorithm, we can deduce that their algorithm requires between twenty and thirty iterations of penalization, as well.

Using a current state-of-the-art algorithm, we are not aware of a way to enable penalization and still offer turnaround times of less than five milliseconds. Therefore, we have to adjust the general process of penalization to operate on as few iterations as possible.

The adjustments we propose for the penalization process can be attributed to three different parts: the *penalization*, the *stopping criterion*, and the *path selection*. In the following sections, we discuss these changes in detail. Another aspect to consider is the running time. In the second part of this chapter, starting in Section 8.3, we discuss how to achieve an efficient implementation of our modified penalty method.

8.1.1 Penalization

For our penalization schemes, we tested a series of different settings. For the most part we kept to the known approaches, though. We distinguish between a multiplicative factor and an additive penalty. The former can apply to both arcs on the path as well as the connected ones, the latter is only used on arcs that are connected to a penalized path. Additionally, we have tested both penalties in an asymmetric setting in which they are scaled with respect to their distance to the source or target. For

[1]Parts of which we contribute to their interpretation of road networks.

the penalization of arcs on the path, we tested both linear scaling penalties along the path($k \cdot \psi \cdot \mu\left(a\right)$, for k denoting the number of times an arc a has been penalized) and exponentially scaling penalties ($\psi^k \cdot \mu\left(a\right)$). For the arcs connected to a path, we applied additive penalties of $\psi_r \cdot \mathcal{D}\left(s,t\right)$, as proposed in [BDGS11]. We also tested asymmetrically scaled penalties, using either $(0.1 + \psi_r \cdot \frac{\mathcal{D}(s,u)}{\mathcal{D}(s,t)})$ or $(0.1 + \psi_r \cdot \frac{\mathcal{D}(v,t)}{\mathcal{D}(s,t)})$, depending on whether an arc is leaving or joining a penalized path. The latter method has been defined this way in [PZ13] but has also been mentioned in [BDGS11]. For all methods, we both tried limiting the increases to a single penalization and to a maximal factor of 1.25[2]. We would like to point out at this point that we are skeptical with respect to the penalization scheme of [PZ13] in a general purpose setting. As [BDGS11] already points out, multiplicative penalties on arcs adjacent to a path tend to be affected by representation choices made within the graph. The choice between a long arc and a series of arcs to represent road geometry influences the applied penalty; the latter one ends up penalizing the graph far less than the former one. This property is undesirable for the penalty method and offers a strong argument against the presented penalization scheme.

Applying Penalties When applying penalties to the graph, multiple methods can be implemented. For example, we could directly apply the penalty and have a way to restore the original arcs afterwards. Another way would be to maintain an additional weight for all arcs. To be able to implement a large variety of different settings, we chose a different way. We apply our penalty in form of flags that indicate for all arcs in what way they contribute to the alternative graph computation.

The limited number of reasonable penalization iterations allows for a simple way to test a wide range of penalization methods. We assume a maximum of thirty-two path computations and store a set of bits with every vertex. This flag is used to indicate its involvement in the different computations. The restriction to thirty-two paths does not impose an actual limitation in our view, as a reasonable penalization strategy should require a fewer iterations anyhow[3].

Using the built-in `popcount` instruction, we are able to determine the number of times an arc (u,v) has been part of a shortest path (`popcount` $(u\&v)$) or an ingoing or outgoing arc (`popcount` $(u\& \sim v)$). If we ignore asymmetric rejoin penalties for a moment, the number of times that an arc should have been penalized by is `popcount` $(u\hat{}v)$[4]. For example, given the flags `11011` and `10100` for u and v respectively, we can deduct that both of them have been part of the same path once. In addition, they have been connected to each other four times, three times of which v having been connected to the shortest path, one of which u was a neighbor to the shortest path. Keeping track of

[2]In this case, we also took care to correct for additive penalties.

[3]This method ignores potential parallel arcs. These small hops are undesirable in an alternative graphs anyhow, though.

[4]&, $\sim$, and ^ describe the unary operators `and`, `not`, and `xor`

the distance between all vertices and the source and target, we are able to implement a wide range of different penalties that can be applied.

After extensive testing of various versions, we decided on an exponentially growing version of the path penalty($\psi^k \mu(a)$) and an asymmetric additive rejoin penalty ($k\psi_r \frac{\mathcal{D}(s,a)}{\mathcal{D}(s,t)}$ and $k\psi_r \frac{\mathcal{D}(a,t)}{\mathcal{D}(s,t)}$ respectively); k specified the number of penalizations applied to the arc a.

To avoid over-penalization of arcs, we use limits on the maximal increase on a single arc, as already suggested in [BDGS11].

8.1.2 A New Stopping Criterion

As we have already mentioned, we have to limit the number of iterations. Rather than limiting iterations by fixed number, we consider series of different parameters, all of which can toggle the end of our algorithm. The easiest criterion to discuss might be the number of decision-arcs used in both [BDGS11] and [PZ13]. Every new segment that we include in the alternative graph constitutes a decision-arc. As a result, we could stop our algorithm the moment the limit of decision-arcs is reached. It does not prove a useful criterion, though. Adding more arcs in combination with thin-out methods results in a better alternative graph and also provides the correct number of decision-arcs.

The second parameter that we considered is a penalization limit on the different arcs. Given the typical limits for bounded stretch (BS), we can introduce a natural limit on the number of times it is reasonable to penalize an arc. Using the limit ϵ for BS, we also stop our algorithm if all arcs have been penalized at least $\left\lceil \log_\psi (1+\epsilon) \right\rceil + k$ times. This bound reflects our decision to not include sub-paths of high stretch under the assumption of our chosen exponentially growing path penalty (ψ, see Section 8.1.1). A stronger penalization would only find paths of higher than the maximally allowed BS afterwards. The addition of the k to the equation corrects for effects in which parts of the shortest path are unavoidable (e.g. bridges). Close to these, the rejoin penalty (ψ_r) influences the penalty along the path, resulting in scenarios in which a stronger penalization might be reasonable. To be exact, given additional penalties on the adjacent arcs, a penalization of a factor larger than $(1+\epsilon)$ can still result in a reasonable path as we have to compensate for ψ_r along the alternative path.

The final aspect considered for our stopping criterion is the length of the path found in the current iteration. We study it both in its penalized and unpenalized length: in case its (unpenalized) length exceeds the length of $\Pi_{s,t}$ by a factor of more than $(\psi \cdot \epsilon)$, we stop our algorithm. Additionally, we check whether its penalized length exceeds $(1+\epsilon) \cdot \psi + 2 \cdot \psi_r$, and stop the algorithm if it does. Again, both criteria are motivated by the limits on bounded stretch, and by extension also the considerations regarding the average distance [BDGS11].

Algorithm 8.1 Penalty Method. This simplified illustration does not show the limited number of penalizations to a single arc.

Input: A graph $G(V, A)$ with associated metric μ, a source s and a target t
Output: An alternative graph $\mathcal{H}$

```
$\mathcal{H} := P := \Pi_{s,t}$
$D = \mathcal{L}(P)$
$G' := applyPenalties(G)$
$P' = \Pi_{s,t}(G')$
$s = 0$                                   ▷ compare bounded stretch and decision-arcs
while $s < 10$ and $\mathcal{L}_G(P') \leq (1+\epsilon) \cdot D$ and $\mathcal{L}_{G'}(P') \leq (\psi \cdot (1+\epsilon) + 2\psi_r) \cdot D$ do
    if isFeasible( $\Pi_{s,t}(G')$ ) then
        $s = s + 1$
        $\mathcal{H} := \mathcal{H} \cup \Pi_{s,t}(G')$              ▷ include good paths in $\mathcal{H}$
    end if
    $G' := applyPenalties(G')$            ▷ further penalize the graph
    $P' = \Pi_{s,t}(G')$
end while
return $\mathcal{H}$
```

8.1.3 Selecting Feasible Paths

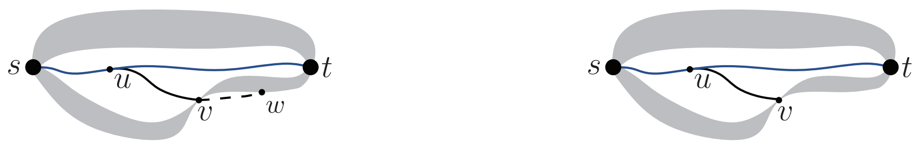
As previously discussed, the usual approach to create an alternative graph initially computes a series of penalized paths. Afterwards, a selection process is applied to tries and optimize $\mathcal{D}_{tot}$ and $\mathcal{D}_{avg}$. From the selected paths, all information – as far as we can tell – is added to the alternative graph, potentially to be removed by thin-out methods. In accordance to the quality criteria of Abraham et al. (see Definition 5.3), we propose the following method.

Instead of adding full paths to the alternative graph, we perform an evaluation and only keep those parts that fulfil the criteria in Definition 5.4. Every path we find is divided into continuous segments. These segments are distinct from the current alternative graph, except for their first and last vertex. We evaluate them for their quality according to limited sharing and bounded stretch. We omit local optimality in these considerations due to its incompatibilities with the penalization process.

We consider only segments that offer an alternative to a sub-path which is at least five percent of the shortest path in length. If, at the same time, their bounded stretch is less than twenty-five percent, we can add them to the alternative graph[5].

The mathematical formulation of our criteria can be found Definition 8.1.

[5]In a commercial system, one might have to consider allowing longer detours in some special scenarios like routes in Berlin for which no alternative routes would exist in comparison to the inner city highway.



(a) *Intersection* **(b)** *Join*

Figure 8.1: *Consideration of a path to use in an alternative graph. The left side shows a segment that leaves the shortest path at u, crosses the alternative graph at v and joins it at w. On the right side, the path joins the corridor directly at v. By considering all segments between corridor vertices individually ((u, v) and (v, w) are treated separately of each other), we might end up keeping only the first part of the deviation for our alternative graph. The latter segment is discarded if it does not constitute a viable deviation on its own. As a result, both paths, though different, could end up contributing the same to the alternative graph.*

Definition 8.1 (Feasible Segment)**.** *A path segment, given as a sequence of arcs $\langle(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\rangle$ in relation to an alternative graph $\mathcal{H}$ with $v_0 \in \mathcal{H}$, $v_k \in \mathcal{H}$, and $v_j \notin \mathcal{H}$, $\forall j \in [1, k-1]$[6] is feasible if the following conditions hold:*

$$\begin{aligned} \alpha \cdot \mathcal{D}(s,t) &\leq \mathcal{D}_{\mathcal{H}}(v_0, v_k) && (8.1) \\ \mathcal{L}(v_0, \ldots, v_k) &\leq (1+\epsilon) \cdot \mathcal{D}_{\mathcal{H}}(v_0, v_k) && (8.2) \end{aligned}$$

The value for α describes the desire for limited sharing and is identical to the parameter in Definition 5.4. In the same way, ϵ is used to limit the bounded stretch.

We illustrate the concept of our Definition 8.1 in Figure 8.1. The main reason for the common segments is that we want to provide a reasonable path to the target from the end of every segment. In a common vertex, there is always a reasonable way to switch from one path to another.

Adding full paths to the alternative graph, rather than feasible segments, unnecessarily increases the number of decision arcs. By using only viable segments, we avoid the addition of unreasonably long and very minor detours. This restriction, however, reduces the value of $\mathcal{D}_{tot}$ that we can achieve. It is rather easy (compare Figure 8.4), however, to construct alternative graphs that contain unreasonable arcs which can survive both global and local thin-out. The problematic part in the thin-out methods is that they only operate on the alternative graph alone. In our comparison, we consider the full set of paths in the calculations and, as a result, prevent the insertion of such arcs in the first place.

[6] (v_0, v_1) is also a valid segment, as long as the arc itself is not part of $\mathcal{H}$.

Table 8.1: *Impact on the average values for $\mathcal{D}_{tot}$ and $\mathcal{D}_{avg}$ in dependence of different optimizations in our technique. Values specified consider random queries on the Berlin network (G8). We use an exponentially increasing scale along a path (factor 1.1) and asymmetric additive penalties (factor 0.002 $\mathcal{D}(s,t)$).*
***BS**: stops if no further penalty was applied (all arcs penalized fully), **limited sharing (LS)** adds only segments ignores short segments, **LI** limits the number increases on a single arc, μ stops the search if the last path without penalties is too long, μ_π does the same on the penalized path.*

						w/o thin-out				thin-out			
set-up	BS	LS	LI	μ	μ_π	$\mathcal{D}_{tot}$	$\mathcal{D}_{avg}$	#arcs	# alt	$\mathcal{D}_{tot}$	$\mathcal{D}_{avg}$	#arcs	# alt
1	-	-	-	-	-	5.23	1.06	24.39	7.16	4.83	1.06	21.96	6.94
2	+	-	-	-	-	4.90	1.06	21.87	6.81	4.57	1.05	20.44	6.73
3	+	+	-	-	-	4.72	1.07	16.74	6.57	4.39	1.05	15.46	6.49
4	+	+	+	-	-	4.42	1.05	14.83	6.79	4.27	1.05	14.43	6.71
5	+	+	+	+	-	4.14	1.05	13.01	6.26	4.04	1.05	12.73	6.20
6	+	+	+	+	+	3.79	1.05	10.65	5.60	3.72	1.05	10.49	5.57

As a side note, our incremental approach allows to stop the algorithm early if a certain quality has been achieved. In our experiments, we always consider the full run, according to the previously mentioned stopping criterion, though.

8.2 Experimental Evaluation – Modifications

After the discussion of our modifications, we first take a look at how they affect the quality of the results. These considerations are mostly independent of running time of a single customization.

8.2.1 Parameterization Influences.

We have proposed a series of different ways to restrict the alternative graph in both the number of iterations used and the segments that are added to it. None of these methods comes free, as can be seen in Table 8.1. In combination with Figure 8.2, we can easily argue that the proposed changes still benefit our calculations. The reduction of $\mathcal{D}_{tot}$ by more than one over all the different set-ups can be justified by the increase in quality. While the last set-up does not exhibit any different behavior in our example graph, we found it to produce the best alternative graphs, based on visual appearance.

Without thin-out, our current set-up 6 may generate graphs with more than ten decision arcs. Using thin-out, set-up 6 stays within the limit, though.

The discussion based on the Berlin network is too restricted. Therefore, we also studied the effects of all the set-up variants on the western European road network

(G1). In general, the observed changes remain the same. More restrictive set-ups lessen the quality ($\mathcal{D}_{tot}$) in a similar way. Therefore, the proposed changes remain valid.

Figure 8.3 indicates that our first restriction has the largest impact. As long as we restrict the calculations to viable segments of reasonable length, the further restrictions have only a minor impact on the quality of the alternative graph. For most of the queries, we can barely report a negative effect. Only extremely long queries can be said to degrade in their quality. We notice a drop in quality from 4.5 to around 3.8 on average for the Dijkstra rank of twenty-four. On the other hand, the positive effect on the number of decision arcs cannot be neglected.

In comparison to the values specified by Bader et al., we reach $\mathcal{D}_{tot} = 3.17$ ($\mathcal{D}_{avg} = 1.04$, # arcs = 7.2) on average compared to their value of $\mathcal{D}_{tot} = 3.34$ on the same graph. At the same time, our algorithm requires less than eight penalization iterations on average. While the values are a bit less, we think that the small decrease is well within reasonable limits. If we perform a fixed number of twenty iterations, we can reach their values, as well.

8.2.2 Quality Considerations

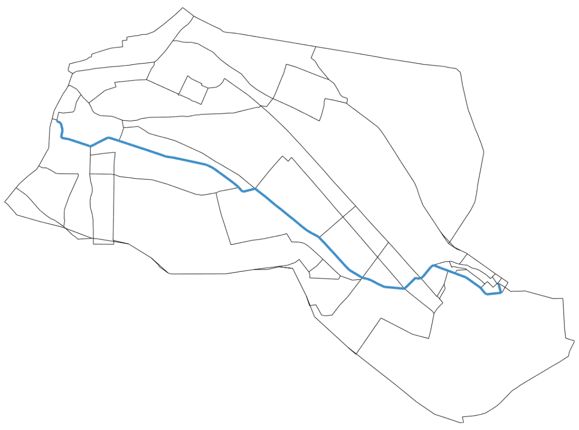
Given the proposed adjustments to the alternative graph calculation, we need to discuss their implications. A problematic finding in the penalty method, at least in our view, is the fact that, as illustrated in Figure 8.4, local detours of poor quality can increase $\mathcal{D}_{tot}$. Especially in subsections of the paths, this can become a problem. This setting is important to consider, as detours to a detour – even though reasonable when considered as a whole – might be considered as too long, locally. Given that a path can be evaluated only with respect to the other arcs in the alternative graph, we could even generate local detours of near infinite relative length.

Therefore, it is difficult to fairly compare different approaches to the alternative graph problem, as one can optimize for $\mathcal{D}_{tot}$ using such detours. In ignoring some connections, it is possible to increase the values of $\mathcal{D}_{tot}$ in a way that may be unreasonable.

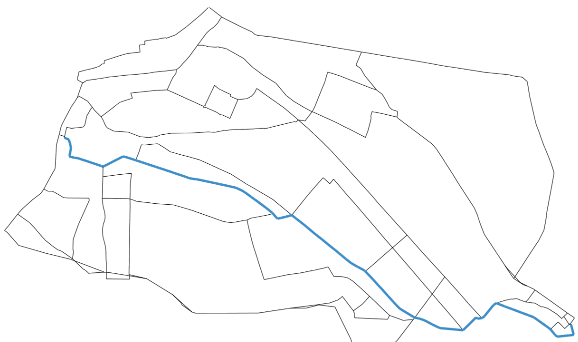
In a direct comparison of the results of [BDGS11] and [PZ13], the improved quality is most noticeable. As we see later on, the method we chose cannot produce a similar improvement. In the following discussion, we provide arguments that justify why our approach is still reasonable, even though we only reproduce the results of [BDGS11].

The main reason for the differences in quality of our implementation and the one in [PZ13] is rooted in the different models of road networks. After a conversation with the authors, we found that [PZ13] uses a model which we deem too simplistic for our purpose. The authors consider only three different road categories. In comparison, our network G1, which has been claimed to be realistic in [DGPW13], uses fifteen categories. They set highways to a travel speed of $130\frac{km}{h}$, trunk roads[7] to a travel

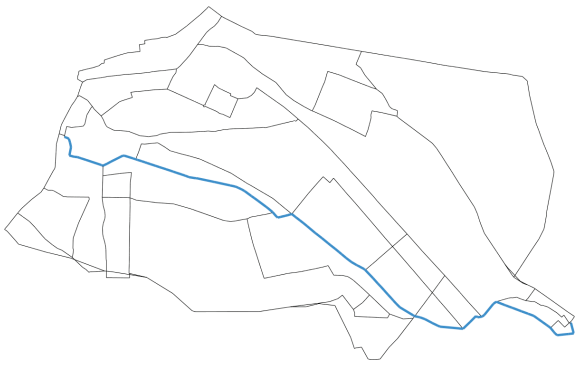
[7]According to the OpenStreetMap Wiki, trunk is a reference to the second most important road category of a country.



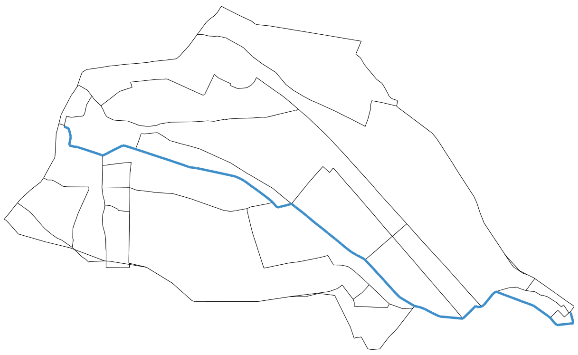
(a) *Set-up 1*



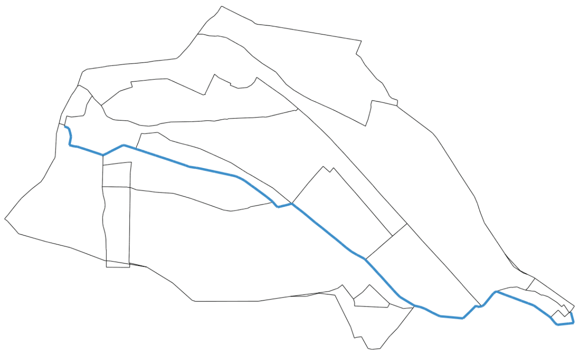
(b) *Set-up 2*



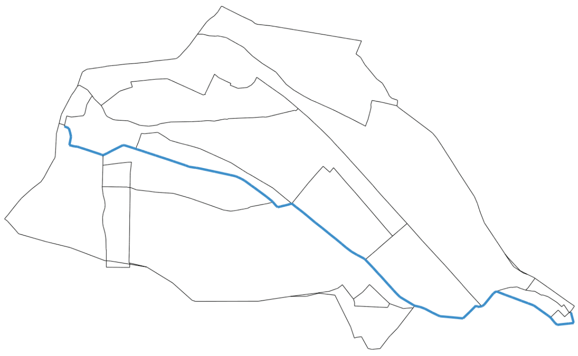
(c) *Set-up 3*



(d) *Set-up 4*



(e) *Set-up 5*



(f) *Set-up 6*

Figure 8.2: *Exemplary evolution over the different set-ups of Table 8.1 for a random route in the Berlin road network.*

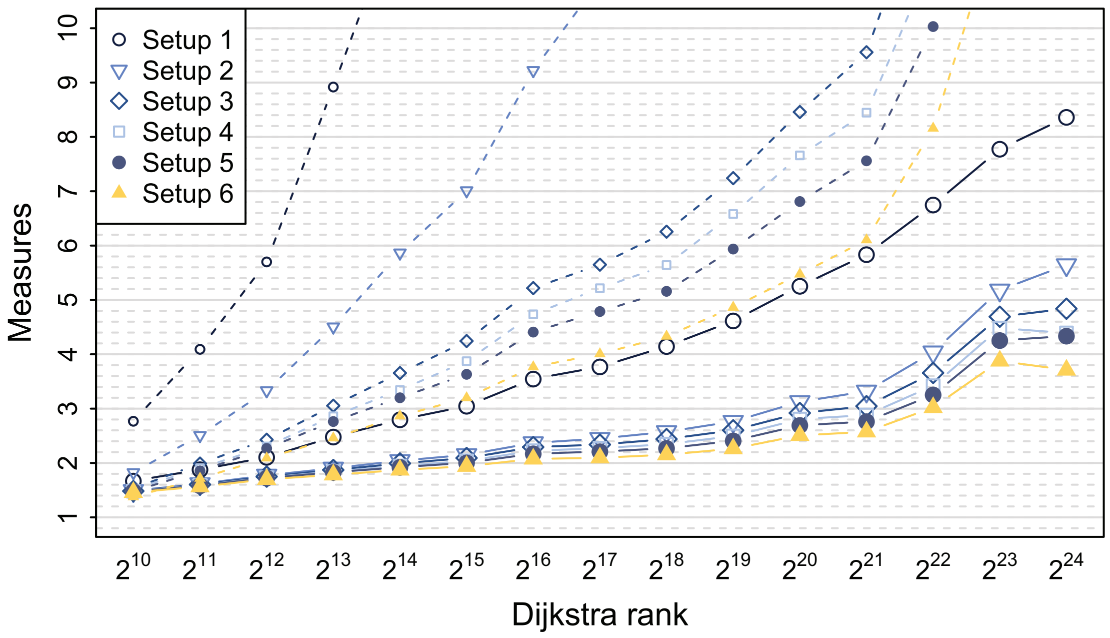


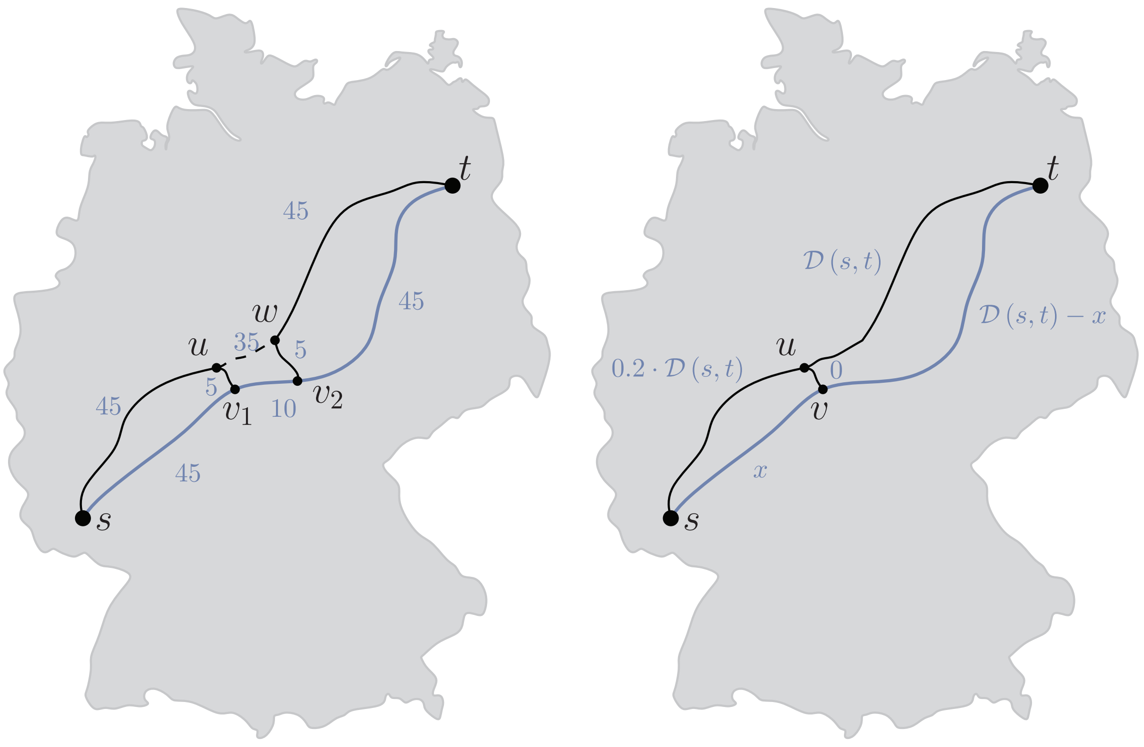
Figure 8.3: *Development of $\mathcal{D}_{tot}$ (dashed) and the number of decision arcs (solid) over the different set-ups. Parameters: $\psi = 1.1$, $\psi_r = 0.003$ using multi-increase for both and asymmetric penalization for ψ_r.*

speed of $110\frac{km}{h}$, and an average travel speed of $50\frac{km}{h}$ to all other roads, including field roads. We suspect that this model may introduce additional alternative routes where none exist. For example in cities or badly connected areas, the interpretation translates into a distance metric. In cities, however, the presence of many paths will most likely result in a high number of alternative routes. At the same time, their model equalizes longer motorways and shorter trunk roads in other parts of the graph. Therefore, we think that the models cannot be compared against one-another.

To prove our approach viable nonetheless, we offer measurements for upper bounds: Our upper bounds present a superposition of all paths that we find during the iterations of our algorithm. In combination, this set gives an impression on the values that our approach can produce in the best case. We construct these modified alternative graphs by ignoring all limits on the number of decision edges and do not performing any thin-out. As a selection method reports a subset of this information, the graph provides an upper bound on the achievable $\mathcal{D}_{tot}$. An example of such a graph can be found in the appendix in Figure D.3.

Evaluating these upper bounds, we can see a large discrepancy between the values in [PZ13] and the results in Table 8.2. Only for G8, our results are comparable to the values presented by Paraskevopoulos and Zaroliagis[8].

[8]Their graph has been obtained from proprietary source, though. Therefore, we cannot compare our



(a) *Concrete example* **(b)** *Theoretic construct*

Figure 8.4: *Illustration of a long local deviation along (u, w) and the influence on $\mathcal{D}_{tot}$. The graph using (u, w), ignoring (u, v_1) and (v_2, w) has $\mathcal{D}_{tot}$ equal to two. The other way around, we get $\mathcal{D}_{tot}$ equal to* 1.95. *The second example is more of a theoretical nature. It illustrates a scenario in which we can essentially generate an infinite local $(x \to 0)$ detour that, in terms of $\mathcal{D}_{tot}$, remains a better choice throughout the full process.*

Using a distance metric on G8, we can increase the upper bound for $\mathcal{D}_{tot}$ to 7.8. In a comparison with their value (4.16 in [PZ13]), their method reaches fifty-three percent of the upper bound. Applying this same reduction to the other inputs, our algorithm provides similar results on all graphs. Therefore, we compare ourselves to the quality values presented in [BDGS11], even though they are lower than the ones in [PZ13]. The comparison to other methods, both from the literature and our own methods, follow in Chapter 12.

algorithm on an identical data set.

Table 8.2: *Upper bounds, consisting of all paths found within twenty penalization iterations that are not more than twenty-five percent longer than the shortest path. This setting is comparable to the one chosen in [PZ13]. Values are calculated using $\psi = 0.1$ and $\psi_r = 0.1$ and the penalization described by [PZ13], using the same type of queries (300 km limit). The comparison on the Berlin road network (G8) does not use an identical graph source. The graph in [PZ13] for this road network is extracted from proprietary data. The number of first alternatives is specified without checks for LO. All reference values marked by an asterisk refer to different data sources or ones that were not defined closely enough in [PZ13].*

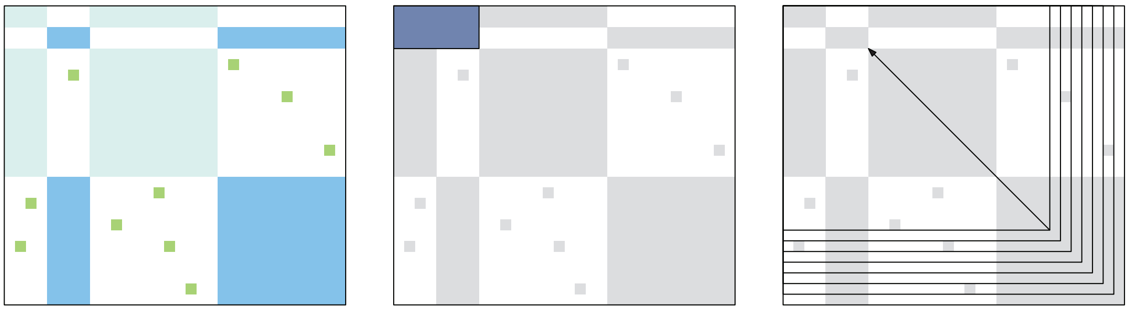
	w/o thin-out				with thin out				
ID	$\mathcal{D}_{tot}$	$\mathcal{D}_{avg}$	arcs	# alt	$\mathcal{D}_{tot}$	$\mathcal{D}_{avg}$	arcs	# alt	[PZ13]
G1	3.44	1.05	77.26	2.87	**3.00**	1.04	64.07	2.78	***5.24**
G2	3.64	1.05	83.06	3.37	**3.19**	1.04	70.41	3.26	***5.46**
G3	3.69	1.04	101.26	3.27	**3.22**	1.04	80.23	3.17	***5.24**
G4	4.63	1.05	125.03	5.13	**4.17**	1.05	103.37	5.00	**5.46**
G5	3.37	1.04	64.90	2.56	2.94	1.04	50.61	2.48	-
G6	4.28	1.05	95.02	4.34	3.82	1.05	77.14	4.22	-
G7	3.64	1.05	55.27	3.60	3.21	1.04	43.34	3.51	-
G8	4.14	1.04	34.26	4.36	**3.72**	1.04	26.31	4.28	***4.23**
G9	3.16	1.05	25.79	2.48	**2.77**	1.04	19.69	2.44	**5.19**

8.3 Customizing a Single Query

So far, we have seen that we can perform an iterative approach with an on-line stopping criterion. We already stated that our implementation is using CRP. We made this choice, as CRP is used in commercial systems already (e.g. Bing Maps) and offers some benefits in its structure. The distance matrices offer good cache locality and as a separator-based approach, we can employ parallelism on independent parts.

In the following parts of this chapter, we discuss how we can achieve reasonable performance on CRP, using parallelism, SSE, and some modifications to the general structure of CRP.

Cell Update Techniques. Before getting into the actual penalization, we have to explore the update techniques we use for customization. [DGPW13] already considers different techniques. Their road network model differs from ours, though; their implementation considers a turn-aware model. We chose to use the simple model over the one incorporating turn-cost (compare [DGPW11, DGPW13, DW13]) to allow for a better comparability to the other alternative route techniques. This choice is validated by the fact that the only turn-cost available to us are a penalization of u-turns. Since we use a different model, we cannot simply rely on their measurements for our selection.



(a) *Input Format* **(b)** *Relevant Result* **(c)** *Algorithm Progress*

Figure 8.5: *Schematic of our update algorithm based on the Floyd-Warshall algorithm.*

Since we update cells very often, optimal customization performance is critical. Next to the previously known techniques, we also added an additional one to the repertoire of available methods based on the Floyd-Warhsall algorithm. For our modified version, we make use of the vertex mapping already present for the CRP algorithm (c.f. Figure 8.5a).

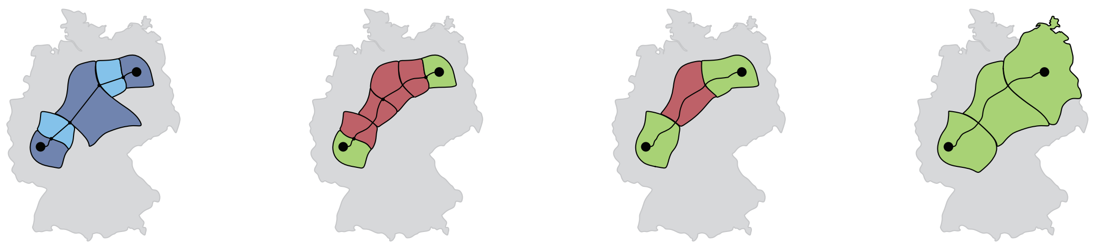
When we compute the distance matrix, most parts of the information are irrelevant to our progress. The vertex labelling we described puts the desired results in the top left corner of the matrix (c.f. Figure 8.5b). Our optimized version of the Floyd-Warshall algorithm is tailored to this setting. We make use of this fact in progressing (c.f. Figure 8.5c) from the bottom right corner of the matrix to the top left. As long as the considered via-vertex is not part of the output, we discard its respective row and column of the matrix. This way, we reduce the future workload in every iteration.

The correctness of this approach follows from a contraction argument: our implementation implicitly contracts all non-border vertices. Among the remaining vertices, we use the usual implementation of the Floyd-Warshall algorithm.

Using the distance matrices, the algorithm can be implemented by limiting i and j (see Algorithm 2.4) to the maximum of $|\mathcal{C}|, k$ in every iteration. This does not affect the asymptotic running time, which is still $\mathcal{O}(n^3)$, but the actual running time. For further speed-up, we use vector instructions.

8.3.1 Omitting Origin and Destination Cells.

The cells that require an update can be calculated via a simple traversal of the extracted path. We trigger an update for every cell on the bottom level that contains a vertex from the shortest path. These updates can be performed completely independently and in parallel. Iteratively, we mark the parent cells of the just updated cells and continue in the same way until all updates have been performed. Pseudo-code for the algorithm can be found in the Appendix (Algorithm D.1).



(a) *The path* (b) *Level zero* (c) *Level one* (d) *Level two*

Figure 8.6: *Illustration of the steps from a shortest path between a source and a target in CRP and the results of our dirty-cell computation with a maximal considered level of one in a three level CRP. The levels are marked zero, one, and two. Dirty cells are marked in red, non-dirty cells are marked in green.*

In this process, we can omit updating the cells containing the source and the target (see Figure 8.6) at the cost of a minor modification to the query. The reasoning in this approach is, that for the most part a query does not cross a cell multiple times. Imagine the search space that grows in a circle around the source. Omitting these two cells on every level (compare Figure 8.6), we enter a trade-off between the performance of the query and the customization cost.

To preserve correctness, we need to descend both into the direction of the source and the target. Figure 8.7 gives an example of a query that cannot use the usual level as the full cell containing the source has not been updated. By descending into both the direction of the source and the target, we prevent the algorithm from using a non-updated cell.

The query is slowed, however, if we have to cross a cell. These cases, however, are rare and do not affect the query in a noticeable way. The rather large cost for updating a cell easily compensates for the small additional query overhead.

Next to this optimization, we can even go one step further and omit full levels in the query and update process.

Dynamic Max Level Selection. CRP itself is optimized to allow long-range queries. For short to mid-range shortest paths, higher levels will not be touched. They are, however, necessary for the optimal performance on transcontinental shortest-path queries. The correctness of the algorithm itself, however, does not require to use all computed levels. Instead, it can be restricted to any subset of them. Due to the iterative process of path computation and penalization, we can utilize knowledge about the shortest path itself in our algorithm. This is substantially different from a usual shortest-path query in which the information only becomes apparent when the computation is finished. In our case, we can use the structure of the shortest path to adapt the behavior of our algorithm during its execution. By analysing the number of hops of an average query of a given Dijkstra rank, we are able to select an appropriate

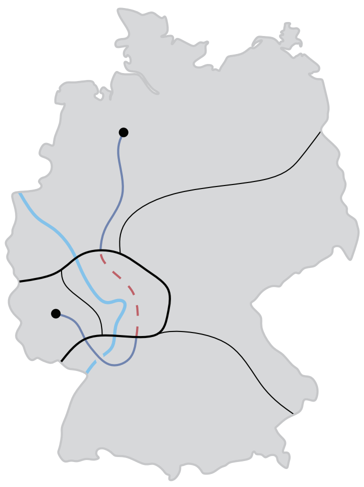


Figure 8.7: *Illustration of the reasoning behind query adjustment. Due to natural borders (e.g. a clear blue river) or other reasons, a shortest path might visit a cell (thick black line) more than once, i.e. return back to the cell. As part of a cell that includes the source or the target, the dashed part of the shortest path requires a shortcut that has not been updated properly. As a result, we have to consider both source and target for the selection of the appropriate level on which we have to process a vertex.*

maximal level.

Integrating such a maximal level only requires a minor adjustment. Whenever we decide on which level a vertex belongs to, we calculate the minimum of the usual level and our new maximal level. This works as every border vertex on level i is also a border vertex on level $i - 1$. For the initial shortest-path query, we utilize all of the CRP levels. This way, we ensure the best performance for the query.

During different penalization iterations, we consider only cells within the first few levels. Again, see Figure 8.6, for an illustration of the benefits of this approach. It would be interesting to further exploit this trade-off by omitting not entire levels but maybe only cells within a level. Descending into some cells, we could make use of unaffected cells on partially invalidated levels.

Adjusted Cell Sizes and Shadow Level. To compensate for the bad trade-off choices available with the current partition, we also tested some variations with respect to the cell sizes. As we show in the experiments, some cells have severely higher customization cost than the average cell. To deal with these cells better, we split the largest ones on the topmost levels into smaller cells. Even though this increases query time, it reduces the cost for updating at the same time. In addition to the adjusted cell sizes, we explicitly store the shadow-level with our CRP structures. This requires additional memory but also reduces the work required for customization.

Cleaning Up. Due to the modifications we make to CRP during the penalization, we have to do some maintenance work after the query. We do this by simply keeping a list of dirty cells for every level. When we modify a cell for the first time, we simply replace the pointer in the data structure that points to the (now) modified cell. The original pointer is remembered with the list of dirty cells. When our algorithm concludes, we simply traverse this list and restore the original data structures for CRP. This prevents a final customization on the dirty cells.

8.3.2 Alternative Route Extraction

To allow a comparison to other techniques, we also extract paths from the graph. We extract paths by employing X-BDV[9]). Given the small size of the graph, the performance is good enough and can be neglected in comparison to the construction of the graph itself. Some checks, however, cannot be performed in the restrictive setting of an the alternative graph. Especially the check for local optimality (LO) does provide reasonable results, if we do not consider the full graph. Therefore, we can only assume the different segments of the graph to be reasonable and not rely on the LO check. Our evaluation on the full graph show that this assumption can be made, though.

Further Potential. The available parallelism has some potential for further improvements, still. The trade-off between the number of levels to customize and the query time is currently only partially exploited for its parallelism. For example one could employ Δ-stepping to hasten the execution. The large matrices increase the degree of the traversed vertices, next to the creation of longer arcs, which lend themselves to a parallel implementation via Δ-stepping, for example. In addition, we could parallelize the individual tasks during the path extraction. We could also add goal direction to subsequent queries by restricting the search to cells that allow for low stretch routes to be found in the first place. Finally, we assume that there are different partitioning schemes we could implement to even better exploit the trade-off between query and penalization performance.

8.4 Experimental Evaluation

After our initial consideration with regards to the penalization scheme, we now discuss the performance of different iterations. At this point, we focus purely on parameter tuning an efficiency. Our experiments use a partition of the European road network (PTV) created by Punch [DGRW11]; the properties of the partition are summarized in Table 8.3. It was kindly provided by Daniel Delling. The partition is similar to the one used in the initial contribution of Delling et al. [DGPW11].

[9]The complete plateau method, see Section 5.2.

Table 8.3: *Overview of the partition used in our experiments to the penalty method. A graphical illustration of the partitions can be found in the Appendix (Figure A.2).*

level	vertex limit	cells	size	border vertices		border-arcs	
				avg	max	avg	max
0	2^5	664 300	27.1	5.2	21	5.3	24
1	2^8	82 278	218.9	10.0	55	9.7	44
2	2^{12}	5 046	3 569.2	26.4	96	24.9	93
3	2^{16}	319	56 458.2	67.9	208	63.4	205
4	2^{20}	20	900 509.0	136.8	384	126.2	361

The level zero is commonly referred to as the shadow-level. For our purposes, we consider two variants of the CRP algorithm. The first one considers all levels, storing the distance matrices explicitly, even for the shadow-level; we refer to this variant as CRP-4S which is essentially a five level variation of CRP. In the second variant, we only use levels one through four; this variant is named CRP-4. This version only uses the shadow-level to guide the creation of the microcode.

For customization, we tested a series of different methods from [DW13] as well as our pruned Floyd-Warhsall implementation, summarizing the results in Table 8.4. In the simple model, our microcode customization only performs best for the first level, no matter whether we study CRP-4 or CRP-4S. Surprisingly, on the first level of CRP-4S, our tuned Floyd-Warhsall implementation manages to outperform all other variants. From level two on upwards, the Bellman-Ford implementation performs best. For our update methods, we select Microcode for the first level of both CRP-4 and CRP-4S. For the topmost three levels, we select the SSE tuned implementation of Bellman-Ford. Finally, we select our tuned implementation of the Floyd-Warshall algorithm for level one of CRP-4S. This does not only speed-up our algorithm but also saves memory consumption in comparison to the microcode for the minor cost of a few lines of code.

Both settings offer comparable running times, when we perform a full update. The larger flexibility in updating only parts of a cell on the bottommost level, led us to choose the version that incorporates the shadow layer for all further processing, though.

8.4.1 Penalization

Penalizing a path in CRP requires the update of multiple cells on different levels of the hierarchy. Figure 8.8 summarizes the affected cells in our partition of the Western-European road network. Short-range queries only affect very few cells – below a tenth of a percent for the bottommost level. For long-range queries and higher levels, this percentage increases significantly. For example, a long-range query covers more than twenty percent of the topmost cells.

In combination, Figure 8.8 and Table 8.4 indicate that updating all levels might be

Table 8.4: *Update cost on different levels of the CRP hierarchy for various techniques in our implementation. The maximal value specifies the 99% quartile to correct for measuring errors. Levels two through four are the same for both CRP-4 and CRP-4S and their measurements are given in [ms]. The former levels specify times in [μs]. The Microcode for levels three and four is omitted due to its large size and decreasing benefits. The values represent a single customization.*

level	Dijkstra		Bellman-Ford		Floyd-Warshall		Microcode	
	avg	max	avg	max	avg	max	avg	max
0	6.84	18.16	3.49	9.75	2.17	3.62	**1.16**	4.37
1	22.09	83.33	11.61	42.77	**8.01**	30.33	8.28	43.98
0+1	132.04	356.56	80.89	273.17	74.83	143.97	**11.03**	43.42
2	0.25	1.12	**0.13**	0.56	0.13	0.71	0.17	1.07
3	2.07	10.72	**1.10**	5.46	2.02	12.08	-	-
4	16.89	62.43	**11.96**	39.24	33.58	103.11	-	-

infeasible. Even though the average case behaves quite well, the maximal time spent updating a single cell is very expensive.

Pruning source- and target cells. When customizing CRP, the most work is done in the lower levels. As we just argued, the distribution of workload shifts to higher levels when only considering a single path, though. In our scenario, the update costs for cells on higher levels are considerably larger than those of lower levels. Figure 8.9a illustrates the cost for the penalization of a single path. It can be seen that in the standard CRP model, even the penalization of a single path can be too expensive. The unmodified customization would not even allow for a single iteration in our setting.

The proposed pruning of the source and target cells can hasten the short-range queries, though. As illustrated in Figure 8.9b, we can decrease the cost of an iteration for Dijkstra ranks up to 2^{20} to less than ten milliseconds on average. This can be achieved by not updating those cells that are only required for long-range queries. Compare Figure 8.10 for an illustration of the full benefits. Longer-range queries, however, still suffer from the high update cost of some of the top-level cells.

For the most interesting queries this modification already allows for a sufficient number of iterations to compute a viable alternative graph, given the amount of iterations we require. We study the possible trade-off between query-performance and customization overhead by reducing the number of used levels in the next section.

Maximum Level Selection. Extracting paths from CRP is expensive, compared to an algorithm like CH. Each extraction process has to recursively find the arcs that contribute to a shortest-path segment in the lower level (c.f. Figure 8.11b). When

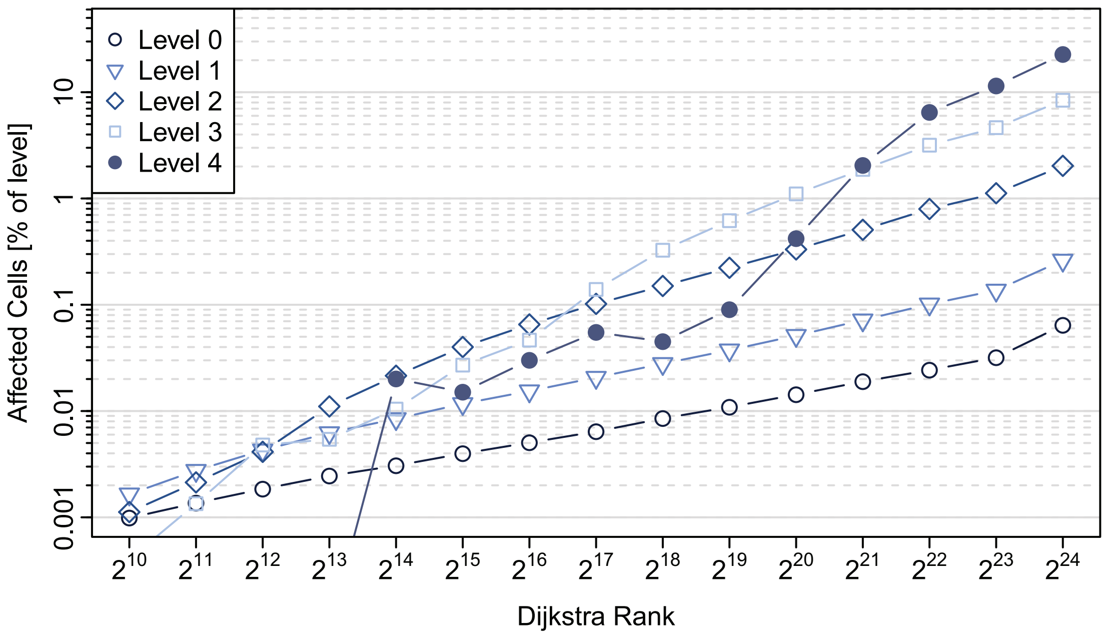


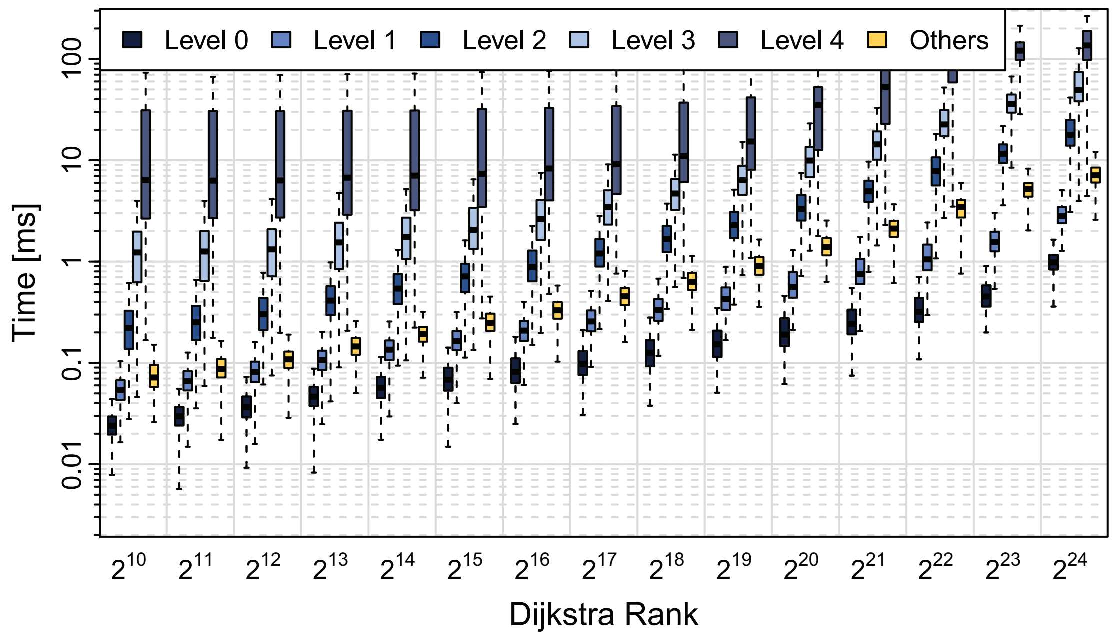
Figure 8.8: *Affected cells on the different CRP-hierarchy levels for the Western-European road network (PTV) and the previously discussed partition. This plot already incorporates the optimization to omit the cells that contain either the source or the target.*

we limit the number of levels used during the query, the query itself is undergoing a slowdown while, at the same time, the extraction process is hastened. In combination, this directly translates into a near equivalence of the settings using all levels and a setting that completely ignores the topmost one. Taking the cost for the customization process into account, it becomes apparent that the topmost level can be ignored during the penalization process. It only adds a slight benefit during the initial shortest-path computation[10].

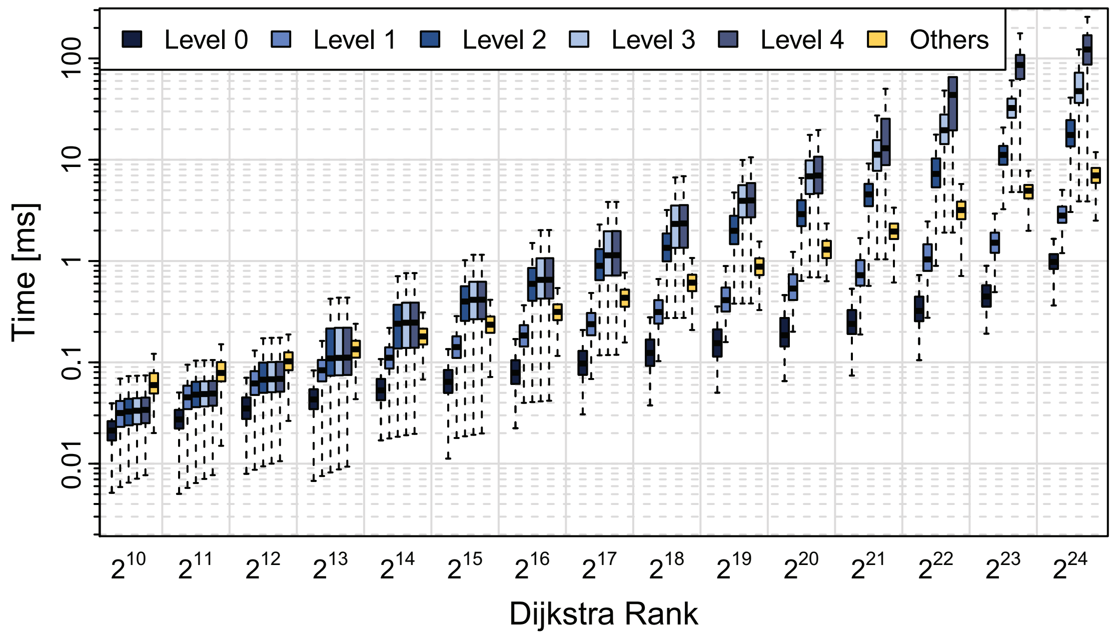
It turns out that the query itself is far more sensitive to the level selection process than the path extraction (c.f. Figure 8.11). Our measurements indicate that a restriction to the two lowermost levels is viable up until approximately Dijkstra rank 2^{15}. The addition of the third level is enough up to Dijkstra rank 2^{22}, and only very long long-range queries require four levels.

The overall process, however, is a bit more complicated. As a result of the interaction between the update costs, the query, and the path extraction, we are required to take a look at the whole picture. The only obvious fact, so far, is that the costs for the

[10]It is possible to hasten the extraction process by adding some flags that indicate whether an arc in a cell of level i is part of an arc in level $i+1$. In the dynamic nature of our algorithm, these flags would add a considerable customization overhead, though.



(a) *All cells updated.*



(b) *Source- and target-cells pruned.*

Figure 8.9: *Cost of a single iteration, depicting the accumulated time to update all cells of a given path up to the specified level as well as the additional cost associated with the respective CRP query and administrative overheads. Dijkstra rank one through ten are omitted for readability.*

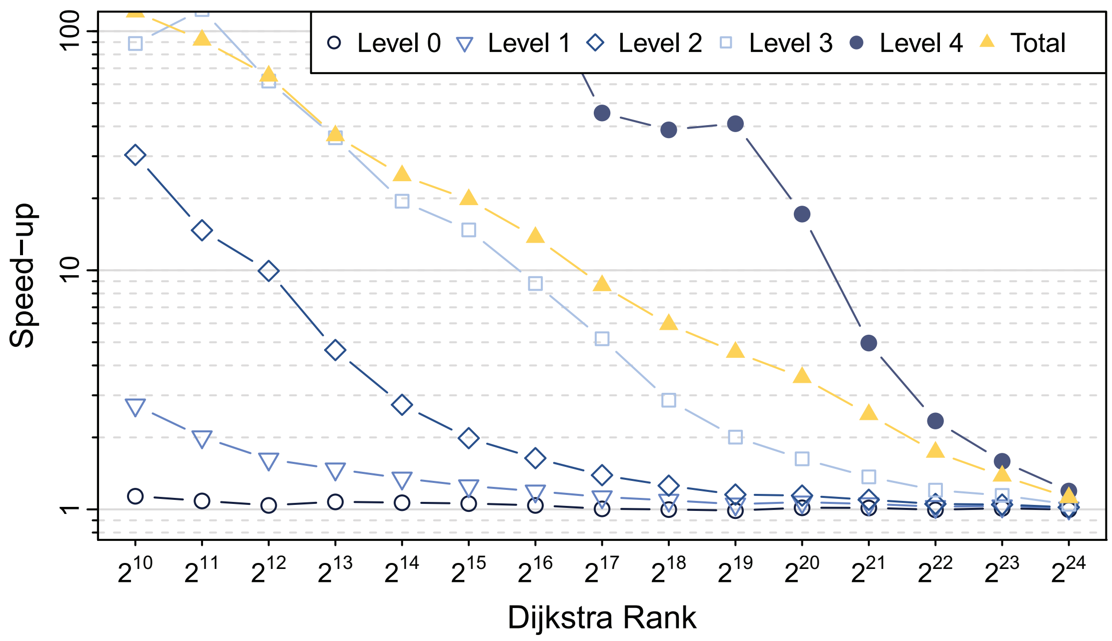


Figure 8.10: *Speed-up of penalization: pruned source- and target-cells over all cells updated.*

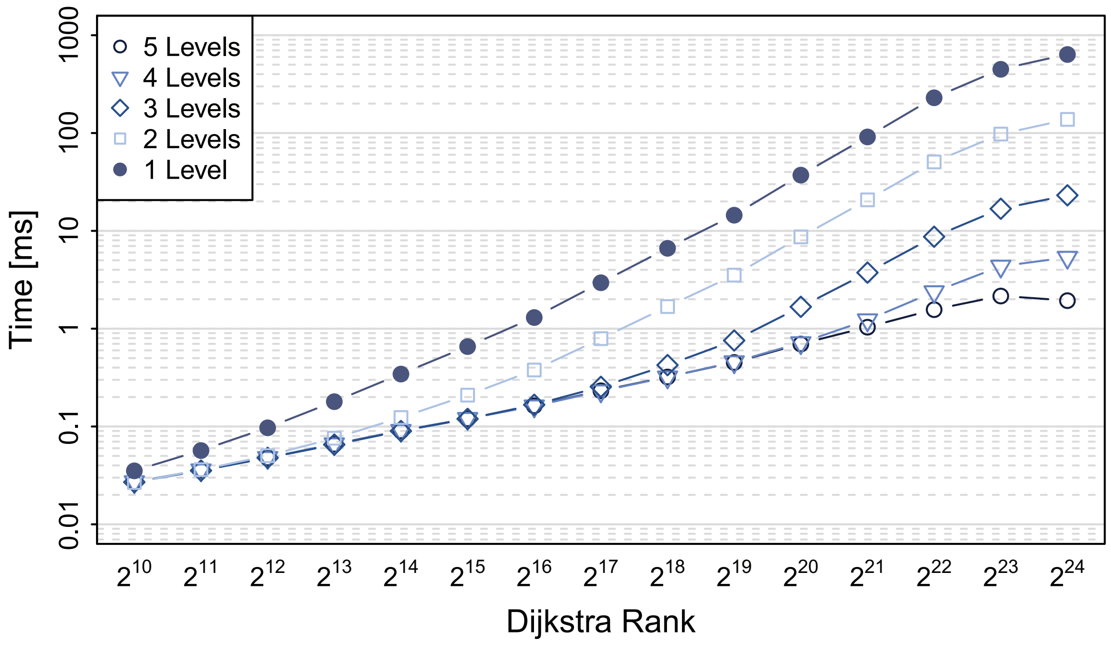
lowest two levels are negligible for longer-range queries. In total, we spent around three milliseconds on them.

In Figure 8.12 we illustrate the cost and the possible count of iterations for an implementation that considers only a certain subset of levels. Considering the full cost of an iteration, it seems that a maximum of three levels performs best. The update cost for the fourth level already outweighs the benefits during the query. Including the cost for the penalized query and all book-keeping overhead, the introduced methods already allow for alternative graph computations of queries up to Dijkstra rank 2^{20}. This is under the assumption of about ten rounds of penalization. For the largest Dijkstra rank, the sequential execution allows for a single round of penalization. A single penalization might already suffice to find an alternative route to a shortest path.

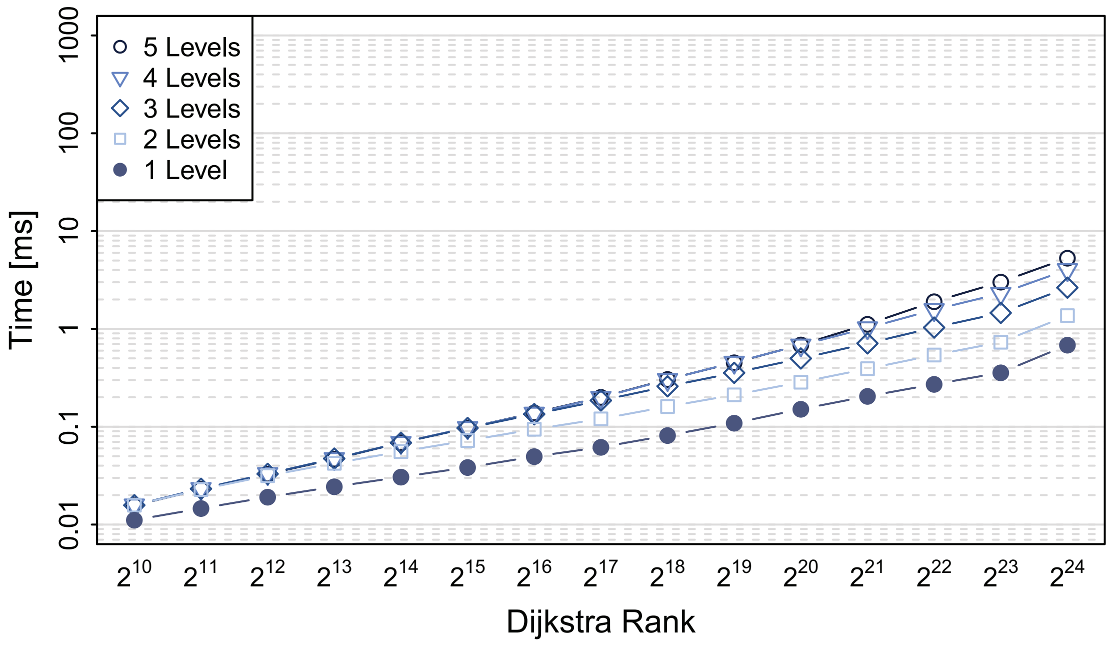
The independent parts of the computation allow for a parallel execution of the algorithm. Using multiple cores, we can increase this limited count of possible iterations to achieve a full implementation of the penalty method.

8.4.2 Parallel Execution

In combination, Table 8.4 and Figure 8.8 indicate that the query itself might be the bottleneck of each iteration. Especially when we use fewer levels, compare Figure 8.11a, the query dominates the effects of parallel-executed customization. If we use more levels,



(a) *Query*



(b) *Extraction*

Figure 8.11: *Query and extraction cost of a CRP query on G1 when limiting the algorithm to operate only on a subset of levels. The legend shows the ID of the topmost level that was used during the query. Optimizing the extraction might speed-up the overall process. It remains unclear, though, how to achieve this in our dynamic setting.*

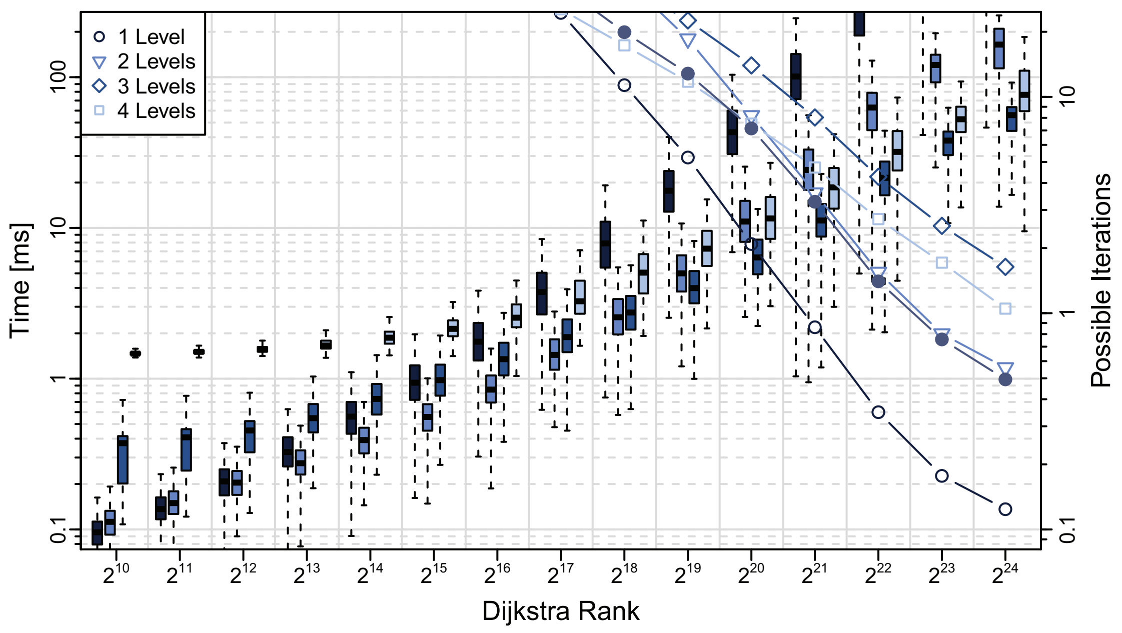


Figure 8.12: *Iteration cost (sequential) for a single penalization round and count of possible iterations in dependence of the number of used CRP levels. The customization time is given as bar plot, the number of iterations that are possible is presented as lines.*

however, we can achieve increased performance, which we summarize in Figure 8.13.

Even when using parallel execution, the cost for updating the different levels remains high. Still, parallel execution enables us to perform up to three iterations, even for the most rare types of ultra-long-range queries. The rarity in which we expect requests for an alternative graph throughout all of Europe is enough justification to not interpret the bound of a hundred milliseconds as strict. If necessary, our algorithm can still compute an alternative graph of lesser quality within the time limit.

In general, the faster memory access of our machine M2, in addition to the improved architecture, results in a faster sequential execution time in comparison to the execution on M1. On average, we can achieve two penalization rounds using M2, even for Dijkstra rank 2^{24} . M1 allows only for an average of one and a half iterations using three levels and sequential execution[11].

In parallel execution, the running times of a query dominate the overall execution time for large Dijkstra ranks, as long as we use less than four levels. The benefit of parallel execution for a setting in which we use three or less levels remains negligible.

What becomes apparent, nevertheless, is that the standard partitioning seems to be not the best choice when it comes to penalization. The update cost for the fourth level

[11]The respective visualization can be found in the Appendix in Figure D.2b.

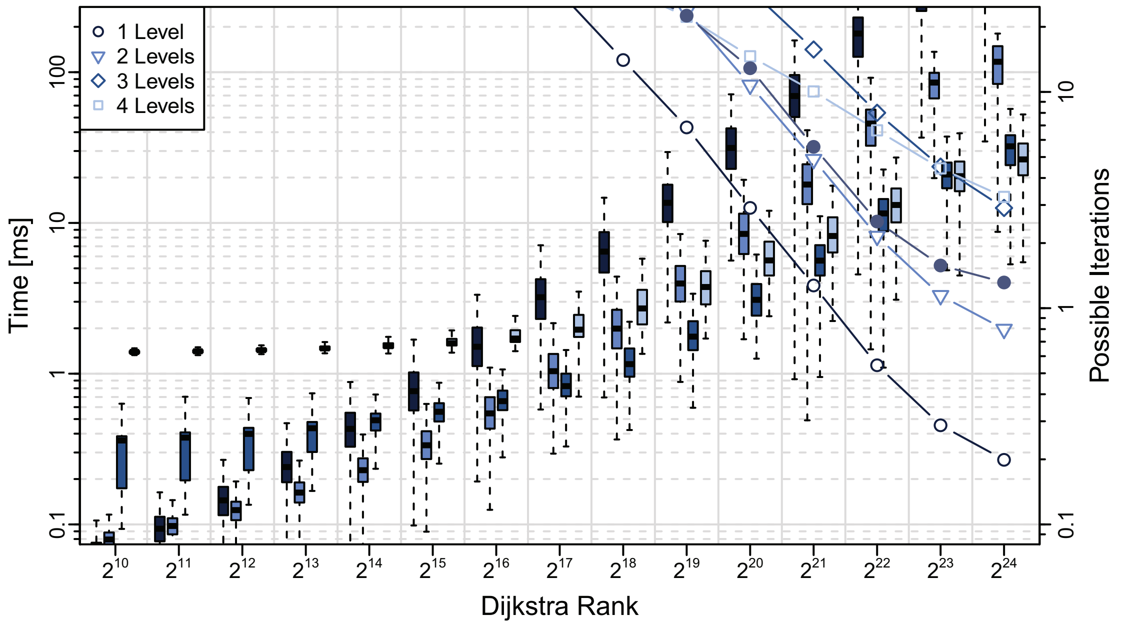


Figure 8.13: *Parallel penalization on G1, using twenty-four threads on M2. The customization time is given as bar plot, the resulting number of possible iterations is presented as lines.*

is too high, as the amount of cells that can be updated in parallel is too low. For three levels and less, the query time is dominant.

Adjusted Cell Sizes. Figure 8.14 illustrates that splitting large cells provides a better trade-off than previously observed. Even in sequential execution, using four levels is already faster than the the setting using only three levels for Dijkstra ranks between powers of twenty-two and twenty-four. Therefore, we chose to use these modifications to improve performance for larger Dijkstra ranks.

8.4.3 Final Configuration

As previously mentioned, after extensive measurements on different configurations, we selected $\psi = 1.1$ and $\psi_r = 0.01$ for our implementation of the penalty method. For the penalization, we chose an exponential penalization of path arcs and an asymmetric additive penalty applied once to all arcs adjacent to a (penalized) shortest path. The chosen configuration provides the results illustrated in Figure 8.15. For our timing measurements, we present values measured on twenty-four threads, an amount of threads only required for peak performance. Measurements using only four threads are barely slower. Most times, using only a few threads suffices. Further details on possible configurations can be found in Appendix D.

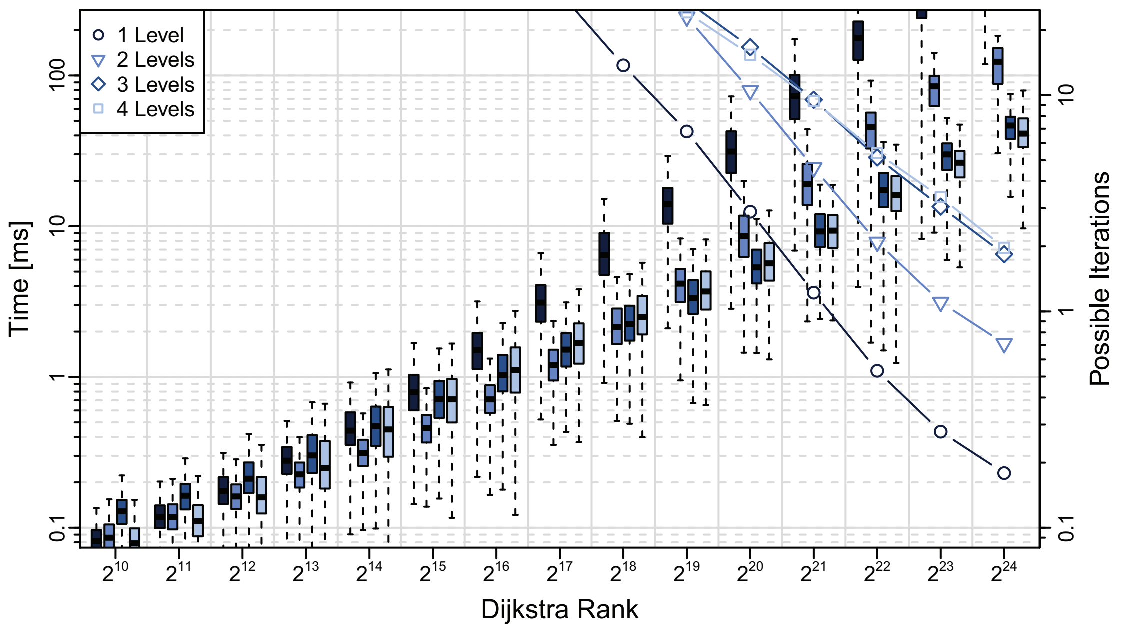


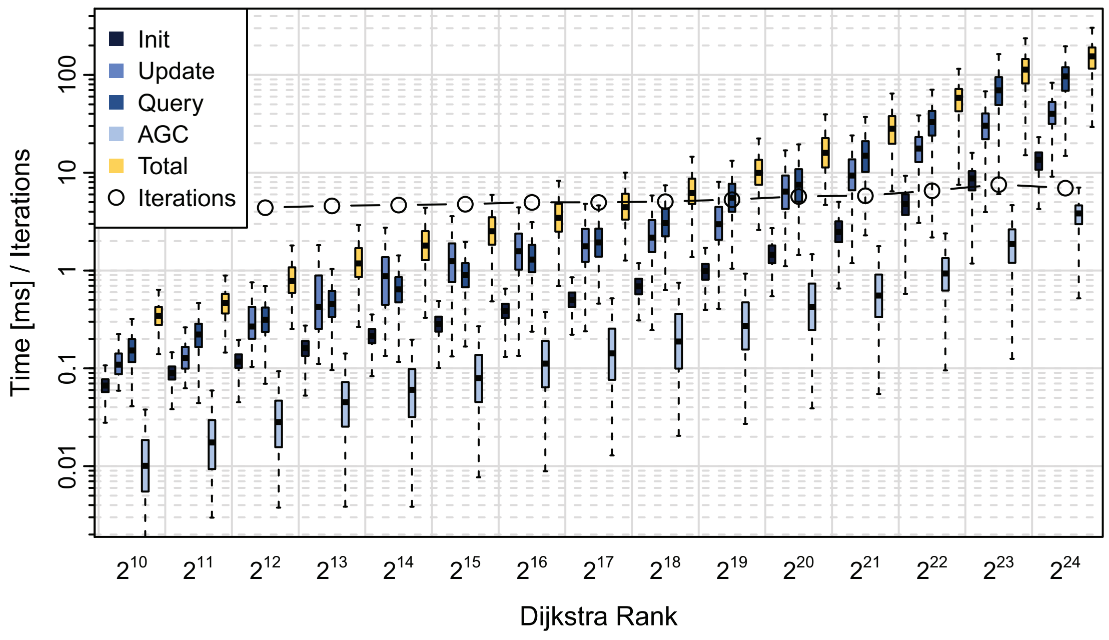
Figure 8.14: *Cost of a sequential penalization of G1 on M2 using a partition in which large cells have been split into four separate cells. Compare Figure 8.12 for a setting without this subdivision.*

On random queries, we reproduce the values specified by Bader et al., reaching a value of $\mathcal{D}_{tot} = 3.17$ ($\mathcal{D}_{avg} = 1.04$, # arcs = 7.2) on average compared to their value of $\mathcal{D}_{tot} = 3.34$, which requires an average query time of a hundred milliseconds. For short-ranged queries (maximum of 300 km between s and t), we achieve $\mathcal{D}_{tot} = 2.23$ in around ten milliseconds. On these short-ranged queries, our Dijkstra implementation requires an approximate eight milliseconds for a single run.

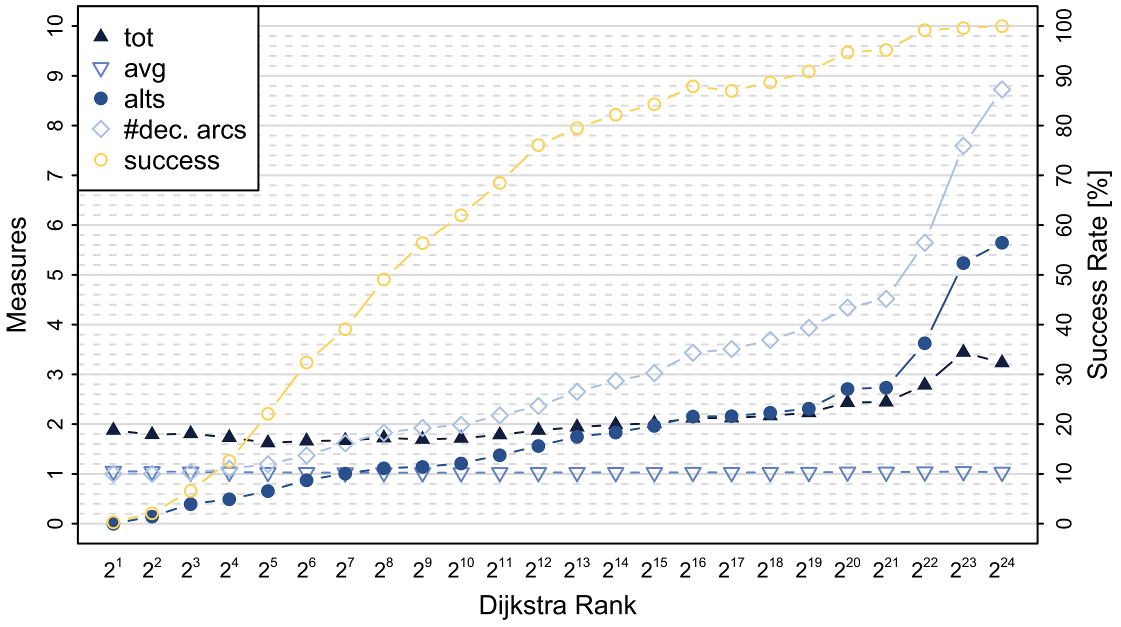
Given our additional restrictions on the paths and our online selection strategy, we are willing to accept this minor decrease in quality, compared to Bader et al.. All of these values can be tuned to provide even better quality (compare Figure 8.3, Set-up 6); we deem this unnecessary, however, as the actual values depend strongly on the underlying graph and especially the used metric.

We achieve near interactive query performance for the full range of Dijkstra ranks (Figure 8.15a).

Quality. In Figure 8.16, we illustrate the development of the different quality parameters. The stretch of different paths increases slowly from short-ranged queries to long-ranged queries. The amount of sharing (α) remains mostly constant at around fifty percent. The local optimality degrades from around ninety percent to around thirty percent. The distance between the points offering the largest values for bounded



(a) *Timing (M2)*



(b) *Quality*

Figure 8.15: *Final evaluation of our implementation of the penalty method, using $\psi = 1.1$ and $\psi_r = 0.01$. The quality specified depicts average values with respect to the successfully executed calculations ($\mathcal{D}_{tot} > 1$). Figure 8.15a provides accumulated timings for the initial query (init), updates (penalization and customization), repeated queries (query) and the construction of the actual alternative graph (AGC). In addition, we depict the average number of iterations performed as well as the total time taken for our algorithm.*

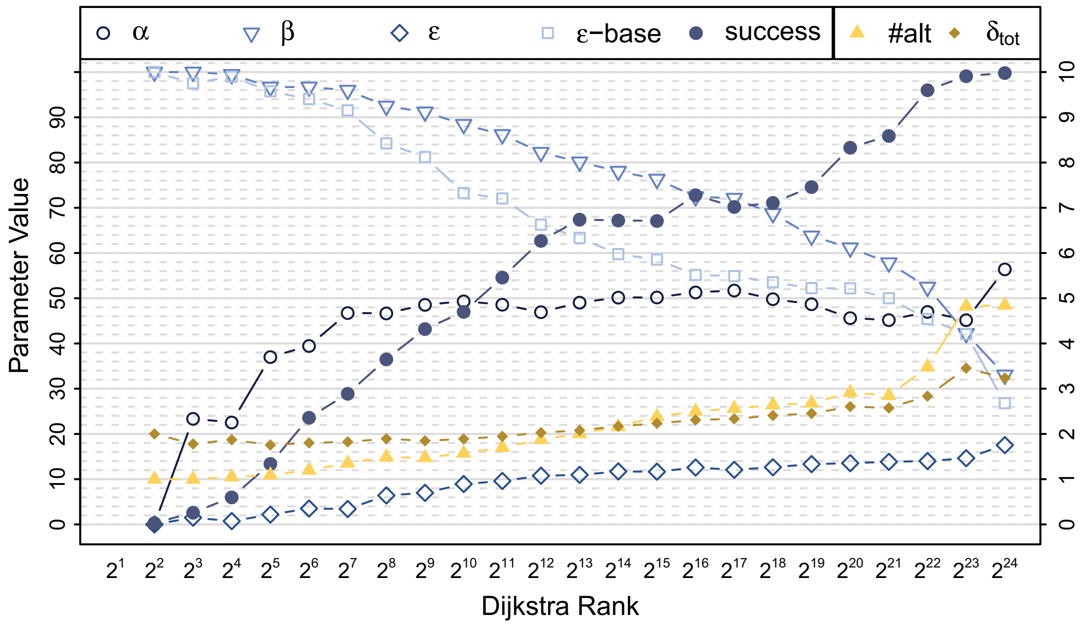


Figure 8.16: *Development of quality for Customizeable Route Planning with Penalization (CRP-π) over different path lengths*

stretch (ϵ) behave in the same way. $\mathcal{D}_{tot}$ increases slowly, peaking at a Dijkstra rank of 2^{23}. The number of first alternatives follows this development, not dropping of at rank 2^{24}, though.

8.5 Conclusion

Even though it seemed unlikely at first, the penalty method can be implemented efficiently. In our studies, we thoroughly explored the different trade-offs associated with an implementation on the basis of CRP and were able to configure the algorithm to our specific goals. We refer to our highly tuned implementation as CRP-π. Exploiting the initial shortest-path search, we can dynamically adjust our technique to handle every possible scenario in the most reasonable way possible. Our final result offers a technique that is comparable to the results of Bader et al. [BDGS11] in its quality and can handle close to any query type within or at least close to a hundred milliseconds.

The mentioned optimization potential for further parallelism could result in even better performance. Also, tuning the partition further to fit the desired purpose even better might prove beneficial. The formerly mentioned radix heap could prove beneficial as a replacement for the utilized binary heap. Finally, also the query itself offers some potential for parallelization. While we found the parallel execution of both forward and backwards search to be subject to too much synchronization overhead to justify the implementation, introducing Δ-stepping could offer a large potential for further exploitation of parallelism. Ours is the only implementation of the penalty method suited for interactive applications, even on continental-sized road networks.

Chapter 9

9 A new Approach to Via-Alternatives

This chapter presents a different approach to the calculation of via-alternatives and alternative graphs in general. We present the first speed-up technique based algorithm that directly computes a full alternative graph, rather than evaluating a series of path candidates. Our algorithm requires a single bidirectional query to succeed and offers high quality alternative routes in a competitive amount of time. This chapters describes the single-author publications [Kob13].

The most common approach for the calculation of alternative routes, so far, is the via-vertex appraoch. The algorithms presented by Abraham et al. [ADGW13] as well as an extension due to Luxen and Schieferdecker [LS12] both operate in a similar fashion. After selecting a via-vertex candidate in a heuristic fashion, the vertex is tested for admissibility and selected on success. The ordering heuristic approximates the quality of an alternative route based on shortcuts. The test itself requires three shortest-path queries. Therefore, both methods present the first viable alternative path as a result instead of evaluating all candidates.

The discovery of candidates is one of the major parts of both algorithms. Abraham et al. require a relaxed pruning mechanism to achieve reasonable success rates. They relax the pruning mechanism by allowing the query to descend within the hierarchy. This descend is controlled via the levels of up to k parent vertices[1]. The relaxation not only increases the success rate. The highly increased number of candidates also increases the workload of the algorithm during its testing phase.

Luxen and Schieferdecker follow a different approach and build an alternative route cover for all vertices within pairs of distinct regions. Their algorithm employs the classical algorithm of Abraham et al. [ADGW13] as a base case to detect possible alternative routes. Every discovered viable candidate is remembered in relation to the regions of the source and the target. For every query, their algorithm first explores

[1]Google recently patented a method [Gei14] that might improve this mechanism.

these stored candidates. If a previously found vertex offers a viable candidate, the vertex is reported as via-vertex. Else, the method uses the algorithm of Abraham et al. to find new candidates. In combination with the Arc-Flags (ArcFlags) algorithm, [LS12] offer the currently fastest algorithm for the discovery of alternative routes.

One point of critique, in our eyes, is the sequential evaluation of different candidates that reports the first viable via-vertex. We think that the heuristic selection of candidates might negatively impact the route quality. In this chapter, we develop an algorithm that can test all its candidates at once. Even though we require a larger initial overhead, the simultaneous tests results in competitive running times to the method of Abraham et al..

Our method borrows ideas from PHAST [DGNW13] and [LS11] to calculate an alternative graph on top of a single bi-directional shortest path query. The method does not require additional relaxation. The basic principles are comparable to the method presented in [Cam05]. Instead of using Dijkstra's algorithm, however, we use CH.

9.1 Alternative Graphs via Plateaux

Previous techniques for alternative routes all follow an iterative approach; they compute a single path at a time. Surprisingly, even techniques that focus on the discovery of full graphs [BDGS11, PZ13] calculate a series of paths. The only exception is Choice Routing [Cam05] and the reference algorithm X-BDV of Abraham et al. (see Section 5.2).

Both techniques require Dijkstra's, though, which is too slow for long-range queries in an interactive setting. We improve upon Choice Routing and X-BDV by applying their concept, i.e. the discovery of plateaux, to Contraction Hierarchies. By testing all candidates in a single bi-directional query, we are unique among all (speed-up technique based) algorithms designed to compute alternative routes.

Our approach operates in three different steps.

Shortest-path Trees The first step computes two (partial) shortest-path trees. Since we are using a CH, these trees consist mostly of shortcuts.

Hierarchy Decomposition As in [LS11], we collapse the hierarchy of the CH. This phase discovers the plateaux and prepares our alternative graph.

Post Processing In a final step, we process the graph to contain less unnecessary data.

The first step is a bi-directional query that computes two shortest-path trees. As in X-BDV, the intersection of these trees defines so-called plateaux, which can be used to perform the T-test for local optimality (see Section 5.2.1). Rather than evaluating single plateaux for alternative routes, we construct a full graph that maintains all the information required for alternative route extraction.

Speed-up techniques in general are optimized for point-to-point queries. For CH, however, techniques exist that compute full and partial shortest-path trees [DGW11, DGNW13]. Both techniques use an initial upward search in combination with a sweep. PHAST processes all vertices, Restricted PHAST (RPHAST) a precomputed subset of the vertices. For our approach, we cannot compute these vertices in advance but have to identify them. We describe our approach to this problem in Section 9.1.1.

By using a CH, we face the same problem as the classical via-vertex approach. The trees found in the first step contain a lot of shortcuts. Within these shortcuts, some of the required plateaux can be hidden. Our second step in the algorithm processes these trees, extracting all shortcuts to discover the plateaux.

Collapsing the hierarchy can be a costly operation. In fact, this step contains the largest part of the workload. As the most costly part of the algorithm, it also provides the name for our algorithm: Hierarchy Decomposition for Alternative Routes (HiDAR). The process is difficult, not only because the amount of arcs represented by the shortcuts is large. In addition, even at first glance fully unrelated shortcuts can share a common segment.

This step, similar to the usually employed relaxation, is only a heuristic, though. We cannot guarantee the discovery of all viable via-vertexs. The heuristic is very good, though, and misses fewer vertices than the usual relaxation. In comparison to X-BDV, we cannot offer the same success rates, though. We narrow the gap to the reference algorithm, though.

Finally, the last part of our algorithm performs post-processing on the computed alternative graph.. This step is independent of the alternative route extraction algorithm and reduces the size of the alternative graph. We both compress and filter the graph, based on some information computed in step two, without actually touching all vertices a second time. The graph that we construct provides all information required to perform a quick evaluation of alternative route viability with respect to the criteria set forth by Abraham et al. (see Definition 5.4).

9.1.1 Building Shortest Path Trees

The first step of Hierarchy Decomposition for Alternative Routes (HiDAR) is the computation of two shortest-path trees. For our algorithm to perform well, we need to restrict the trees to a reasonable subset between the two extremes: a full shortest-path tree and a point-to-point query.

In the decision process, we performed some initial experiments using the algorithm of Abraham et al. [ADGW13]. The experiments revealed that all viable via-vertexs were actually contained within one of shortcuts of either the source or the target. While we cannot guarantee this in general, the measurements justify our approach to calculate full shortest-path trees for the union of search spaces of the source and the target.

Due the the large overall size of all possible search spaces, we cannot compute the search space of all vertices in advance, as done for RPHAST [DGW11]. In our algorithm, we combine the discovery of the vertices with the initial upward search.

1. Upward Search and Sweep Space Generation. Both the generation of the sweep space and the initial distance computations traverse the graph in an upwards fashion. We can exploit this identical mode of operation and combine both into a single parse. Whereas we require Dijkstra's algorithm to compute distances in a general graph, we can utilize a different order of processing in a CH. Since the search is directed upwards, we can order the vertices with respect to their level in the CH [2].

Theorem 9.1 (Correctness of the Upward Search)**.** *Processing the vertices in order of their CH-level does not influence the distance calculation.*

Proof. Similar to the correctness of PHAST, we can employ an inductive argument. Whenever a vertex v is processed on level ℓ, its distance is correct and all arc relaxations use a correct distance value. The source offers the base case with a correct distance of zero. The distance of a vertex v on level ℓ depends only on vertices at level $\ell - 1$ and smaller. The inductive argument claims them to be assigned a correct distance value. As a direct result, the distance value of v is also correct. □

Naturally, this correctness argument only holds with respect to the search space of the CH. An alternative argument would be that a breadth-first search (BFS) can compute correct distance values in a weighted DAG, as well.

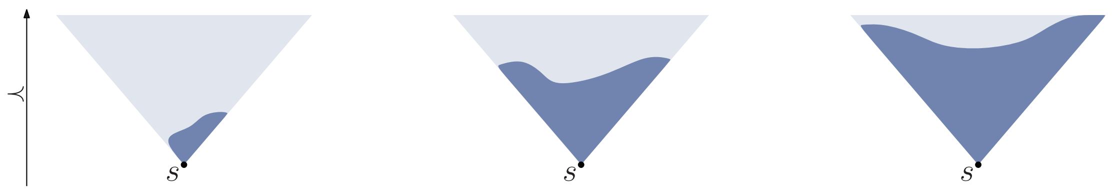
Using the CH level as ordering criterion, we can compute both all reachable vertices and the initial distance values at the same time, saving an additional parse over the vertices. The exact process is described in Algorithm 9.1. We describe it the way it works starting at the source. The algorithm works the same way from the target, though.

During the upward search on the forward graph, we add additional vertices to the queue. All vertices that are reachable in the backward graph are added to the queue with an infinite weight. When we process a vertex of infinite weight, we only consider the backward graph for relaxation.

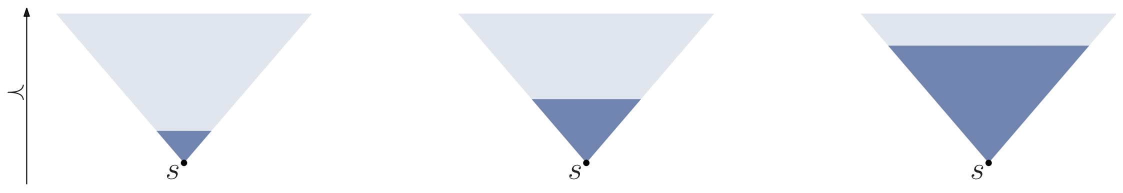
The union of all touched vertices forms the set of vertices that we sweep in the second phase.

We can also employ some initial pruning at this point, throwing away all vertices for which the upward part already indicates a larger distance than $1 + \epsilon$ times the best known value for $\mathcal{D}(s, t)$ so far. In doing so, we employ the usual stopping criterion of a CH. The correctness of this approach follows directly from the correctness of a CH query. Similar to the usual CH query, we advance the two searches simultaneously and keep track of a tentative distance value for the full path.

[2]The level in a CH is defined as the number of hops from a vertex to the topmost vertex.



(a) *Classic*



(b) *Bucket*

Figure 9.1: *Changed operation order: the top row shows a schematic of the classic order that uses a priority queue to calculate distances within the search space of the source s. The bottom row shows our operation order that processes the search space of s in a per-level fashion. Due to the full exploration of the search space, we gain no benefit from the priority queue based ordering.*

The method proposed seems promising as we do not require the ordering capabilities of a priority queue for the vertices, perform only a single parse over the search space and still do not lose much in terms of pruning potential over the standard CH query.

Formally speaking, the result of this initial phase, when starting at the source, consists of all vertices that are reachable in the union of $G^{\uparrow}$ and $G^{\downarrow}$ from either source or target. This is different from the search space of Dijkstra's algorithm as we search backwards in $G^{\downarrow}$; both searches only go up the hierarchy. In addition to the sweep space, we perform the full initialization as done in PHAST or RPHAST.

2. Sweep. In the second phase, we can operate purely using the information gained from the first phase. Exactly as in RPHAST, we can sweep the set of vertices that we have computed in the previous step (RPHAST calculates the sweep sets in advance). This phase extends and corrects the tentative shortest-path trees found in the first step of this phase. This second part, the sweep, concludes the first phase of our HiDAR algorithm.

Theorem 9.2 (Tree correctness)**.** *The query of HiDAR calculates two shortest-path trees.*

Proof. The correctness is a direct implication of the correctness of RPHAST, the regional version of PHAST, in combination with the sweep space. As seen in Theorem 9.1, the modified upward search offers a correct initialization.

Algorithm 9.1 ComputeShortestPathTreesAndSweepSpace

Input: A graph processed into a CH, a source s and a target t

Output: The shortest-path distances between s and all vertices reachable from the source in $G^{\uparrow}$ and ∞ for all vertices reachable in $G^{\downarrow}$

```
1:  $\mathcal{Q}[\cdot] := \emptyset$
2:  $\mathcal{Q}[\ell[s]].push(s)$
3:  $distance[] = \infty$
4:  $distance[s] = 0$
5:  $sweep = \emptyset$
6:  for all $level \geq \ell[src]$ do
7:      for all $v \in \mathcal{Q}[level]$ do
8:          $sweep.push(v)$
9:          if $distance[v] \neq \infty$ then
10:             for all $(v, w) \in G^{\uparrow}$ do
11:                 $distance[w] = min(distance[w], distance[v] + \mu((v, w))$
12:                 $\mathcal{Q}[\ell[w]].push(w)$
13:             end for
14:         end if
15:         for all $(v, w) \in G^{\downarrow}$ do
16:             $\mathcal{Q}[\ell[w]].push(w)$
17:         end for
18:     end for
19: end for
20: return $(distance, sweep)$
```

For the correctness of the sweep, we can directly use the proof for RPHAST which requires the full upwards search space of a vertex. Since we are extending the search in $G^{\downarrow}$ from every vertex, this is obviously the case. □

9.1.2 Discovering Plateaux

The second phase of HiDAR is concerned with the discovery of the different plateaux. The separation of a CH into two different graphs and the presence of shortcuts prevents a similar process as done for X-BDV and Choice Routing. The plateaux are hidden within shortcuts. These long paths, condensed into a single arc, might share a common segment, even if their respective origins and destinations are distinct.

The problem we are facing is similar to the decision of whether a common substring can be constructed in a very specific context free grammar [Cho59], starting at two different symbols. To the best of our knowledge, a full exploration of the respective words that can be constructed from the grammar is the only viable option. We take

Algorithm 9.2 unpackShortcuts

Input: A graph processed into a CH, a list of arcs $\mathcal{A}$ to unpack
Output: The unpacked shortcuts

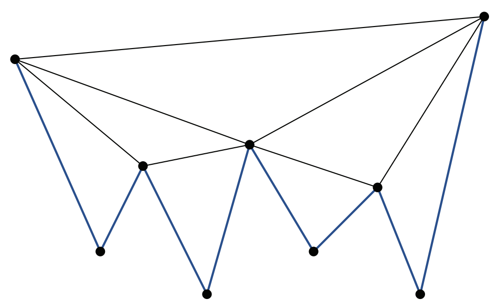
```
$\mathcal{Q}[\cdot] := \emptyset$
$\mathcal{H} := \emptyset$
for all $a \in \mathcal{A}$ do
    if isShortcut( $a$ ) then
        $\mathcal{Q}[\ell[middleVertex(a)]].push(a)$
    else
        $\mathcal{H} = \mathcal{H} \cup a$    ▷ remember non-shortcuts for the alternative graph
    end if
end for
for all level do    ▷ form top to bottom
    for all $a \in \mathcal{Q}[level]$ do
        if isShortcut($a$) then
            if not $\mathcal{Q}[\ell[middleVertex(a)]].contains(a)$ then
                $\mathcal{Q}[\ell[middleVertex(a)]].push(a)$
            end if
        else
            $\mathcal{H} = \mathcal{H} \cup a$    ▷ remember non-shortcuts for the alternative graph
        end if
    end for
end for
return $\mathcal{H}$
```

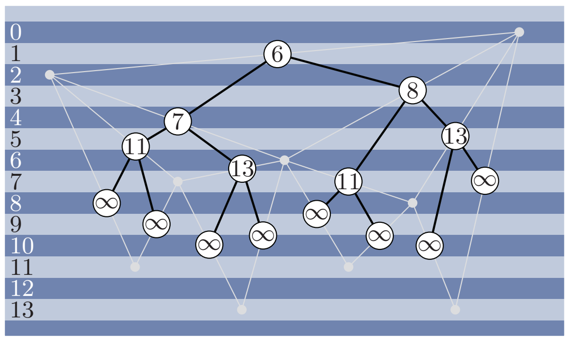
care, however, to reduce the work done as much as possible by carefully extracting every unique shortcut exactly once.

To achieve this, we follow an idea similar to the one already discussed in [LS11]. Their algorithm scans all arcs within the CH in reverse contraction order; they call this process hierarchy decomposition. Since we cover only a very small part of the graph, we cannot scan the full set of arcs, though. Instead, we extract arcs in the same fashion that we already employed for initialization; the only difference is that we operate in reverse contraction order.

We stick to the notion to unpack all shortcuts simultaneously, though, instead of fully extracting one shortcut after another. Our algorithm is using the same bucket-structure as in Algorithm 9.1. To guide the process, we order the different tasks by their respective middle vertices. Algorithm 9.2 provides the respective pseudo-code. For every shortcut, we issue an unpacking request to queue of the level of the shortcuts' middle vertex.

We can interpret this approach as an instance of Dijkstra's algorithm in a special


(a) *CH graph*



(b) *Priority based extraction algorithm*

Figure 9.2: *Unpacking process of a single shortcut (9.2a, represented shortest path in blue at the bottom). During the execution of our algorithm, we deal with a superposition of a multitude of such trees. In the case of a full extraction, we are operating on a DAG instead of a tree.*

graph, compare Figure 9.2 for reference. The source of the algorithm is an artificial vertex that is connected to all arcs we want to unpack. The extraction tasks themselves represent the vertices in this paradigm. Every shortcut is connected to its comprising arcs with the difference in levels between their middle vertices for a weight. If an arc is not a shortcut, it is connected with infinite weight. The extraction process now performs Dijkstra's algorithm on this graph until all remaining vertices are of infinitive distance. Due to the processing order, every shortcut is extracted exactly once, as the unpacking is the equivalent to the relaxation in Dijkstra's algorithm.

So far, we only talked about how to unpack the information in the shortest-path trees and have not even mentioned the plateaux. In discovering the plateaux, we have to deal with the fact that a plateau might span multiple shortcuts or arcs. As a result, fully marking them and calculating their length is not possible yet.

To do so, we require some additional information at the shortcuts. We enable our unpacking to detect plateaux by storing information on the next and prior vertices with every middle vertex. In the upcoming section, we describe how to use this information to not only discover plateaux in an efficient way but also to create a compact representation of the alternative graph at the same time.

9.1.3 Generating a Compact Representation and Actually Finding the Plateaux

The method for discovering plateaux and directly compacting the graph at the same time is the central component of HiDAR. We develop it in multiple steps, refining it along the way.

We start by first discussing some observations and introducing some additional concepts. A central concept of HiDAR is the so-called *important vertex*. In our interpretation, an important vertex is a vertex that offers important information within the alternative graph. This information can take one of the following forms: it is either a possible decision (offering more than one vertex as possible successor) or is required to mark the beginning/end of a plateau. Finally, a vertex might be required for correct overlap calculations, i.e. a vertex with more than one predecessor. The formal definition of an important vertex is given in Definition 9.1.

Definition 9.1 (Important Vertex)**.** *A vertex in an alternative graph is important if one of the following conditions holds:*

1. *a plateau begins at the vertex*
2. *a plateau ends at the vertex*
3. *a decision can be made at the vertex* *(more than one arc begins here)*
4. *two paths join* *(more than one arc ends at the vertex)*

The search for important vertices forms the most basic step on our way to the full plateau algorithm.

The Conceptual Plateau Algorithm. First consider a setting without any shortcuts. In this setting, we can easily describe how to detect important vertices: at each vertex we only have to observe the arcs associated with it. The most obvious aspect of this is the number of possible predecessors and successors which can be calculated via a simple count. What is less obvious is the start/end of a plateau. Given two shortest-path trees, however, we only need to compare the two. A plateau starts/ends at v if only a single arc at v is part of both trees.

The algorithm is described in pseudo-code in Algorithm 9.3.

As can be seen from this initial Algorithm, the decision whether to keep an arc or mark a vertex as important can be done locally at every vertex. The only requirement is to have information on possible next or previous vertices along the tree.

Plateau Discovery Algorithm. As we just stated, we only require the direct predecessors and successors of every vertex to construct the alternative graph. In Algorithm 9.3 we assumed this information to be already given in form of the unpacked shortcuts. For the full algorithm, we have to store the predecessor/successor of the middle vertex with every shortcut. The information can easily be computed in advance, though. Given this data, we process the shortcuts as described in Algorithm 9.2.

Following the argument of running Dijkstra's algorithm on a modified graph (c.f. Figure 9.2), we are guaranteed to know about all shortcuts that contain the same middle vertex v at the moment we process it

When we allow shortcuts, some of them will represent identical segments. For every one of these segments, we choose a representative to be used in the algorithm To allow

Algorithm 9.3 Plateau Algorithm Concept

Input: A list S of shortcuts, a source (s), and a target (t)
Output: An alternative graph

```
1: $V := \{v : \exists_u (u,v) \in S \vee (v,u) \in S\}$    ▷ alternative graph vertices
2: $A = \emptyset$    ▷ alternative graph arcs
3: $I = \emptyset$    ▷ important vertices
4: for all $v \in V$ do
5:     $plateau = \texttt{false}$
6:     if $\exists_{u \in V}(u,v) \in S \wedge (v,u) \in S$ then    ▷ $(u,v) \in G^{\uparrow} \cap G^{\downarrow}$
7:         if $|\{(w,v),(v,w) \in S \setminus (u,v),(v,u)\}| \geq 1$ then    ▷ start or end of a plateau
8:             $I = I \cup v$
9:         end if
10:        $plateau = \texttt{true}$
11:    else
12:        if $|u : (v,u) \in S| > 1 \vee |u : (u,v) \in S| > 1$ then
13:            $I = I \cup v$
14:        end if
15:    end if
16:    for all $u, (v,u) \in S$ do
17:        $a := min_{ID}((v,u) \in S)$
18:        $A = A \cup (a, plateau)$
19:    end for
20: end for
21: return $V, A, I$
```

for a consistent choice, we keep track of the arc-ID of the original shortcut. This ID is used as a priority. In our unpacking process, we only process the arc with the minimal ID when we encounter two arcs that have the same source and target. As can be seen in the set notation used in Algorithm 9.3, we only require a single shortcut rather than all parallel ones.

To allow for correct discovery of plateaux, however, we need to carry over forward and backward flags to the chosen shortcut when discarding a parallel one. The only difference with respect to Algorithm 9.3 is that we need to operate on the predecessors/successors stored with the middle vertices when we process the arcs as most of the arcs describe a shortcut at the moment they are processed.

This simple transition allows us to compute important vertices as well as all vertices and arcs for the alternative graph during the unpacking process. One could ask, however, why we require integrate this process with the unpacking as we could simply perform the unpacking process and afterwards run Algorithm 9.3. The reasoning for this can be found in the next section.

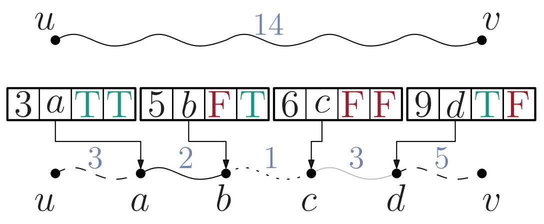


Figure 9.3: *Interaction between a shortcut of length* 14 *connecting the two vertices* u *and* v *and a list of segment indicators. The segment indicators define 5 different arcs in relation to the original shortcut. Assume the shortcut* (u, v) *to be an arc from* $G^{\uparrow}$*, i.e. directed forwards. In order with respect to their distance along the path, the segment indicators indicate the start of a plateau, a backwards directed segment, an inactive segment and finally another forward segment. The respective length can be calculated directly from the distance values stored in the segment indicators as well as the length of the shortcut.*

9.1.4 Direct Alternative Graph Compaction

The final goal of our algorithm is a compact representation of the alternative graph. The number of vertices we process during Algorithm 9.2 is expected to be very large. In comparison, we expect far fewer choices to be encoded in the graph. For the optimal performance, we try and minimize the work performed on all of the vertices. While we do not know a way to circumvent the full extraction of all paths from the shortcuts, we are able to avoid operating on the entire set of vertices more than once. We do so by directly computing a compact representation during the unpacking process.

The parts of a shortcut that we use in the alternative graph can be encoded in a simple way, using an additional concept: Given an unpacked shortcut as a sequence of vertices $\langle v_1, \ldots, v_k \rangle$, we can represent its contribution to the alternative graph as a list of vertices and associated flags, ordered by their occurrence in the sequence. The exact sequence of vertices in between can be omitted. We call this concept *segment indicator*. The vertices between two segment indicators form a segment. Formally, we define both segments and segment indicators in Definition 9.2.

Definition 9.2 (Segments, Segment Indicators)**.** *A segment indicator to a sequence of vertices* $S = \langle v_1, \ldots, v_k \rangle$ *is a vertex* $v \in S$ *with associated flags. The flags mark (in)activity and plateaux. In addition to these flags, every segment indicator is associated with a distance value that marks its distance from the first vertex of the sequence.*

Given two ordered segment indicators $v_i, v_j, i < j$*, the (sub)sequence* $\langle v_i, \ldots, v_j \rangle$ *defines a segment. Depending on the flags associated with* v_i*, the segment is either active or inactive, plateau or not plateau (only available in forward or backward direction). Refer to Figure 9.3 for an example of segment indicators to an arc.*

As long as active segments are distinct over all shortcuts, the respective segments can be used in the evaluation of alternative route admissibility. This allows for the direct creation of a compact representation of the alternative graph by using important vertices and adding arcs for all active segments.

Knowing the extended version of Algorithm 9.3, the correct segment indicators can be generated, as very segment indicator is associated with an important vertex. When we discover an important vertex, we simply generate a single active segment indicator for the shortcut with the lowest ID (which we chose to represent an arc in the alternative graph). For all other shortcuts at the important vertex we generate an inactive segment indicator. By focusing on the shortcuts with the lowest ID, we guarantee correctness of the algorithm. The details are discussed in Theorem 9.3. Before discussing its correctness, we first describe the process.

Segment Indicator Generation. Every important vertex is either directly connected to a plateau (start or end), offers a choice for the next traversed arc (compare decision arcs), or combines two paths into a single path. Segment indicators correspond to the respective arcs that describe the corresponding graph underneath. As a result, we obviously only require segment indicators at important vertices. At every such vertex, we simply compare the inbound and outbound (shortcut) arcs with regard to their last/next vertex along the way.

If we find more than a single vertex as a predecessor/successor or find a shortcut from both $G^{\uparrow}$ and $G^{\downarrow}$ with the same predecessor/successor, we have to generate a segment indicator. The exact decision is given in Algorithm 9.4. The presented algorithm does not cover the assignment of distances to the different segments. An additional missing detail is the interaction with the unpacking process presented in Algorithm 9.2 and Algorithm 9.3. To propagate information throughout the unpacking process, we assign flags to the shortcuts that can make an arc both part of $G^{\uparrow}$ and $G^{\downarrow}$. As such, we only need to unpack it once but still get the benefit of detecting plateaux correctly.

It is not obvious that the results of Algorithm 9.4 correctly represent the alternative graph. Even though we create a single active segment for every successor at a vertex, it is not directly clear that it would be impossible to create two independent arcs that share a common segment. Over the course of Algorithm 9.4, we generate segments starting at important vertices. To ensure the correctness of our approach, we need to prove that this local setting can never generate distinct arcs for the alternative graph that actually share inner vertices. Finally, we need to ensure that no information on potential plateaux is lost. Both claims regarding the correctness of HiDAR are discussed in Theorem 9.3.

Theorem 9.3 (HiDAR correctness)**.** *The segment generation process introduces no parallel arcs. The start vertices and the end vertices of all plateaux are marked correctly. Every arc of the alternative graph is represented by a single active segment, identified by the minimal ID over all shortcuts containing it.*

Algorithm 9.4 (Schematic) Segment Generation

Input: A list S of shortcuts to a middle vertex v
Output: A list of segment indicators

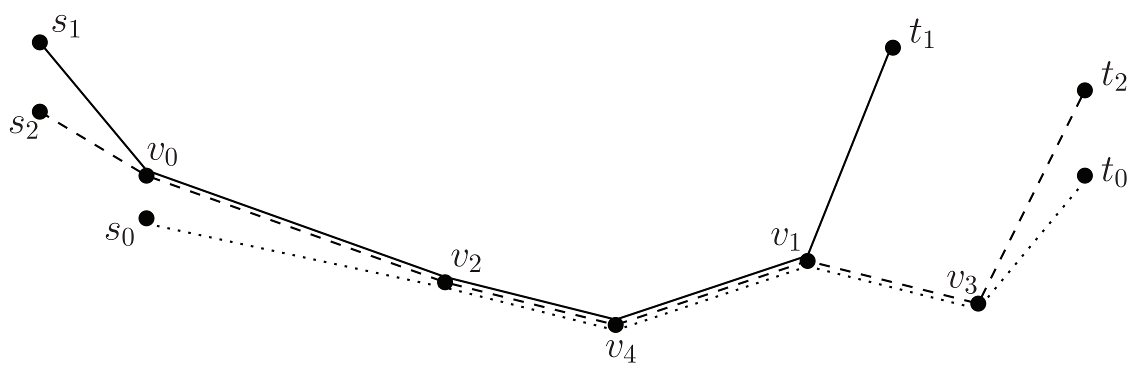
```
1:  $Pred := \{u : u$ is predecessor to $v$ for some $s \in S\}$    ▷ All predecessors of $v$ in $S$
2:  $Succ := \{u : u$ is successor to $v$ for some $s \in S\}$    ▷ All successors of $v$ in $S$
3:  $imp := |Pred| + |Succ| \geq 3$    ▷ More than a single predecessor/successor?
4:  $Seg := \emptyset$    ▷ The list of output segments
5:  if $imp \wedge |S| \geq 2$ then    ▷ No segments required if no other shortcuts touch $v$
6:      for all $w \in Succ$ do    ▷ Handle each successor on its own
7:          $\widehat{S} := \{s \in S : w$ is successor to $v$ in $S\}$
8:          $\widehat{s} = \min_{ID}(s \in \widehat{S})$
9:          for all $s \in \widehat{S} \setminus \widehat{s}$ do
10:             $Seg := Seg \cup \langle \texttt{ID}(s), v, \texttt{false}, \texttt{false} \rangle$    ▷ De-active all other shortcuts
11:         end for
12:         $Seg := Seg \cup \left\langle \texttt{ID}(\widehat{s}), v, \exists \widetilde{s} \in \widehat{S} : \widetilde{s} \in G^{\uparrow}, \exists \widetilde{s} \in \widehat{S} : \widetilde{s} \in G^{\downarrow} \right\rangle$
13:     end for
14: end if
15: return $Seg$
```

The proof of Theorem 9.3 relies on the correctness of the important vertex definition as well as the contraction order employed in the CH process. The order in which vertices are contracted defines a total order over all vertices and we can utilize it to correctly schedule the work in the extraction process.

Proof. Due to the total order induced by the CH contraction process, Theorem 9.3 holds true if all important vertices are discovered and the process does not create parallel arcs where none exist. The discovery of all important vertices follows directly from the previous interpretation of running Dijkstra's algorithm on a modified graph. Processing the arcs in the reverse contraction order of their middle vertices, we are guaranteed to process all shortcuts that contain a given vertex at the same time.

For the generation of segments, we make use of the fact that for any two arcs that share a common part, there exists a first and a last common vertex (follow along Figure 9.4 for better understanding of the arguments of this proof). We only discard shortcuts that are completely covered by higher priority tasks (compare Figure 9.4d and Figure 9.4e). As a result, no arc can be lost during the extraction process. If a task t_1 enables us to discard a task t_2, the associated flags from task t_2 are carried over to task t_1 and no information is lost, even though only one of the tasks will be completed. As long as a shortcut contains information not present in another shortcuts as well, the respective task will not be pruned.

What is left is proof that the generated segments translate to a correct representation of all unpacked paths encoded in the initial tasks. As can be seen in Figure 9.4, a task



(a) *Graph*

v_1	s_0	t_0	0	T	F
v_0	s_1	t_1	1	T	F
v_0	s_2	t_2	2	F	T

(b) *Initial Tasks*

v_1	v_0	t_1	1	T	F
v_1	v_0	t_2	2	F	T

(c) *Tasks generated at v_0*

v_2	s_0	v_1	0	T	F
v_3	v_1	t_0	0	T	F
v_3	v_1	t_2	2	F	T
v_2	v_0	v_1	2	F	T
v_2	v_0	v_1	1	T	F
v_2	v_0	v_1	1	T	T

(d) *Tasks generated at v_1*

v_4	v_2	v_1	0	T	F
v_4	v_2	v_1	1	T	T
v_4	v_2	v_1	0	T	T

(e) *Tasks generated at v_2*

1	v_0	T	T
2	v_0	F	F

0	v_1	T	T
1	v_1	T	F
2	v_1	F	F

0	v_2	T	T
1	v_2	F	F

0	v_3	T	F
2	v_3	F	T

(f) *Generated segments; each segment generated at the depicted vertex*

Figure 9.4: *Generation of segments and tasks on small problem – consisting of three shortcuts – chosen to exemplify the process. Lower numbers translate to higher priorities (both for extraction order and for tasks). Grayed out tasks are temporary tasks – shown for illustration purposes – that are not actually created.*

may cause a deactivation segment which is followed by an activation segment later on due to a completely different task; in the figure the task that extracts the shortcut with ID two is deactivated at v_0 by the task extracting the shortcut of ID one. The coverage in terms of the segments is handed over to the task with ID zero at v_1 and the task is finally activated again at v_3. The argument for this process is simply based on the first and last common vertex. When a task t is first deactivated at some vertex v, then v is the first common vertex with another task. These two tasks also have a last common vertex w. Either we create an active segment for task t at vertex w, or – in the case that another task of higher priority is present – transfer the coverage over to this higher priority task. In the case of the transfer, w acts as new first common vertex between t and the new task and the last common vertex argument is postponed for later on but remains valid[3]. □

From the list of shortcuts and the segment indicators, we finally construct an alternative graph. We scan the segment indicators associated with every shortcut and generate arcs for every active segment. Their length can be directly computed from the segment indicators. Every vertex that is part of an active segment indicator is selected as a vertex for the alternative graph.

9.1.5 Artificial Shortcuts

A final piece of information is still missing from our algorithm. Until now, we only considered middle vertices and did nothing to handle the different origins and destinations of the initial shortcuts. We can easily integrate them transparently into the full process, though. Our algorithms as presented so far consider all shortcuts in the order of their middle vertices. To correctly integrate all arcs in this process and to process their origin and destination correctly, we simply wrap each arc into two artificial shortcuts that have the origin and the destination as their respective middle vertices. As respective origin and destination of the artificial shortcuts we introduce new dummy vertices to the graph.

This extension comes cheap as we only require a simple scan over all arcs from the first phase. Due to the small CH search spaces the number of arcs is manageable. To integrate the artificial shortcuts into the unpacking process, we only need to append information for the new vertices and their adjacent (artificial) shortcuts to the data that guides the unpacking process. This augmentation presents the final step of our alternative graph generation algorithm.

9.1.6 Additional Processing

Finally, our algorithm performs a filtering step to keep the processing during the execution as simple as possible. In this effort, we reduce the overhead required later in

[3]The order in which these local decisions are made does not matter for the correctness.

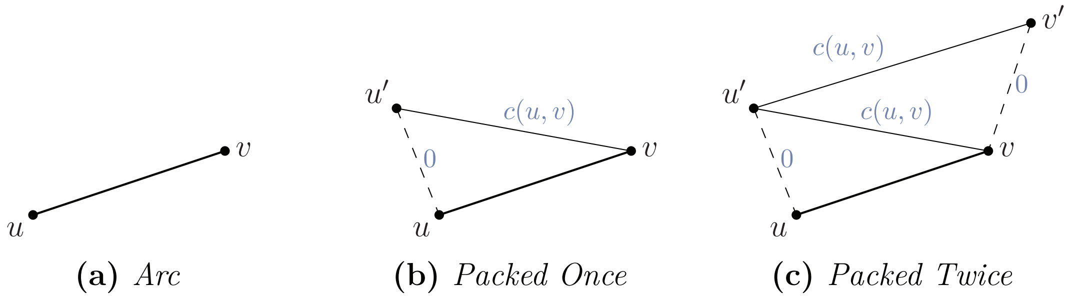


Figure 9.5: *Creation of artificial shortcuts to an arc (u, v). The additional vertices u', v' are connected to u, v via arcs of weight zero. The further process describes a straight forward contraction (see Algorithm 4.1) without any witness-searches.*

an alternative route extraction.

During the computation of the shortest-path distances, we prune all arcs from the search space that offer a larger distance than desired[4]. Vertices with larger values cannot influence the quality of our alternative graph in a positive way, as they would fail the admissibility criteria anyhow.

As a second optimization, we perform some additional post-processing that further hastens the alternative route extraction later on. Even though route extraction is not part of HiDAR, we still assume the minimal quality criteria with respect to the definition of Abraham et al. (see Definition 5.3). We consider them by reducing the alternative graph to only contain plateaux of viable length in comparison to their associated detours.

For this filtering process, a localized view of the alternative graph is not enough. We tackle this problem with a DFS algorithm that keeps track of the current length travelled along a plateau and the distance from the source; see Algorithm 9.5 and Figure 9.6 for reference.

The tree-like structure of the alternative graph, as long as we restrict it to the arcs from the forwards search, guarantees our DFS to always find correct values for the length of our plateaux. The moment our algorithm discovers the end of a viable plateau, we track the arcs of the backward tree to the target. Along the way, we mark which parts of the tree have to be remembered. During the backtracking steps of the DFS, we also mark the paths that lead to a plateau that has been deemed long enough.

When tracking the paths to the target, we could end up performing work quadratic in the number of vertices. We prevent this by marking the vertices appropriately, which ensures that we only touch every arc in the graph once.

[4] $\mathcal{D}(s,v) + \mathcal{D}(v,t) \geq (1+\epsilon) \cdot \mathcal{D}(s,t)$.

Algorithm 9.5 Depth First Search Filtering

Input: An uncompressed alternative graph $\mathcal{H}$, based on a set of forward ($\overrightarrow{A}$) and backward ($\overleftarrow{A}$) arcs, distances ($\mathcal{D}[\cdot][2]$) from the source s and to the target t for every vertex

Output: A compressed alternative graph

```
procedure MARK_PATH($v$)
    if not marked[$v$] then
        $marked[1][v] = $ true
        MARK_PATH($w : (v,w) \in \mathcal{H} \in \overleftarrow{A}$)                ▷ Exactly one exists
    end if
end procedure

procedure REC_DFS($v$,$pdist$)
    if $\exists a = (v,u) \in \mathcal{H} \in \overleftarrow{A} : a \notin \mathcal{H} \in \overrightarrow{A}$ then        ▷ Only one such arc can exist
        if $pdist \geq \gamma \cdot \mathcal{D}(s,t)$ then
            MARK_PATH($v$)                      ▷ Recursively mark path to target ...
            $marked[0][v] = $ true                  ▷ ... and via backtrack to source
        end if
    end if
    $mark = $ false
    for all $a := (v,w) \in \mathcal{H} \in \overrightarrow{A}$ do       ▷ No tracking of visited vertices in a tree
        $mark| = $ REC_DFS($order, w, next,$ is_plateau$(a)$ ? 0 : $pdist + \mu(a)$)
    end for
    $marked[0][v] \,| = mark$
    return $marked[0][v]$
end procedure

$marked[2][] := \{$false$\}$
REC_DFS$(0,0)$                                    ▷ Recursively calculate markings

return $marked[][]$
```

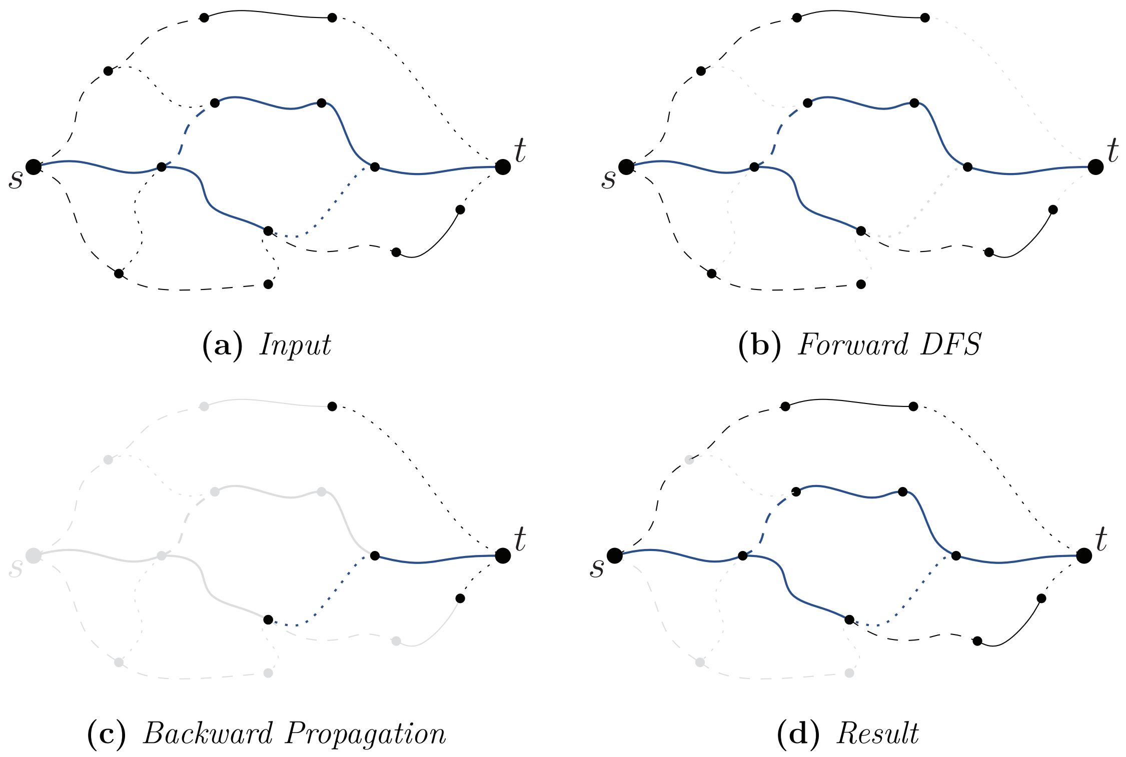


Figure 9.6: *Sequence showing the development of the extracted alternative graph through the DFS and the marked backwards paths to the final alternative graph of our algorithm.*

9.2 Experimental Evaluation

For the evaluation of HiDAR, we take a look at the different phases in the algorithm, both in terms of their workload as well as their measured running time. Since HiDAR is basically parameter-free and depends only on the used CH and the maximal delay accepted for an alternative route, the performance evaluation can be presented in only a few paragraphs. During the execution of our algorithm, we choose $\epsilon = 0.25$ – the value limiting the maximal allowed bounded stretch – in accordance to the related [ADGW13] and [LS12].

Workload. Even though the theoretical justification for CH is not yet fully understood (compare Section 4.1), its practical use on road networks is without question. In practice, the search spaces of a CH behave very well and grow slow with respect to the Dijkstra rank. We can observe this behavior in Figure 9.7 for the size of the explored search spaces as well. The pruning mechanism works surprisingly well, considering that we discover tentative distances later than the usual CH query – remember that we need to generate the search space in reverse contraction order for the following sweep. The selected number of vertices for further processing remains very small. For the most part, the resulting graphs contain far less than a thousand vertices. We visualize the number of number of vertices and arcs in our output in Figure 9.9.

The workload during the processing of the different tasks, as well as the generated segments, grow stronger than the selected vertices. They behave very similar to the search space, though. Luckily, the amount of vertices and arcs encoded within the shortcuts remains manageable for all distances (see Figure 9.8). The resulting alternative graphs grow rather large. The employed filtering method, however, reduces the graphs to a very reasonable size. The size still exceeds what would be suitable to present to a human. It remains small enough for further processing using expensive algorithms for the extraction of alternative routes, though. The number of vertices and arcs to represent all plateaux as well as the possible paths is relatively low and stays far below a thousand, even for extremely long travels. For a typical journey, we would expect roughly around a hundred vertices and arcs.

Running Time. The running time required for HiDAR is, at least for the longer journeys, dominated by the extraction process that can end up processing close to 70 000 different tasks (or arcs). In spite of the potentially high costs for the extraction process, the running time for HiDAR remains well within the limit for an interactive application, not exceeding fifty milliseconds. All reasonable queries are well within ten milliseconds. The pruning mechanism, while effective, only proves relevant to short range queries as the shortest-path query itself plays only a minor role in the total cost of the algorithm. The high cost for the extraction process fully justifies the approach to directly create the compact representation during the extraction process over a variant that performs an additional parse of the extracted data.

Extracting a series of alternative routes from the small alternative graph is easy: The precomputed plateaux can be chosen and selected and the adjacent paths have to be traced back to the current graph for overlap. This requires only a few hops, though. The required running times are irrelevant, even for a full exploration, in comparison to the previous extraction overhead and are well within a single millisecond for all queries.

Quality. In terms of quality, our implementation directly compares to previous via-vertex approaches, while offering higher success rates, closing the gap to X-BDV. In comparison to other via-vertex approaches for the success rates, we close the gap to the far slower 3-REV by fifty percent, ninety-nine, and a hundred percent for the first, second, and third alternative route. The gaps to the upper bound in the form of the full plateau method (X-BDV) are nearly identical. At the same time, our algorithm offers nearly identical running times to 3-CHV and is up to five times as fast as 3-REV. We still loose out on some of the possible via-vertexs, though.

As can be seen in Figure 9.11, the alternative routes found by HiDAR offer very

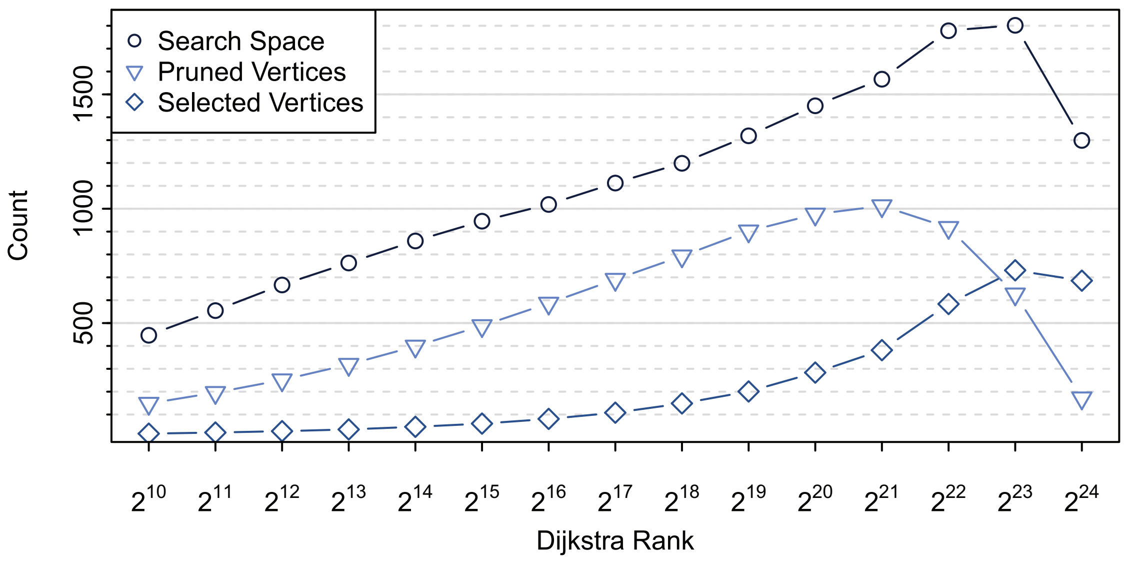


Figure 9.7: *Workload of HiDAR in terms of vertices over different Dijkstra ranks. Even though just a heuristic, the pruning mechanism works well.*

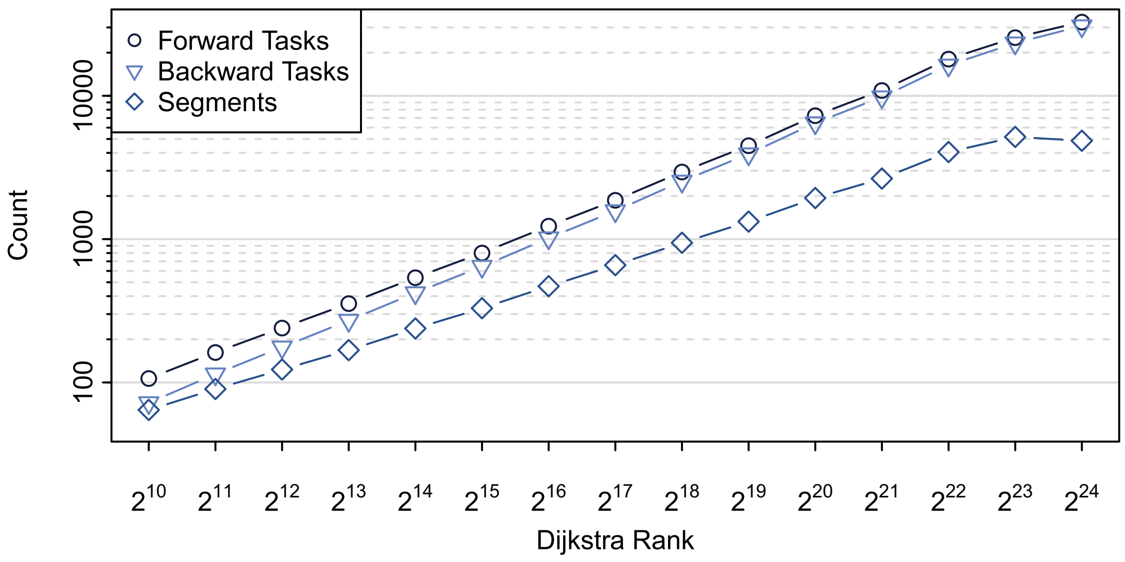


Figure 9.8: *The number of tasks reflects the growing number of arcs represented in shortcuts. Our pruning and compaction mechanisms manage to keep the number of generated segments small, though. The final graph contains even fewer arcs as many of the generated segments are inactive.*

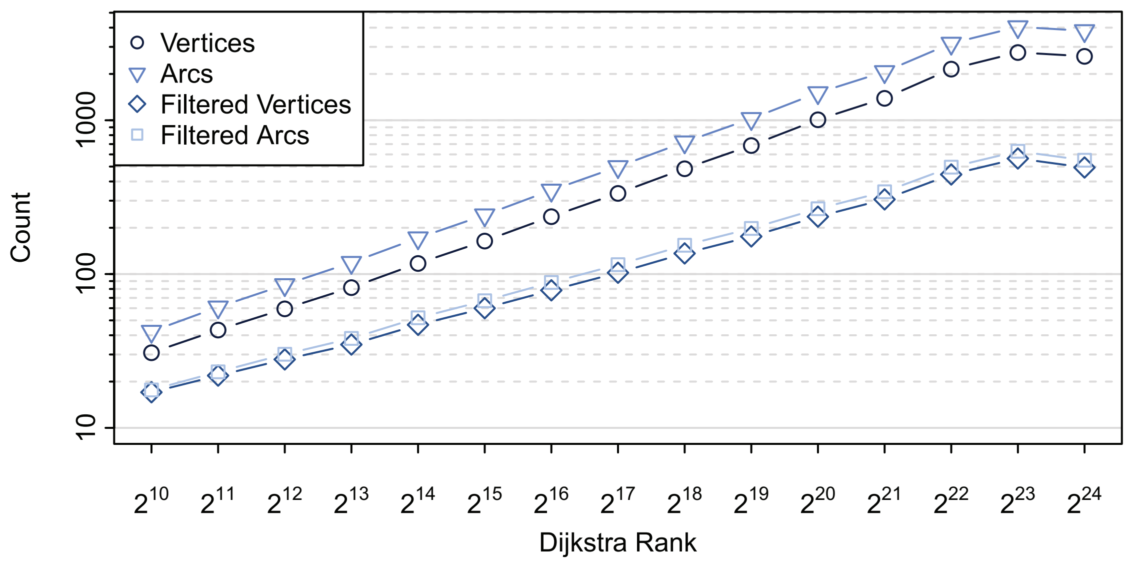


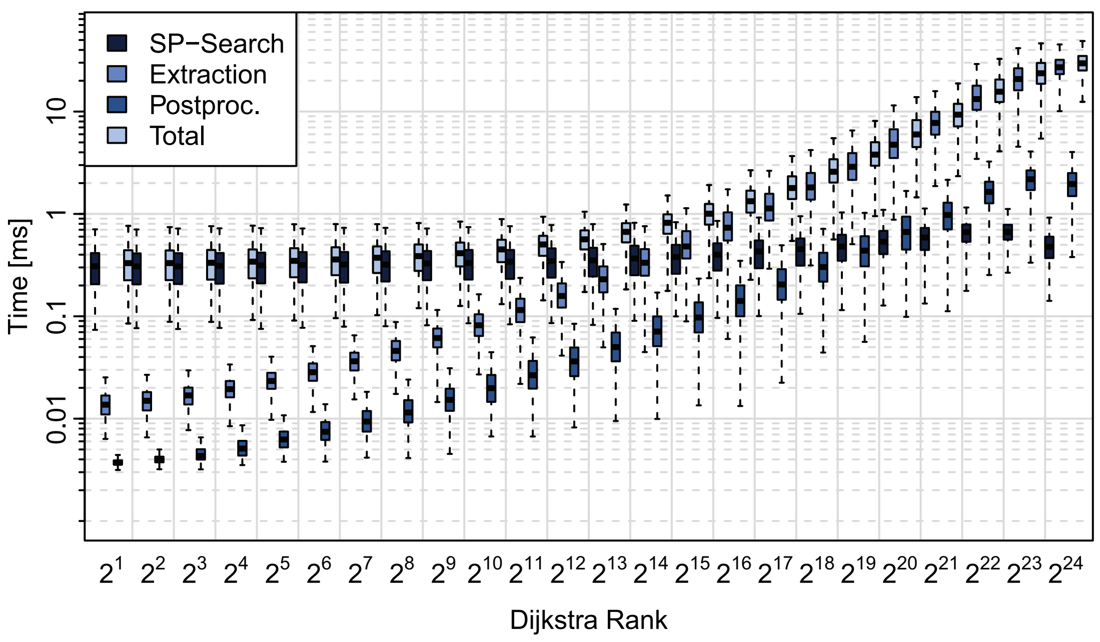
Figure 9.9: *Juxtaposition of the filtered and unfiltered alternative graphs created by HiDAR. The result of our algorithm presents a manageable graph that can easily be exploited for further considerations. The size of the output does not exceed a thousand vertices.*

slow stretch (ϵ). As a direct result, the values for local optimality (β) are also very good. These high values come at the price of a large amount of limited sharing (α), though. As with most CH based techniques, we can see a drop-off in quality at Dijkstra rank 2^{24}. This drop-off affects the success rate, $\mathcal{D}_{tot}$, as well as the number of first alternatives we can find.

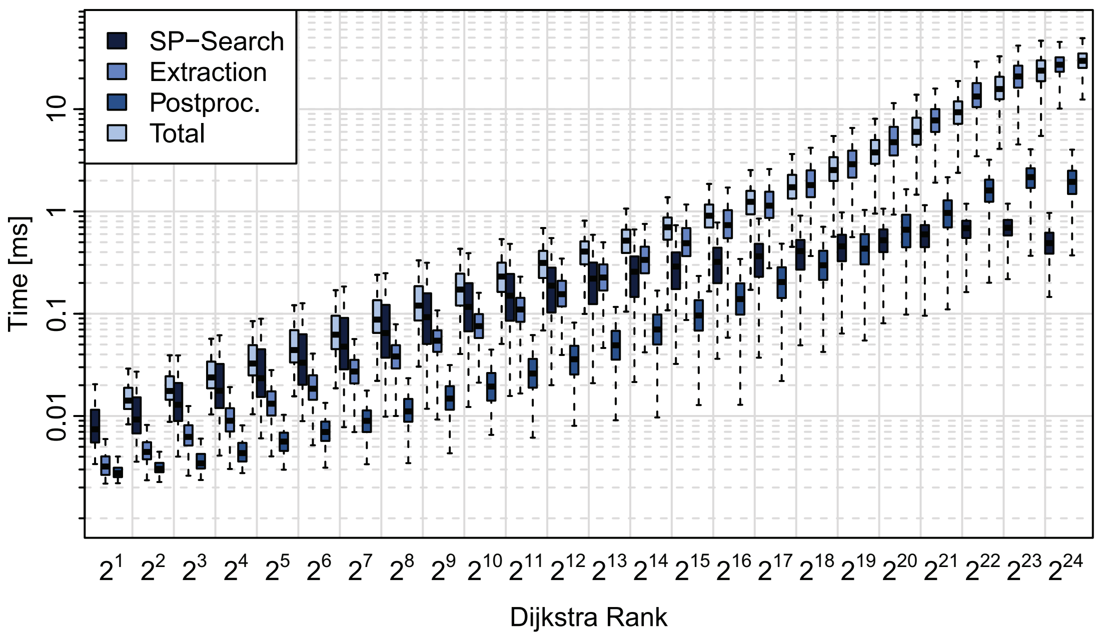
9.3 Conclusion

The HiDAR algorithm can be seen as a middle ground between the initial via-vertex approach [ADGW13] and the calculation of alternative graphs in [BDGS11]. In contrast to the work discussed in [BDGS11], we present the – to the best of our knowledge – only algorithm that actually computes all paths that form the alternative graph on a speed-up technique at once. Even though Bader et al. focus on full graphs in their paradigm, they only present methods based on single shortest-path computations.

While we follow the general idea of Abraham et al. [ADGW13] in what constitutes a viable alternative route, we manage to alleviate one of the major problems that comes with the calculation of a collection of paths: the high cost that is associated with making an informed choice during the path selection. We manage to perform all viability checks required over the course of a single run. As such we manage to stay competitive for a well informed choice. In this, we allow for more sophisticated ways to select alternative routes. Our algorithm is able to evaluate all candidates in full and to report the best alternative route, instead of the first viable one.



(a) *Without pruning*



(b) *With pruning*

Figure 9.10: *Running time of HiDAR and its different components over a variation of different Dijkstra ranks. Especially lower Dijkstra ranks benefit from the pruning mechanism.*

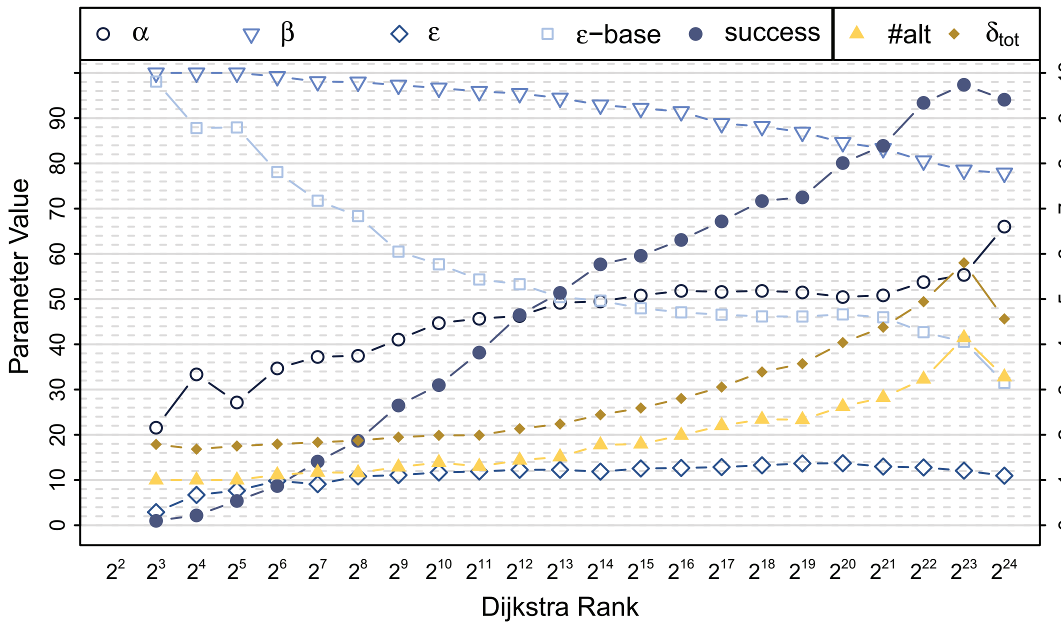


Figure 9.11: *Development of the quality parameters, according to Definition 5.3, for HiDAR.*

Chapter 10

10 Alternative Routes via Segment Exclusion

Vertex exclusion, the dual approach to the via-vertex paradigm, offers a link from the via-vertex approach to the penalty method. We study an intermediate approach, the exclusion of segments, for its potential and discuss some extensions. This unpublished work is a proof of concept implementation of my own and uses the mechanisms of CRP-π. The presented concept does not offer the same quality as CRP-π but reduces the workload. In addition, it could offer a way to introduce penalization to Contraction Hierarchy (CH)-based routing engines. The evaluation on a CH is a topic of future work, though.

Although seemingly unconnected at first glance, the exclusion of vertices introduces a connection between the penalty method and via-vertex algorithms. The exclusion of a single vertex[1] can be described as dual to the via-vertex paradigm. At the same time, it is a special form of penalization. In fact, our implementation uses the same code for penalization as CRP-π. The main difference is given in our approach of directly choosing a very high penalty. This penalty has to be chosen high enough as to essentially block off the vertex while still keeping the graph connected to avoid special cases. Due to this near infinite penalty associated with segment exclusion, we call our algorithm Customizeable Route Planning with Infinite Penalization (CRP-∞). As already stated, our implementation uses CRP-π and simply applies a very high penalty factor.

Basically, the concept addresses two shortcomings of the previous techniques at the same time. The main problem in the via-vertex method consists of discovering potentially good candidates (c.f. Section 5.2.3). To find candidates, current methods [ADGW13] relax the search criteria, using a heuristic that depends on up to k parent vertices. While heuristic relaxation of the hierarchic properties can yield surprisingly good results, knowing about all possible candidates is preferable. Especially in a

[1]Our implementation actually considers a series of vertices.

production environment, a heuristic that depends on a CH order can produce unwanted inconsistencies. If a dependency between the ordering mechanism and the discovery of candidates exists, different preprocessing runs can offer different results.

While the penalty method is consistent in our implementation of CRP-π, the the penalization is depending on the metric and can be hard to tune correctly. Some special cases also require adaptive penalties. For example, if we consider a near optimal detour for a sub-path, see Figure 10.1, we might need small penalties to find it. If we penalize the full path too strongly, we can end up discovering a worse alternative instead. In advance, it cannot be known what constitutes a good penalization scheme to a path, though. In comparison, see Figure 10.2, exclusion can handle the same situation well.

Furthermore, we currently require a lot of cores to effectively lower the cost of a single iteration. If we apply parallelism, it would be more reasonable to test multiple candidates at the same time. However, the iterative process of the penalty method prevents this.

By avoiding regions of the path instead of penalizing a full path, we can solve many of these problems: we can both test multiple candidates in parallel and can avoid finding good penalization schemes. Furthermore, we avoid the difficult discovery of via-vertexs. Finally, the restriction to a local penalization might allow an implementation on a CH as well.

These benefits do not come without their own attached drawbacks, though. The highest cost to pay, when we exclude full segments from a path, can be in the form of very high bounded stretch. Our experiments show, however, that high stretch is barely a real concern.

Parametric Penalization. The exclusion of segments is not the only possible way to potentially handle penalization better. Another option would be to consider advanced penalization schemes that combine multiple forms of penalization. We could, for example, start the algorithm using low penalty values and increase the values over the progress of the algorithm. As we have seen in Section 8.1 and the accompanying experiments, we currently require to keep the number of iterations low. The penalty factor itself has only little effect on the cost of an iteration. Advanced penalization schemes might required to increase the number of iterations, resulting in slower running times.

10.1 The Concept of Segment Exclusion

The discovery of segments to consider is straightforward: the initial computation of a shortest path provides all potential candidates. Since we always search for the shortest path anyway, their discovery implies no additional computation overhead.

Even though a study of the dual to the via-vertex approach – excluding a single vertex – would have been nice, we found it to be unreasonable. Although working well

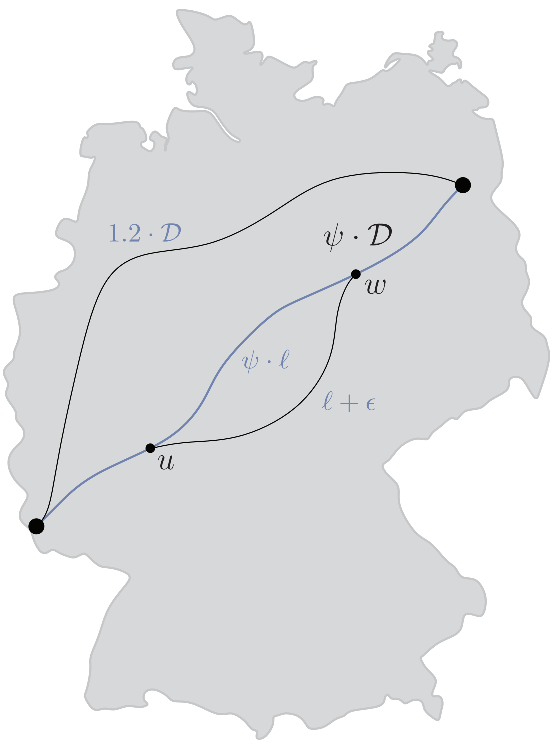


Figure 10.1: *Illustration of the idea behind our consideration of segment exclusion. The discovery of the alternative between u and w highly depends on the form of penalization and has a good chance to be missed. In cases of a large value for ψ, the path with the deviation between u and w could still end up more expensive than the second and longer alternative path. This is the case for $\psi > \frac{1.2 \cdot \mathcal{D} - \ell - \epsilon}{\mathcal{D} - \ell}$. Small values for ψ require a longer time to discover the long path, though.*

in many situations – for example when blocking access to an inner city highway –, it does not work in general. For the most part, the exclusion of a single vertex results in a local detour that offers only little variation in the route. Therefore, we decided to use an intermediate approach: instead of penalizing the full path or excluding a single vertex, we consider a segment for exclusion. For our purpose, we define a segment as a consecutive set of vertices on the shortest path[2].

This intermediate solution can still offer multiple benefits over the via-vertex and the penalization approach. In segment exclusion, we vary the metric in a very simple form, compared to advanced penalization schemes. To realize the exclusion of a segment, we can choose a high additive or multiplicative penalty. Doing so, we essentially forbid the usage of any blocked segment, as long as it is not required to maintain a connected graph. Therefore, we can even reuse our implementation of CRP-π.

[2]We explicitly chose vertices over arcs, as the exclusion of vertices prevents the discovery of very short detours.

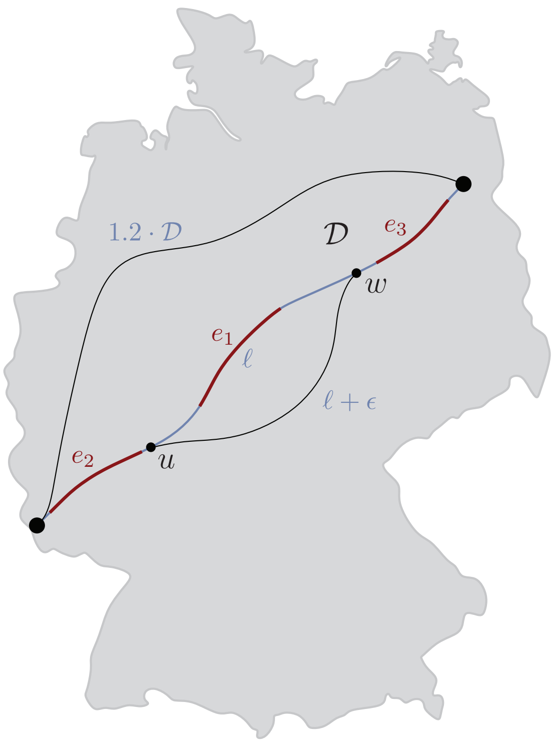


Figure 10.2: *In contrast to Figure 10.1, the variations on what to exclude result in the discovery of all reasonable alternative routes. Excluding* e_1*, we discover the shorter alternative path of length* $\mathcal{D} + \epsilon$*. Excluding* $e_{\{2,3\}}$*, we can only discover the longer path of length* $1.2\mathcal{D}$*.*

While the approach is very similar to the penalty method, it is in many ways preferable. The most obvious reason is in the computational overhead. Whereas the customization of the full path requires a lot of cells to be considered, a small segment is restricted to only a few of them.

In addition, there is no benefit in testing multiple increasing penalties. Vertices in the segment are either not used or essential to the query (e.g. due to a bridge or a border between countries). Blocking a segment a second time by applying even stronger penalties does not change this result.

Furthermore, we can perform many independent queries in parallel.

The iterative process of the penalization still offers some benefits. The application of penalties to detours does not come as naturally for the blocking approach as it does for penalization.

Segment Selection. Using segments of consecutive vertices for exclusion offers an interesting trade-off between the number of starting locations to consider and the discovery of new routes. As formulated in Definition 5.4, the amount of allowed sharing

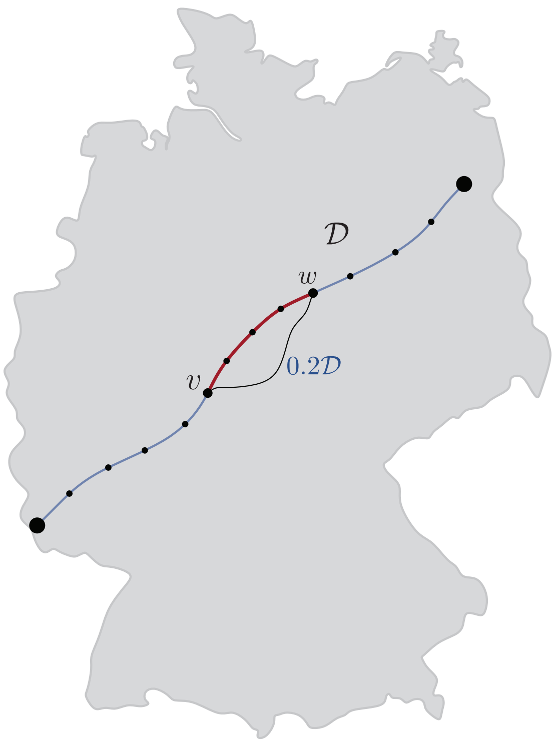


Figure 10.3: *Schematic illustration of sufficient blocking locations. Considering a blocked segment length of* $0.1 \cdot \mathcal{D}(st)$ *and an alternative to at least* $0.2 \cdot \mathcal{D}(s,t)$ *, the locations marked by the dots are sufficient to find most reasonable alternative routes.*

between an alternative route and the shortest path is limited. Typically, the research community considers a maximum of eighty percent of sharing between the two paths as viable. This, however, requires the alternative path to deviate from the shortest path for a segment (σ) with $\mathcal{L}(\sigma) \geq 0.2\mathcal{D}(s,t)$.

While we might find a good alternative route at practically every vertex, the pure number of possible starting locations prohibits an exhaustive test. Luckily, we can approximate the result of such an exhaustive exploration in a reasonable way: We consider segments that overlap by a fixed amount of their length. We provide the motivation for this argument in Figure 10.3, under the assumption of the usually required difference between the shortest path[3]. If we consider blocking a segment σ of length[4] $\mathcal{L}(\sigma)$, we test every segment that starts at distances equal to a multiple of $\frac{\mathcal{L}(\sigma)^2}{\alpha\mathcal{D}(s,t)}$. For example, if we consider $\mathcal{L}(\sigma) = 0.1\mathcal{D}(src,t)$ and $\alpha = 0.2$, we block every segment that starts at multiples $0.05\mathcal{D}(s,t)$. We refer to this method as *a-priori division*. For every possible deviation of length $\alpha\mathcal{D}(s,t)$, this guarantees full segment

[3]Typically, the sharing is limited to eighty percent [ADGW13] ($\alpha = 0.2$).
[4]$\mathcal{L}(\sigma) \leq \alpha\mathcal{D}(s,t)$.

to be chosen between its end-vertices. In our implementation, the twenty different starting positions are also our limit for segments that are longer than $0.1\mathcal{D}(s,t)$.

Figure 10.3 illustrates this in the form that any segment that consists of at least five dots, has an inner segment of three dots in length[5]. The alternative that is starting at the vertex v, cannot be discovered with when considering a segment starting at v . The next starting location, however, is completely contained within the bridged segment between v and w, though. Of course, we cannot guarantee to find all possible viable deviations this way. In case we miss out on one of the deviations, the one that we found instead has to offer a shorter or equal detour, though.

10.1.1 Implementation Details

Handling multiple segments in parallel is essential to the success of CRP-∞. Since the number of cells to be processed for a segment is very low, we cannot parallelize over the tasks of a single segment. Instead, we have to perform multiple tests at a time. The data structures of the CRP algorithm make it easy so implement a segment-penalization scheme with very low memory overhead. Given short segments, the number of cells affected by the change remains very small (compare Figure 8.7 for the count on a full path). A query algorithm accessing these cells only requires its own subset of modified cells. We can keep an updated versions of the required cells and prioritize their usage when accessing cells. This requires only a minimal change to the general CRP algorithm and a few MB of storage [6].

Further Optimization Potential. In the implementation using CRP-π, using a maximal level to consider sacrifices a lot of optimization potential. Since most of the cells on the highest level remain valid, we could improve the performance a lot by only descending into dirty cells. Figure 10.4 presents an illustration of this setting. We leave this task for future work.

On a side note, our parallel implementation of CRP-∞ uses a thread pool based on features of the standard template library[7]. We implemented the pool in reaction to some bottlenecks introduced by our straightforward application of the Open-MP multi-threading.

Preselecting Promising Segments. In comparison to the full penalty method, the workload for CRP-∞ is small. We only have to test a few possible starting locations (e.g. twenty locations when blocking segments amounting to ten percent of the shortest path

[5]This scenario equals a detour of at least twenty percent and a segment of length ten percent.

[6]Our experimental evaluation keeps a full set of cells for each update thread. This is, however, more costly in terms of computation, as restoring cells requires to copy a full cell instead of setting a pointer.

[7]The Standard Template Library (STD) library bases its parallel features on `pthreads`.

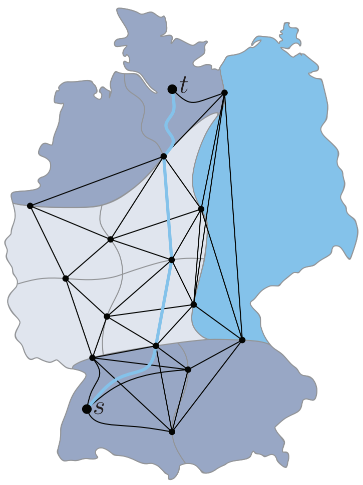


Figure 10.4: *Illustration of a fine grained query mechanism, only descending into dirty cells. The large cell on the right is used. Only in the updated part we use the lower level during the query.*

in length). In addition, we can perform many of these tests in parallel. Nevertheless, we would like to to reduce unnecessary computations as much as possible.

Even if the number of threads does not matter, we cannot simply assign a single thread to each of the tasks, as the memory bandwidth is limited. In a production environment, the use of excessive parallelism is undesirable, as well.

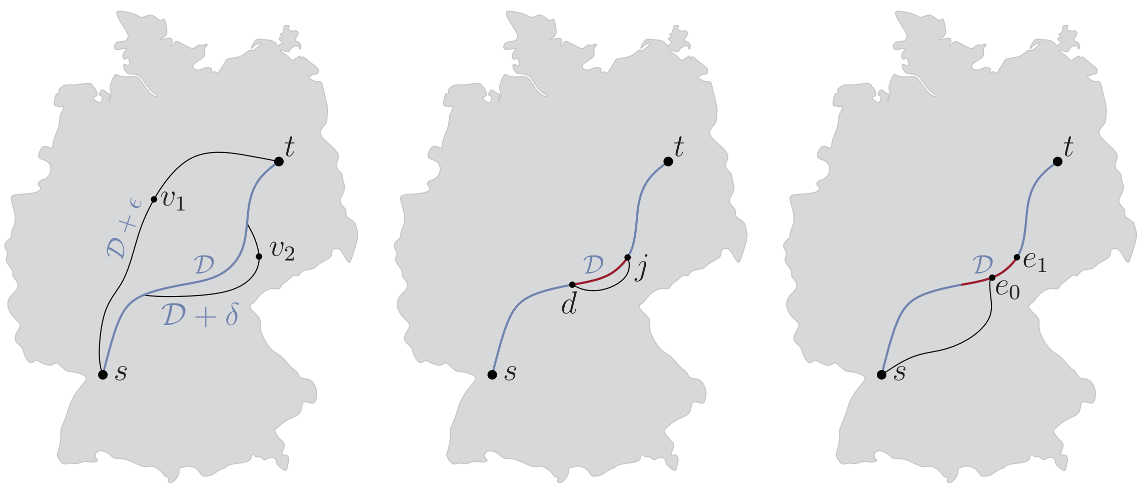
Next to the previously discussed a-priori division, we can take a divide and conquer approach as well. By recursively generating tasks, we can take the information discovered in every step into account. If, for example, we find a very good detour for a large part of the shortest path, it is very unlikely to find a different detour by blocking another segment between its end-vertices.

Figure 10.5a illustrates this setting: if $\epsilon < \delta$, no segment exists for which we can find the path using v_2. At the same time, the figure presents an example of a case in which the iterative approach of the classic penalization has a better chance of discovering both paths[8].

Such a detour does not necessarily have to be an alternative to the full shortest path, but rather to a significant portion of it. For the ranges next to the provided alternative, we continue in the normal recursive fashion.

The next case we consider is the discovery of essential vertices. If the penalized path contains some of the vertices that have been blocked off (see Figure 10.5c), we can assume that the respective vertices are essential to the shortest path. This definition of essential is not as strict as the general concept of bridges in graphs, for which the removal corresponds to the graph becoming disconnected. It is closely related, though.

[8]We can improve the exclusion method for this kind of setting, though. We describe how to do this after the current considerations.



(a) *Good Detour* **(b)** *Local Detour* **(c)** *Essential Vertices*

Figure 10.5: *Illustration of some scenarios we can encounter during the recursive generation of tasks.*

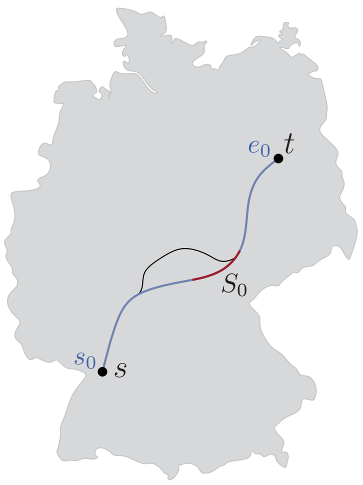
A vertex or arc is essential if its removal from the graph results in a significant detour; for example: the case of a bridge in a graph would result in a detour of infinite length. Due to our high penalization factors, a significant detour is equivalent to a penalized arc to be part of the discovered path (e_0 to e_1 in Figure 10.5c). In case we discover such an essential range of vertices, we restrict the divide and conquer progress to omit these vertices.

The final scenario we can encounter is the local detour (see Figure 10.5b). If we discover a very high value for bounded stretch, we can handle it in the same way as we do for the discovery of essential vertices.

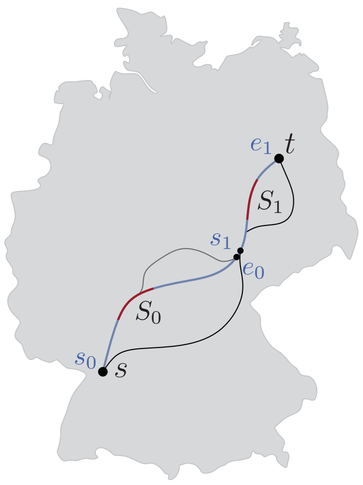
Divide and Conquer Discovery. In our implementation, we assign regions to different tasks. We describe the process for a segment length of $0.1\mathcal{D}(s,t)$ and limited sharing set to $\alpha = 0.2$. In the example of a single thread of execution, the initial region would be the full shortest path and we would consider a segment enclosing $\frac{\mathcal{D}(s,t)}{2}$ in the most balanced way[9]. After the task has been processed, we analyse the detour to decide how to generate further tasks. In case none of the formerly mentioned scenarios occurs, we generate a task to the left that is in the middle of the region $\left[0, \frac{\mathcal{D}(s,t)}{2}\right]$ and one for the region $\left[\frac{\mathcal{D}(s,t)}{2} + 1, \mathcal{D}(s,t)\right]$.

In case of a good detour, we simply stop the recursion. In the other cases, we adjust the regions to omit essential (or close to essential – compare the local detour) vertices. This method can be applied recursively until the respective regions get too small to place a blocked segment of the desired length. By assuring an overlap of half a segment

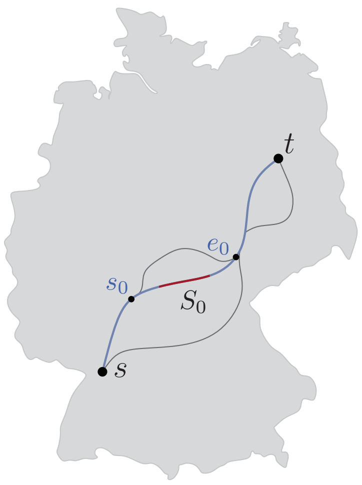
[9]In this case balanced refers to the ranges to the left and to the right of the named location.



(a) *Level 0*



(b) *Level 1*



(c) *Level 2*

Figure 10.6: *Recursive levels during the task generation.*

at the respective end vertices, we prevent our algorithm from blocking a segment more than once.

We illustrate the process in Figure 10.6. The initial segment chosen is S_0 in Figure 10.6a. The found detour crosses the blocked segment, influencing the choice of the recursive segments tested in Figure 10.6b, leaving a small gap between the respective end-vertices of the regions to be considered. After this initial first level, the range containing S_1 is considered too small for further recursion. Due to the different arcs on the path, we can encounter the same situation for one part of the range containing S_0 . In this example the effect results in a single range to be considered for recursion instead of two ranges. The different positions of the respective S_0 in Figures 10.6a and 10.6c can result in subtle differences in the found alternative. These differences can be problematic (c.f. Section 10.1.2) but can also be exploited (c.f. Section 10.1.3).

Heuristics. Instead of this divide-and-conquer approach, one could try to choose segments heuristically based on many different influences. Local detours, the level of different vertices in the CRP data structures, or even vertex-reach could be considered. They are a topic of currently ongoing research, though.

Recursive Extension. The case presented in Figure 10.5a indicates a possibility of further candidates if we recursively apply penalties to good detours. During the previously described divide and conquer approach, we remember which detours resulted in the pruning of sub-trees. For these detours, we perform additional steps after the initial algorithm has finished. We start by first excluding the middle third of the alternative – the one that we remembered earlier – from the search. Afterwards, we simply reinitialize the divide and conquer algorithm to continue at the previously pruned tree.

10.1.2 Potential Problems

As previously mentioned, one problem of this approach is that we can force the alternative route to start or end unreasonably. If we block off parts of the path completely, we could end up producing very high stretch values. We can avoid most of these locally bad decisions with a simple trick: we only accept detours that start/end in a certain minimal distance to the blocked segment. To be exact, we postulate a distance between the segment and the deviation vertices of at least one percent of $\mathcal{D}\left(s,t\right)$. This reduces the occurrences of high stretch values to almost none. We only found fifteen cases within our twenty-four thousand queries, for the most time offering at most a fifty percent increase on a very short segment. Even though these problematic stretches are very undesirable, other techniques suffer from them as well. Abraham et al. [ADGW13] report maximal stretch values of up to sixty-three percent increase (1-CHV, see Section 5.2.3), which is comparable to the fifty percent increase that we experience in these rare cases.

In the cases for which we found high values for the bounded stretch, the values were very close to the shortest path and described segments of very short length. We could probably corrected them by doing some local searches around the blocked segment. Due to the low rate of occurrence, we did not implement this, however.

10.1.3 Constructing an Alternative Graph

In the via-vertex paradigm, we compute a full graph in one query. In the setting of CRP-π, we computed a growing number of paths and extracted viable segments from the set. Here, we take yet another approach. Unlike our previous approaches, CRP-∞ computes many different paths in parallel.

The set of paths that we calculate can contain many similar ones, parts of which may be reasonable, parts of which might not be reasonable. We tried two different methods to construct a viable alternative graph. Our first idea to rank the different paths by a quality and select the best one for the graph did not work[10]. The reason for this lies within the previously mentioned segments with bad stretch. Considering the paths on their own, the rate in which we find locally high values for bounded stretch increases, even though not to unreasonable levels. Still, we can combine the segments in a way that limits the number of bad paths that we encounter. We add all discovered segments into a large graph. On this graph, we apply a simple filter to then remove some unreasonable arcs. If two segments are parallel to each other, we discard one of them, if they are too short on their own (less than the desired values for limited

[10]We considered the length, the distance between the deviation vertices and the blocked segment, as well as the stretch.

sharing). In cases of many arcs connecting a vertex to the shortest path, we keep only the shortest way to reach the vertex. Again, we only remove an arc if is too short to be considered a viable alternative on its own. The filters are closely related to the paradigm of thin-out, already described by Bader et al. [BDGS11].

This construction process yields a reasonable graph and discards only segments that could never contribute to a viable alternative route. We present an illustration in Figure 10.7.

Additionally, the figure shows that, even though not identical, the exclusion method and the penalty method both can yield very similar results. This similarity is visible in close to all the cases we encountered. It indicates possibilities for further studies, though, as some routes that could be considered viable are not yet discovered using our exclusion approach.

10.2 Experimental Evaluation

As our implementation is directly based on CRP-π, we keep the evaluation at this point very short. We do not reiterate our considerations on the different tuning settings and different components. The method is to be seen as a proof of concept. Timing measurements only give an impression in how far the different methods influence the amount of work to be done. We expect that far better running times are possible, for example by using a dynamic CH. At this point, we only present a small overview of the parts that differ from our implementation of CRP-π.

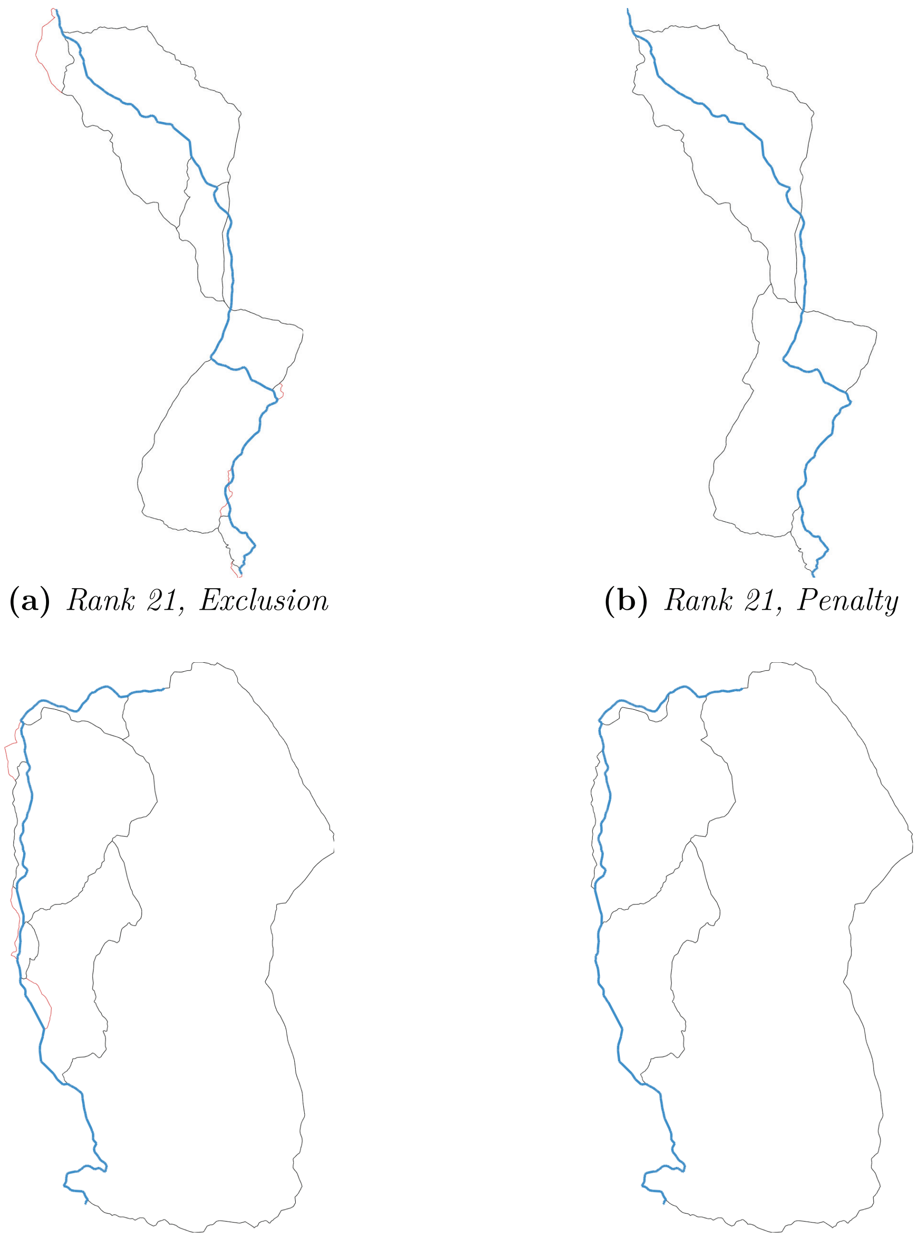
10.2.1 Running Time

Taking a look at the workload, it becomes clear that our approach has still a lot of potential to be exploited. In comparison to the number of segments we can add to the alternative graph, we process a far greater amount. Many of the tested segments do not offer a viable contribution to the alternative graph.

In the presence of good heuristics, we might be able to reduce this number considerably and achieve faster turnaround times as a result. In total, due to the low cost of a single iteration, the general workload remains manageable nonetheless.

Even though we perform far more tests than the ten iteration we require for CRP-π (see Section 8.1), every single one is cheap in comparison. For example, a segment of ten percent of the length of the shortest path also affects only around ten percent of the cells.

This reduction in workload can be seen directly in the performance of CRP-∞ using one or two CPU cores. Two cores already result in a similar running time to CRP-π on eight cores.. Using even more threads, we can outperform CRP-π.



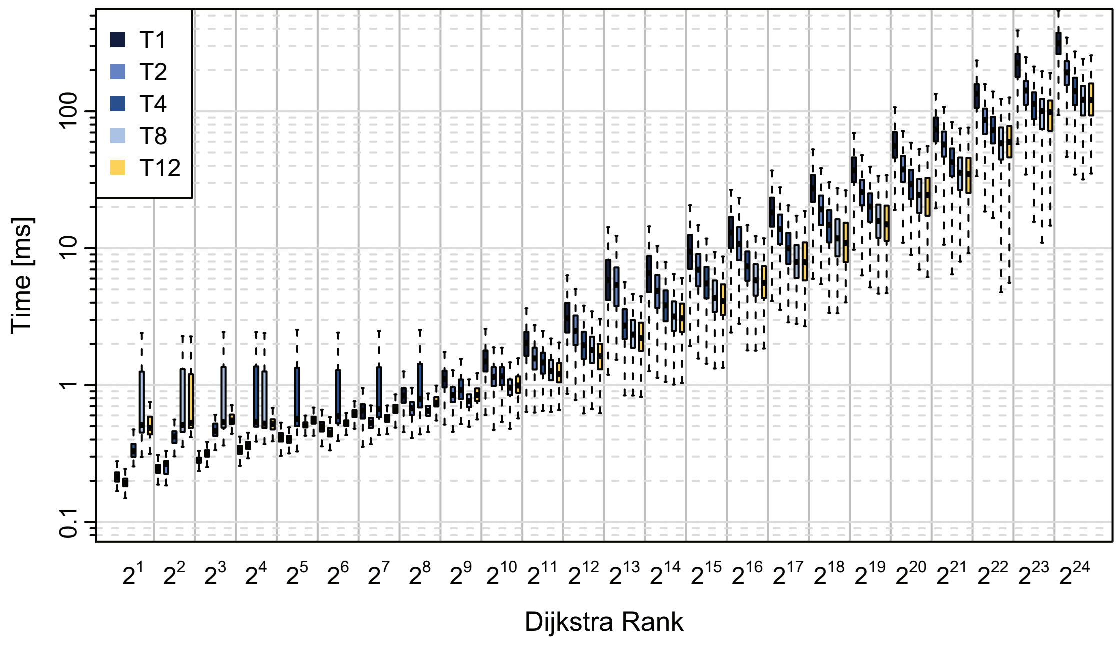
(a) *Rank 21, Exclusion*

(b) *Rank 21, Penalty*

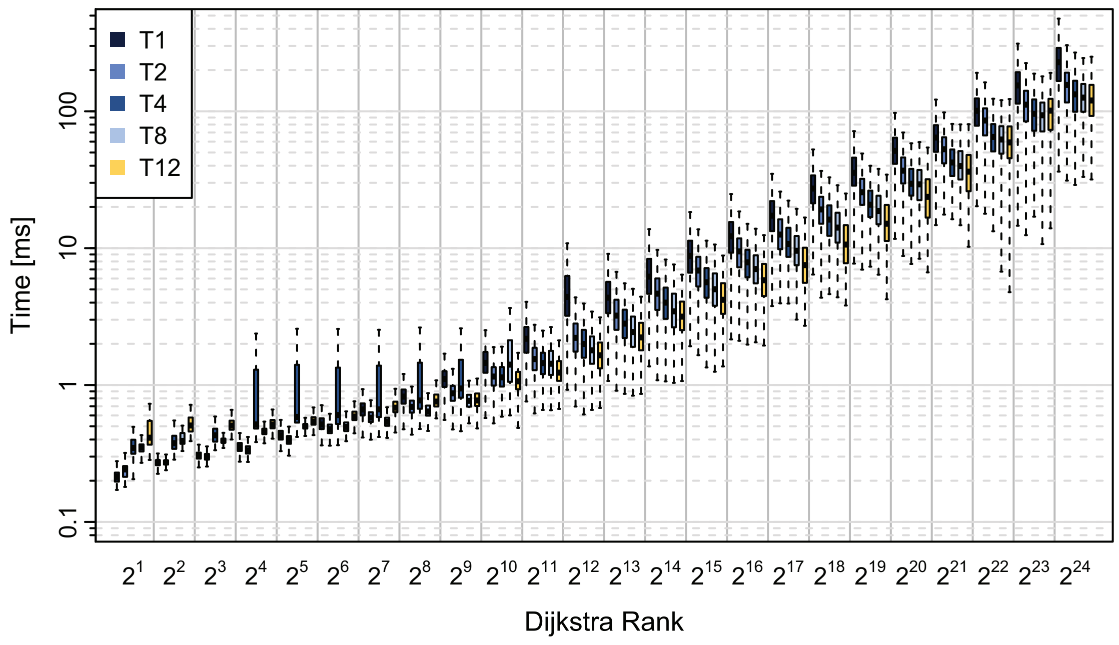
(c) *Rank 17, Exclusion*

(d) *Rank 17, Penalty*

Figure 10.7: *Examples of the generated graphs of our exclusion approach. Red arcs are only shown to indicate the discarded parts of the graph. For comparison, we show the same graphs as computed by CRP-π as well.*



(a) *A-Priori Division*



(b) *Divide and Conquer*

Figure 10.8: *Parallel execution of CRP-∞ on multiple cores. We only show experiments for M2, as the memory on machine two has already proven faster in the experiments for CRP-π. The segment length is chosen as ten percent of $\mathcal{D}(s,t)$.*

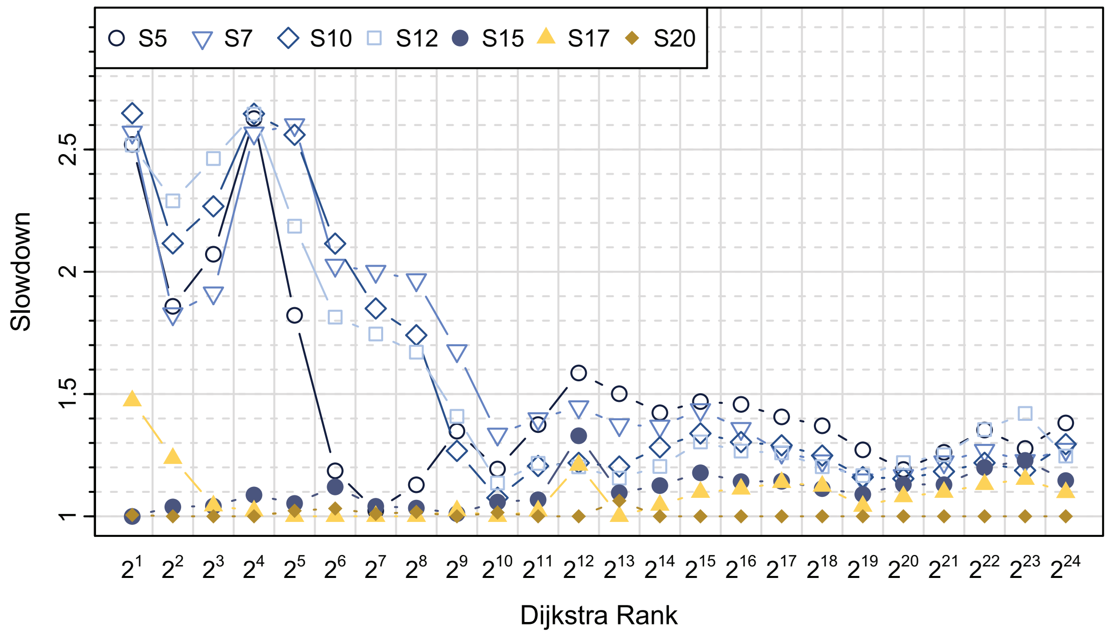


Figure 10.9: *Slowdown in comparison to the fastest run for different segment sizes (SX specifies a segment of $\frac{X}{100} \cdot \mathcal{D}(s,t)$ in length.*

Parallel Execution. Our measurements for the parallel execution of the algorithm, visualized in Figure 10.8, indicate that our algorithm is bound by the memory bandwidth, as long as we use more than four threads. Up to four threads, the parallel algorithm scales well, which is to be expected for a set of independent tasks. This effect is similar to the results that we experienced for the penalty method (CRP-π) and indicates potential for further improvements by reducing the memory footprint of our application.

Segment Size When it comes to running time, it turns out that the actual segment size barely influences our algorithm. This is a result of the direct dependence between the number of update tasks and the workload of a single task. In the end, both factors balance each other out. We visualize this effect in Figure 10.9, which shows only a slowdown of around forty percent between the slowest and the fastest setting. For this comparison, we ignore the effects of the short-range queries, as their running time is fast enough for all variants. In general, segment sizes result in faster queries.

Divide and Conquer vs. A-Priori Division When comparing Figure 10.8b to Figure 10.8a, we can see that the divide and conquer approach helps in speeding up the algorithm. The information gained in the process can actually reduce the computational overhead. Still, the low number of unique routes we discover in comparison to the large number of tested cases indicates that we could do even better with heuristics.

Nevertheless, the divide and conquer approach is a step into the right direction. The sequential performance is improved greatly in comparison to the a-priori division. Starting at four CPU cores, however, the two approaches break even.

10.2.2 Quality

When comparing our efforts for segment-exclusion to the general penalization (see Figure 10.10), we can identify some interesting developments. Apparently, the quality of the first chosen alternative for exclusion offers far greater values for local optimality (β) than we can see for CRP-π. This comes at the cost of limited sharing (α). The bounded stretch (ϵ) remains lower than the values for CRP-π, indicating higher quality for the found detours. We can, however, see a severe drop-off in terms of number the of alternatives and $\mathcal{D}_{tot}$ for long-range queries. Depending on the desired setting, blocking segments can offer a viable choice, though. They are competitive in their quality for low-range and mid-range queries. Especially in the combination with a dynamic CH, the method should be explored further.

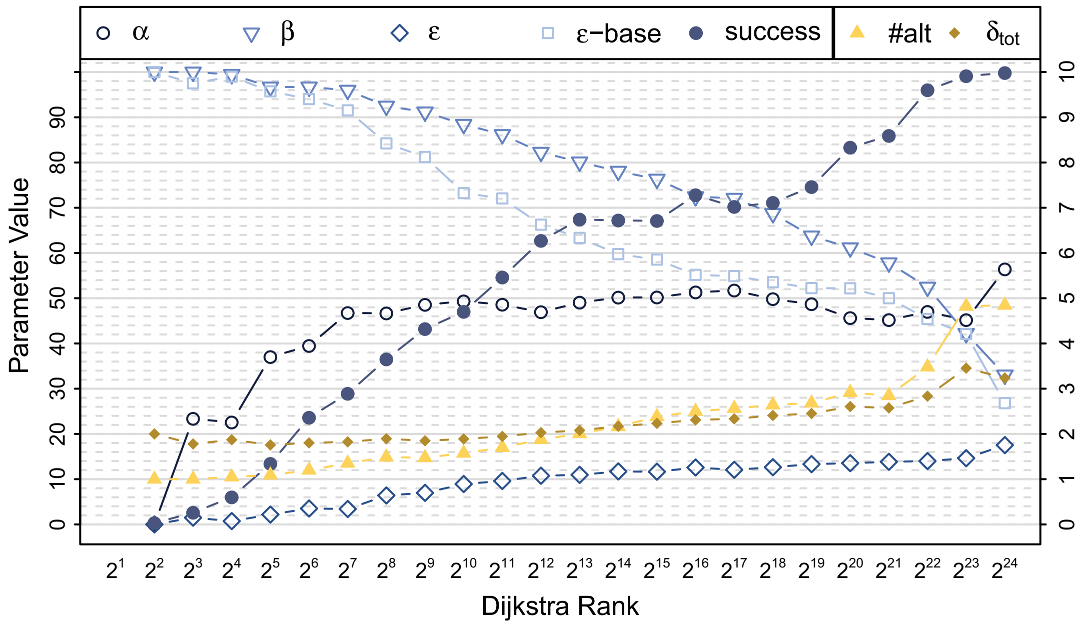
Divide and Conquer. The decrease in computational cost we have seen comes at a minor decrease in quality.

Considering individual factors in our algorithm is difficult as all parameters influence the algorithm in a major way. To test these influences, we performed a series of experiments varying the sequence size to be selected from $\{5, 7, 10, 12, 15, 17, 20\}$, using four cores and performing our recursive approach as well as a-priori division for task generation.

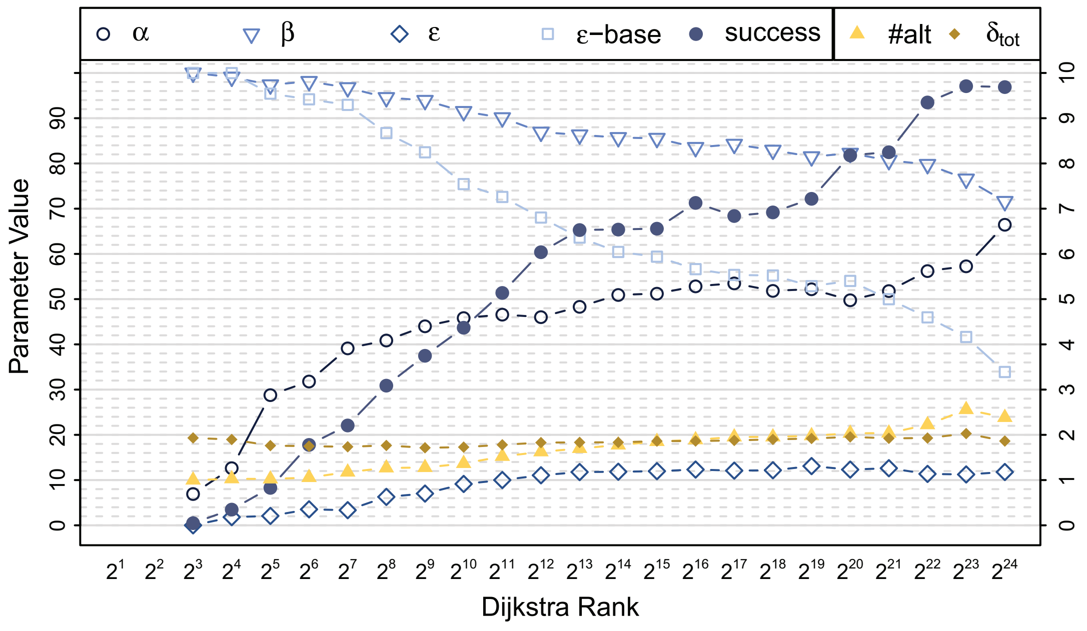
While there is some minor variation, the number of segments we can add to the alternative graph remains close to identical for most Dijkstra ranks. Higher ranks are more sensitive with respect to the number of alternative routes found. The variation comes at the price of a larger number of decision arcs. Their count still remains within the acceptable limit of at most ten, though. The general success rate for the first alternative route remains the same.

Segment Size In general, it seems to be possible to select a wide range of possible segment lengths, depending on the distance between the respective source and target vertices. Most choices yield a very similar result in pure numbers, even though the actual paths differ. In our experiments, the union of multiple alternative graphs, computed with different segment sizes, yields better results than the graphs on their own. Nevertheless, we propose a segment length of $0.1\mathcal{D}(s, t)$ as a general setting that works well in practice.

Increasing the length past the chosen ten percent only shows little benefit.

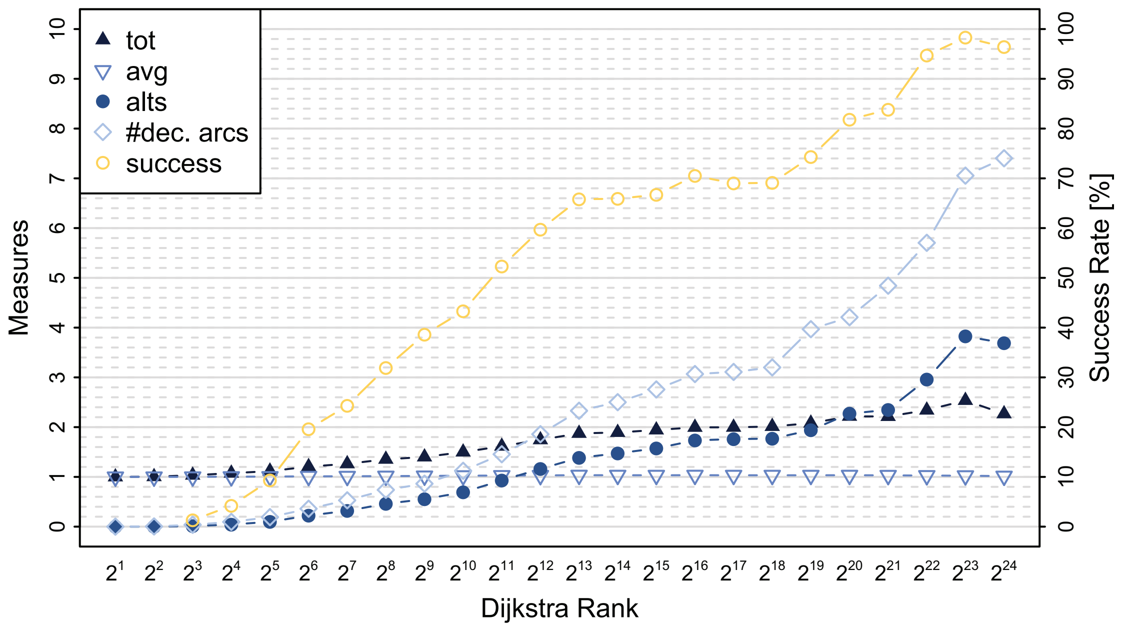


(a) *CRP-π*

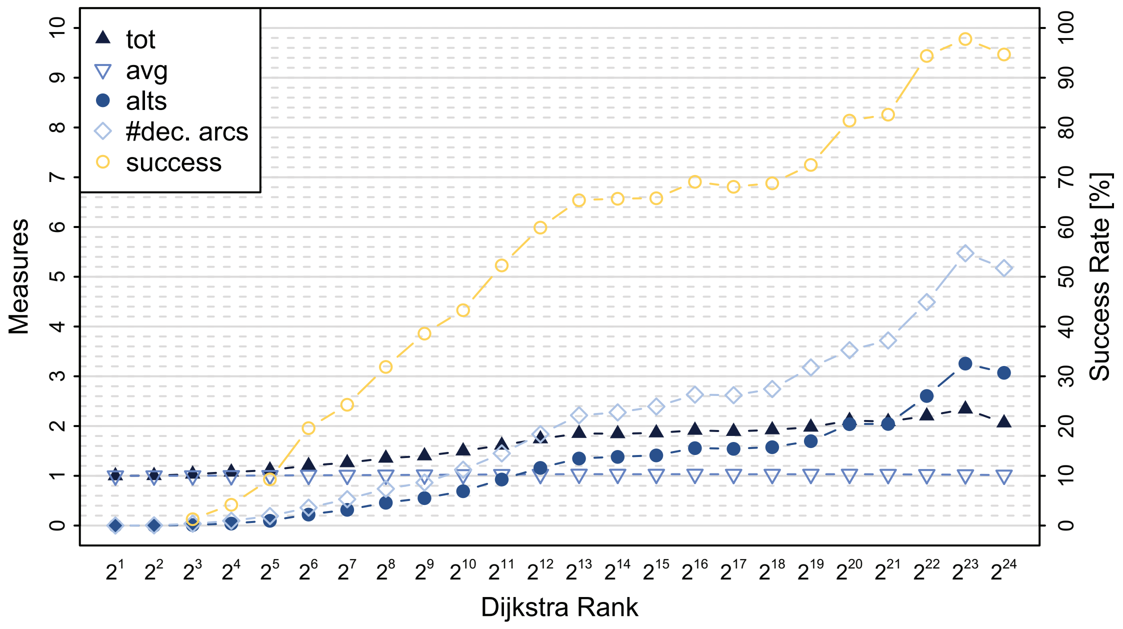


(b) *CRP-∞*

Figure 10.10: *Development of the quality parameters, according to Definition 5.3, for CRP-π and CRP-∞.*

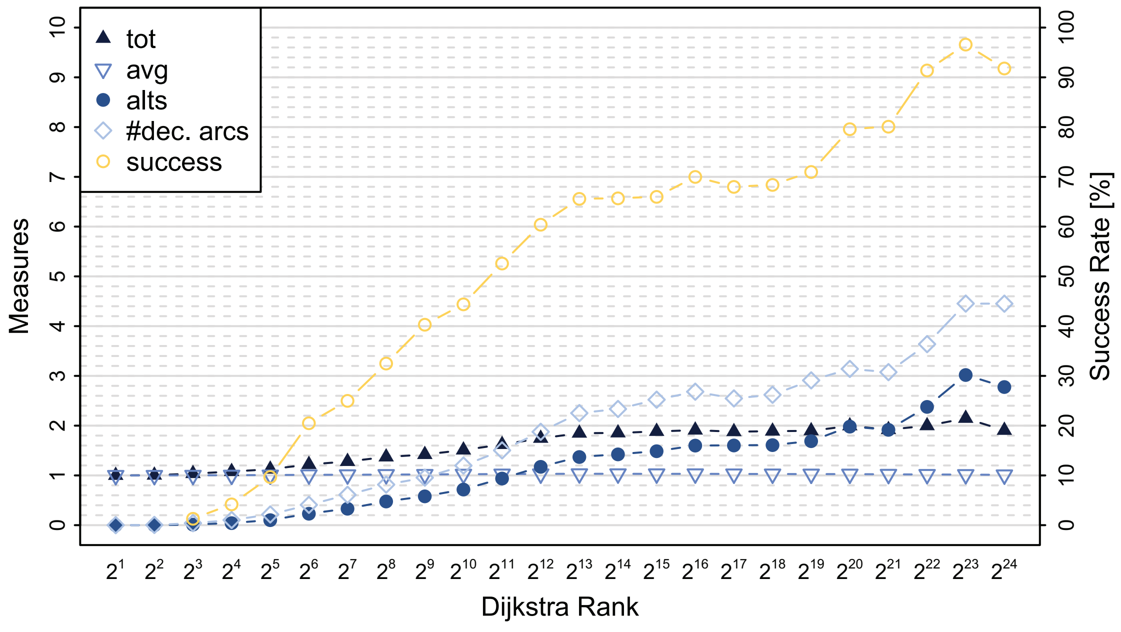


(a) *A-Priori Division*

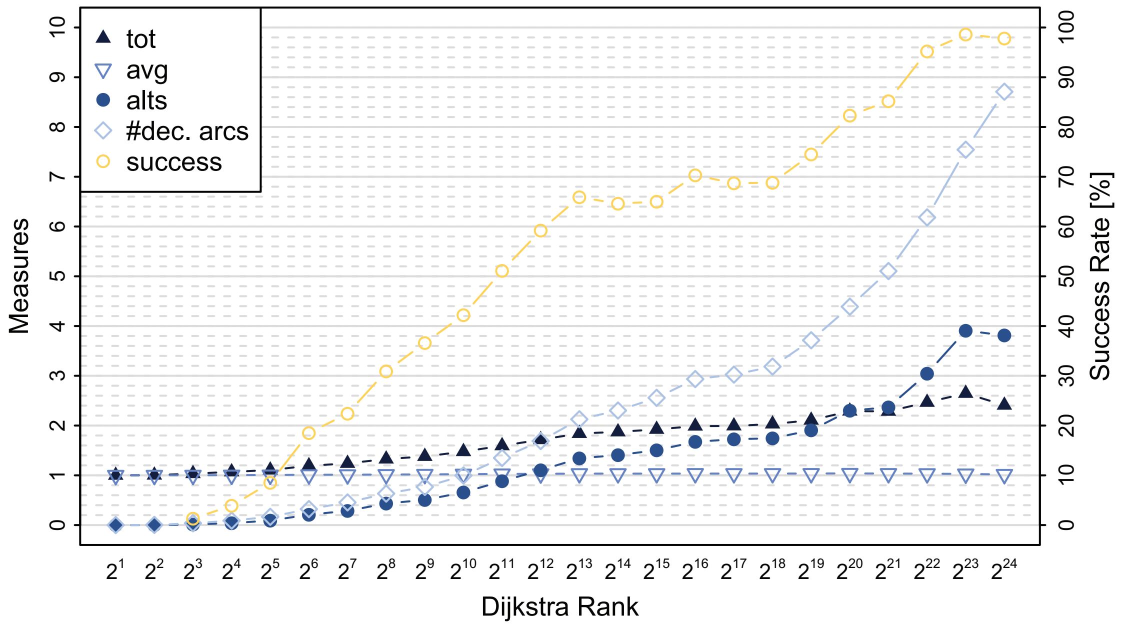


(b) *Divide and Conquer*

Figure 10.11: *Juxtaposition of the quality of results from the divide and conquer as well as the a-priori division concepts. Our algorithm considers a segment size of ten percent of the shortest path in length.*



(a) *Five Percent*



(b) *Fifteen Percent*

Figure 10.12: *Quality comparison of additional segment sizes (A-Priori Division).*

10.3 Conclusion

Excluding segments from the search provides a middle ground between the two major paradigms considered in the computation of alternative routes, the via-vertex and the penalty method. Finding potential points of application is straightforward, opposed to the difficult search for candidates in the via-vertex paradigm. The penalization is less complex than the one required for the penalty method (CRP-π). The restriction to different parts of the shortest path allows for the parallel discovery of multiple candidates, potentially even in a distributed setting. These benefits come at the price of less guarantees with regard to the quality of the route. The only guarantee that we can possibly give is that the stretch along the found alternative will, at no point, exceed the factor of penalization. This factor depends on the length of the segment, though, and can be potentially very high. Even though this effect seems to occur only in a negligible number of cases and for far lower stretch values, possible additions to completely prevent it are required before such an approach can be used in a commercial system.

Also, while requiring less computational overhead than CRP-π, it would be desirable to operate sequentially by finding a good heuristic to select segment candidates instead of recursively creating them. In even further additions, one could forgo the update entirely and check the possibility of dirty regions which the search has to descend into, as would be the case with a dynamic CH. Since only a small set of them is affected, this should allow for a reasonable query time without the required update overhead. Even though our current implementation does not allow for a computation below the desired bound of a hundred milliseconds for all types of queries, the single-core performance is already very impressive. Using as much as four cores, we reach an average of around a hundred and fifty milliseconds, even for the longest long-range queries. For future research, the discovery of better heuristics could replace the divide and conquer approach. This could allow for an algorithm that provides high quality alternative routes while operating within a hundred milliseconds for all queries.

In total, we have proven the exclusion of segments to be a viable method for alternative route computation. For good performance, heuristics to find reasonable locations have to be explored. The expected problems in quality are nearly non-existent.

Chapter 11

11 On a Hybrid Navigation Scenario

Our final method for alternative routes originates in a hybrid routing scenario that combines mobile navigation with powerful servers for computation. The chapter is based on a fruitful collaboration on [DKLW12]. The implementation, which was done by me for the original publication, has been greatly extended for this chapter. Also, the analysis is a result of my own efforts. The initial idea of a corridor can be attributed to Daniel Delling and Renato Werneck.

The wide distribution of smart phones makes mobile navigation services omnipresent. Internet connectivity is available not only in the form of stationary Wireless Local Area Networks(WLANs) but also over the phone network itself. Most techniques for fast shortest-path computation require powerful hardware to work efficiently. Some of them can be implemented on a mobile phone with limited compute capabilities (compare [GSSV12]), though.

A common approach to offering navigational services on a mobile device is to distribute the workload between a server and the device. In most scenarios, the phone takes care of the rendering process as well as the voice assists. The computation of the shortest path takes place on a powerful compute server that only transfers the result to the phone. By splitting the computation in such a way, the mobile device only needs to be able to render and display the route but not to compute it itself. As a result, the navigation algorithm can be far more complex.

By focussing only on the displaying part, the driver may experience a potential loss of service in areas with bad or even no connectivity at all. Consider a situation in which a driver comes upon a complicated intersection offering a number of possible choices on how to continue. If the driver misses the desired turn, the navigation device has to offer a new path.

In case of a loss of connectivity, a potential driver would have no choice but to turn around. If a u-turn should not be possible, he might even have to guess a way back to the proposed shortest path. In the presence of multiple viable paths from a source to a target, the option to turn around might be unnecessarily bad in comparison to an updated path. When we take reaction times into account, it becomes even more important that an updated route can be presented quickly. A quick response can end up saving the driver from a very long detour. As such, not only the complete loss of connectivity might influence the quality of service but also a slow mobile Internet connection could suffice for a negative experience.

We present a method –initially published in [DKLW12] – that provides means of handling such situations quickly and without the requirement of communicating with a server. We approach this scenario by not only transferring a shortest path to the mobile device but rather a meaningful Directed Acyclic Graph directed at the target. Within this DAG, which we call a corridor, the mobile device is able to correct some driver-side errors on its own. This capability requires close to no additional compute capabilities on the device.

This chapter is separated into two parts. First, we discuss general ideas for corridors, independent of the way of computation. After this discussion, we focus on algorithms to efficiently compute corridors using Contraction Hierarchies.

11.1 A General View on Corridors

We refer to a corridor in relation to a source vertex s and target vertex t in a graph $G(V, A)$. Within this graph, a corridor is a DAG that is directed at the target in a way that for every vertex v there exists a single next vertex to go to that is also within the corridor. Following these vertices results in a shortest path from v to t. Formally, we provide Definition 11.1.

Definition 11.1 (Corridor)**.** *Given a graph $G(V, A)$, a source s, and target t: a corridor to s and t is a set of vertices Γ with $s, t \in \Gamma$ that respects the shortest-path distances in G, i.e. $\mathcal{D}_\Gamma(v,t) = \mathcal{D}_G(v,t)\,\forall v \in \Gamma$. Furthermore, every vertex v in Γ is assigned a parent (p) vertex, such that $\mathcal{D}(v,t) = \mu(v,p) + \mathcal{D}(p,t)$.*

Some simple examples of valid corridors are the shortest path from a source s to a target t, or even a full shortest-path tree, directed at t. The main goal of this chapter is to explore different options in between these two extremes.

The benefits of the corridor approach to navigation are twofold. First, the corridor provides robustness in a scenario where the loss of connectivity poses a threat. Secondly, a corridor provides immediate updates to the driving direction in case of a missed turn/exit, due to the stored parent vertices. Using these vertices, a shortest path computation translates to a simple traversal.

Even additional information, like the estimated arrival time, can be stored with every vertex, if it should be required. This simplicity in the shortest-path computation predestines corridors for fast computation of new routes on devices with low computational power. This compute capability is limited tot he corridor, though. As a direct result of this limitation, we ask the question of how robust a corridor is against leaving it.

The property of robustness is difficult to grasp, as it is near impossible to describe a *typical* traversal of a corridor. Of course, a full shortest-path tree is robust against all possible kinds or errors. At the same time, the shortest path alone cannot be used to correct for even a single deviation. It is unclear however, what the best choices in between might look like. For example, we expect far more errors in highly complicated highway ramp configurations or complex intersections than on general highway segments in which the directions request only to continue on the same road. The subjective impression of complexity is hard to translate into an accurate measurement. We provide Definition 11.2 for the evaluation in the context of this thesis.

Definition 11.2 (Robustness). *We define the robustness of a corridor Γ with respect to a type of driver. Any driver can be seen as the model of some form of random behavior, not following the proposed route with some given probability. The robustness of a corridor is defined in form of a success rate indicating the percentage of drivers that reach their destination without leaving the corridor. We refer to the robustness of a corridor as $\Re(\Gamma)$.*

In the following Sections, we explore reasonable ways to compute possible corridors.

11.2 Search Space Corridors

The most obvious method for computing a corridor is to utilize the information already provided by a known speed-up technique. Pruning methods, for the most time, are not able to reduce the information to the shortest path alone. The – in terms of shortest-path computation – superfluous information might actually present a viable selection for corridors.

11.2.1 Goal Direction

To lessen both the computational and memory overhead involved with corridors, we can prune parts of the shortest-path tree. The most obvious choice would be to not consider the full shortest-path tree but rather restrict the search to all vertices that can reach the target within $(1+\epsilon)\cdot\mathcal{D}(s,t)$. In the case of driving directions, this approach considers a lot of unreasonable vertices for potential deviations; detours that originate from failure to follow the given directions can be expected to stay within some vicinity of the shortest path. Considering both the source and the target, we can achieve better pruning by employing goal-direction techniques like ALT or ArcFlags. Both techniques offer reasonably low query times to be considered, we argue that they are unreasonable nonetheless.

ALT-corridor. When we consider a complete run of Dijkstra's algorithm, the shortest-path tree – or even a pruned one – provides information on vertices far beyond the desired destination of a drive. The circular way in which the algorithm operates (compare Figure 3.1) results in a large amount of information on paths leading away from the desired target. A way to circumvent this could be to perform a bi-directional search and only to consider vertices v with $\mathcal{D}(s, v) + \mathcal{D}(v, t) \leq (1 + \epsilon) \cdot \mathcal{D}(s, t)$, which we can, for example, achieve using the ALT algorithm.

We can even execute some control over the technique and the number of vertices included by choosing different values of ϵ . Results in the context of the penalty method ([PS13]) indicate a large associated overhead for this technique, though. The large amount of required vertices brings us to the decision to not investigate this method any further.

ArcFlags Corridor. An alternative method for a the calculation of a corridor would be to consider ArcFlags as a pruning technique[1]. Searching from the target backwards along the flagged arcs to the source, we can keep all vertices v in the corridor that do not violate $\mathcal{D}(v, t) \leq (1 + \epsilon) \cdot \mathcal{D}(s, t)$. In relation to the ALT-based version, we expect a less dense corridor than a ALT-based corridor at the cost of far lower robustness. The main reason for this loss in robustness is founded in some artefacts that are typical for the ArcFlags algorithm, fans and local queries. Figure 11.1 presents two exemplary corridors and justifies our decision to also discard ArcFlags as a possible method.

These effects (compare Figure 11.1b) can be reduced by introducing multiple levels of ArcFlags and using a bi-directional query. Doing so, we are still left with long sub-paths that offer no robustness. It is unclear, how the robustness along such a path might be increased in a natural way. A similar approach to the one we discuss in Section 11.8 could be possible, though.

11.2.2 Hierarchical Techniques

The other most frequently used methods are CH and CRP. We also considered the search spaces of both techniques. In terms of CH, the reasonable search space to consider equals the graph computed by our own HiDAR algorithm. Similar to the setting of the long paths already seen in Figure 11.1b, hierarchical techniques contain many long arcs. These long arcs lead to the same argument as we have already seen for goal directed techniques; we give an example in Figure 11.2.

As a result, the search spaces of hierarchical techniques alone cannot be considered a viable choice for a reasonable corridor. They could offer a viable starting position, though, forming a reasonable base for one of the two techniques that we describe in the next sections.

[1]We thank Dennis Schieferdecker for providing us with the ArcFlags for our graph.

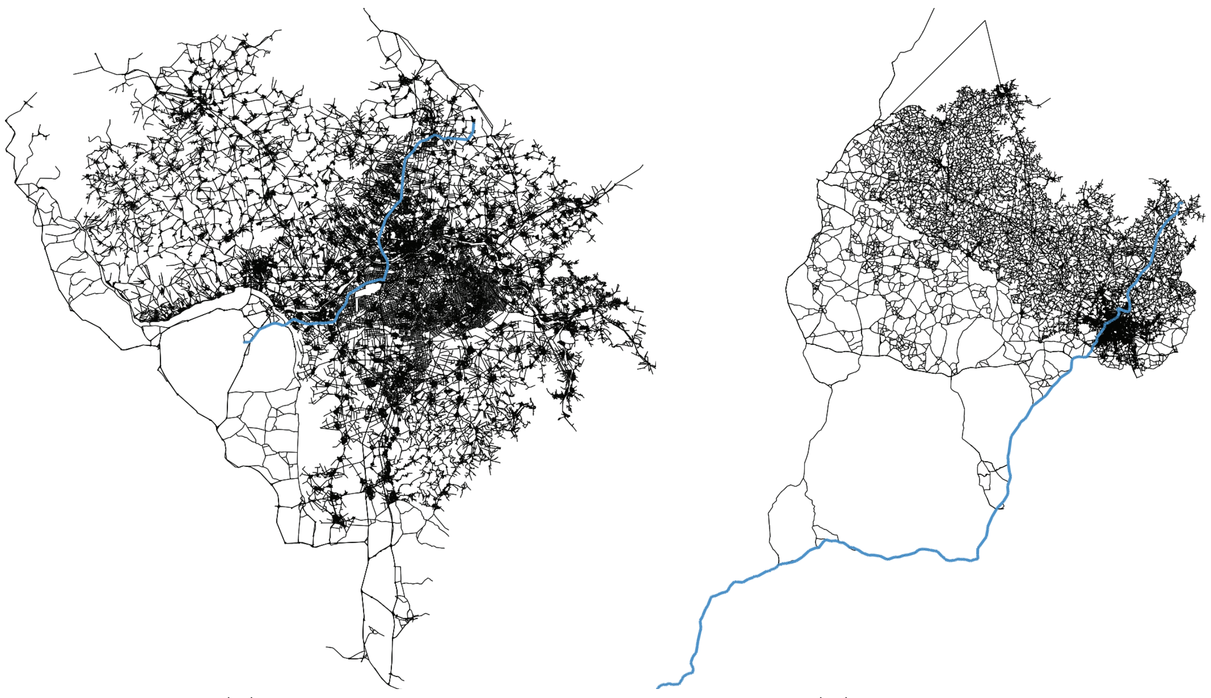

(a) *Local Area* (b) *Long Path*

Figure 11.1: *Graphical representation of two ArcFlags corridors. In a local area (cell), all flags are set to* `true`*. This prevents any pruning close to the target. At the end of a long path, ArcFlags suffer from a fan-like effect. Both problems can be alleviated (multi-level ArcFlags / bi-directional implementation – see Section 3.1). The technique does not provide promising solutions for all problematic parts, though.*

11.3 Turn Corridors

Our first model considers the scenario which we mentioned a few times already: a driver might exhibit erroneous behavior. These errors could manifests themselves in wrong turns along the given directions. We directly translate these wrong turns into our corridor definition and, accordingly, name this model a *turn corridor* (compare Figure 11.4a).

A turn corridor can be inductively described: initially we start with the shortest path which forms a turn corridor of *degree* zero. Given a corridor of degree d, we can calculate the next degree corridor by augmenting the degree d corridor with all vertices that are directly connected to it. Finally, we compute the shortest-path hull of the vertices by including the shortest paths from all of these vertices to the target. The formal description of a turn corridor is given in Definition 11.3.

Definition 11.3 (Turn Corridor)**.** *Given a source vertex s and a target t in a graph $G(V, A)$, a turn corridor of degree zero is given by $\Gamma_0 := \Pi_{s,t}$. Inductively, we define*

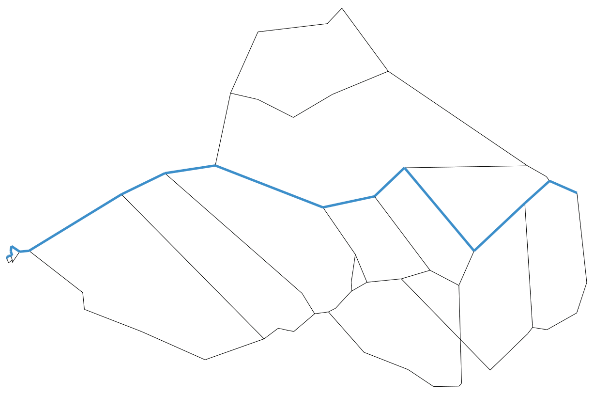


Figure 11.2: *Exemplary representation of a corridor based on HiDAR. In the same way as we experience when using CRP, shortcuts (depicted here without their respective underlying road geometry) result in long (sub)-paths without means for correcting erroneous behavior. The query depicted illustrates a shortest path of Dijkstra rank of two to the power of nineteen.*

a turn corridor of degree $k+1$ to a corridor of degree k via the deviation vertices $\Delta\left(\Gamma_k\right) := \left\{v \in V : \exists a = (u, v) \in A \wedge u \in \Gamma_k\right\}$ *as:* $\Gamma_{k+1} = \Gamma_k \cup \left\{u \in \Pi_{v,t} : v \in \Delta\left(\Gamma_k\right)\right\}$.

This inductive definition provides a natural way of managing the required robustness in the trade-off with relation to corridor size. The calculation is simple as we do not require a multi-source shortest-path computation. Instead, we can simply traverse the corridor and scan arcs to find the deviation vertices. Next, the inductive definition also provides means to correct errors on the paths we have found in the previous iteration, treating every vertex within a corridor in the same way.

11.3.1 Theoretical Discussions

Turning to highway dimension (h), we can show this version of corridors to be limited (non-trivially) in their size. As we explained in Section 4.1.5, highway dimension postulates a small (h) shortest-path cover for all shortest paths of a given length in a local area. Under a few reasonable assumptions, we can show that the size of a turn corridor is limited by $\mathcal{O}\left(h \cdot n^{0.75}\right)$. While, of course, a size this large is still impractical for large graphs, the limit only specifies a (non-trivial) upper bound. Our experiments show that the actual size of a corridor does not increase as much as this limit suggests.

Theorem 11.1 (Turn Corridor Size)**.** *Let $G\left(V, A\right)$ be an undirected graph with constant maximum degree $\delta\left(\cdot\right)_{\max}$, highway dimension h and assigned metric μ, $\mu\left(a\right) \in \Theta\left(1\right) \forall a \in A$. Moreover, let $\mathcal{D}\left(s, t\right) \in \mathcal{O}\left(\sqrt{n}\right) \forall s, t \in V$. Then, $\left|\Gamma_1\left(s, t\right)\right| \leq h \cdot n^{0.75}$ holds.*

Assuming a constant upper bound for the metric is a strong assumption. It can be justified, though, as we can explicitly split long arcs, if necessary.

Proof. Let $c := \Theta\left(\sqrt[4]{n}\right)$ and by extension $\mathcal{D}(s,t) \in \mathcal{O}\left(c^2\right)$. We consider two cases.

Case 1 ($\mathcal{D}(s,t) \in \omega(c)$): We partition the shortest path $\Pi_{s,t}$ into $\mathcal{O}(c)$ parts, each of which is at most c in length. In each of the resulting parts P^i, we can select a vertex v_i. Consider the balls of size $2c$ and $4c$ around v_i : Due to $\mu(a) \in \Theta(1)$, any deviation vertex is contained within the ball of size $2c$ around v_i . The subpath optimality, the principle that guarantees the correctness of Dijkstras' algorithm, states that the prefix of a shortest path is a shortest path itself. In our case, we can postulate an optimal path of length $2c$ at the beginning of each shortest path to the target; due to highway dimension, this path is covered by one of h vertices within the ball of size $4c$. Even if all of the $\mathcal{O}(c)$ deviation vertices were assigned a distinct shortest path to one of these h vertices, the total number of vertices would still be bounded by $\mathcal{O}\left(c^2\right)$. Since no shortest path can be longer than the equivalent of turning around and following the original shortest path, the length of the shortest path from the covering vertices to the target is also bounded by $\mathcal{O}\left(c^2\right)$. Given that our partition consists of $\mathcal{O}(c)$ parts, the total amount of vertices in a turn corridor of degree one is bounded by $\mathcal{O}\left(h \cdot c^3\right)$. See Figure 11.3 for an illustration.

Case 2 ($\mathcal{D}(s,t) \in \mathcal{O}(c)$): The limited maximum degree gives $|\Delta(\Pi_{s,t})| \in \mathcal{O}(c)$. As a direct result, due to $\mathcal{D}(s,t) \in \mathcal{O}(c)$ and G being undirected, we get a bound of $\mathcal{O}\left(c^2\right)$ for the size of Γ_1 .

□

Especially the assumption of distinct paths overestimates the size of the corridor. We expect many of the paths to the target to actually share vertices. In addition, we currently do not know any way of integrating the interaction between neighbouring balls.

11.4 Detour Corridors

Another possibility to compute corridors is to consider the notion of a realization period. This realization period describes the reaction time that a potential driver might require after making an error. Focused on the situation, a driver might take a certain time δ_t to realize new instructions from the navigation device. To compensate for this, it seems reasonable to consider every location that is reachable within the time δ_t when starting somewhere along the shortest path. In accordance to this limit on the detour δ_t, we refer to this setting as a *detour*-corridor.

These locations, or vertices, can be computed in a straightforward manner with Dijkstra's algorithm (Algorithm 2.2). Instead of staring an instance at a source, we simply start at all locations along the shortest path at once. We can achieve this behavior by initially adding all vertices of the shortest path into the priority queue

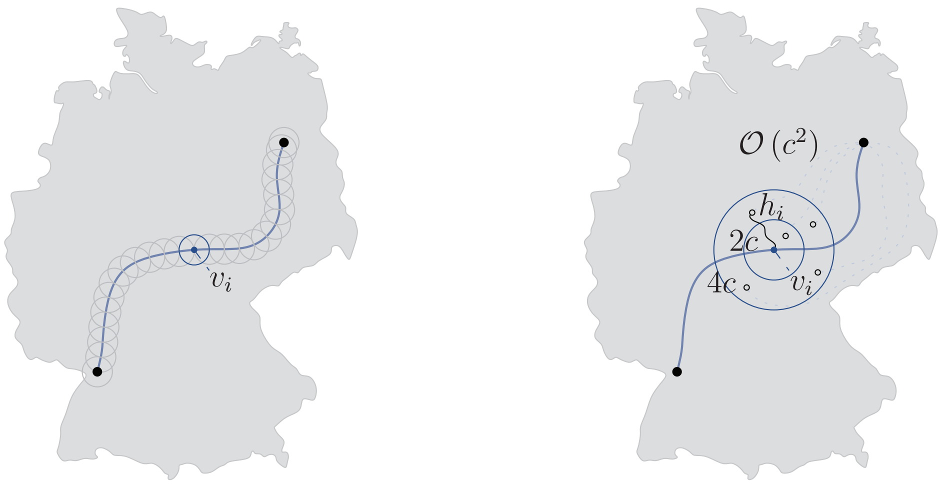


Figure 11.3: *Illustration of the proof to Theorem 11.1.*

with a distance value of zero. During the algorithm, we limit the relaxation to only consider vertices that are within the bounds given by δ_t.

If the value for δ_t is reasonably small, the number of vertices within this tube around the shortest path can be expected to be small as well. The method already requires more effort than the turn corridor, though.

These vertices, settled in the expansion algorithm, are the initial part of the corridor. To form a valid DAG directed at the target t, we have to compute the shortest-path hull of the vertices and add all found paths to the corridor as well.

Recursive Extension. The corridor can be extended in a recursive manner. The process does not come as naturally as for the turn corridors, though. A direct recursion does not seem appropriate as it would, for the most part, double the tube around the shortest path in its size. Instead, we propose to focus on the vertices that currently offer no deviation possibilities. We limit the recursion to vertices that part of paths that leave the tube of size δ_t around the shortest path. These vertices are discovered after the extension phase and are identified by not having any neighbours within the corridor besides their respective predecessors/successors. To reduce the additional cost, we also propose to reduce the allowable detour with every recursive extension. For our experiments, we use $\frac{delta_t}{2^r}$, for the recursion level r.

Schematically, these vertices can be seen in Figure 11.4b in form of the two additional paths.

Detour Corridor Properties. As with turn corridors, we can limit the size of detour corridors. As long as we limit the detour time to $\delta_t \in \Theta(1)$, the arguments given in the proof of Theorem 11.1 can be applied to detour corridor size as well. The only

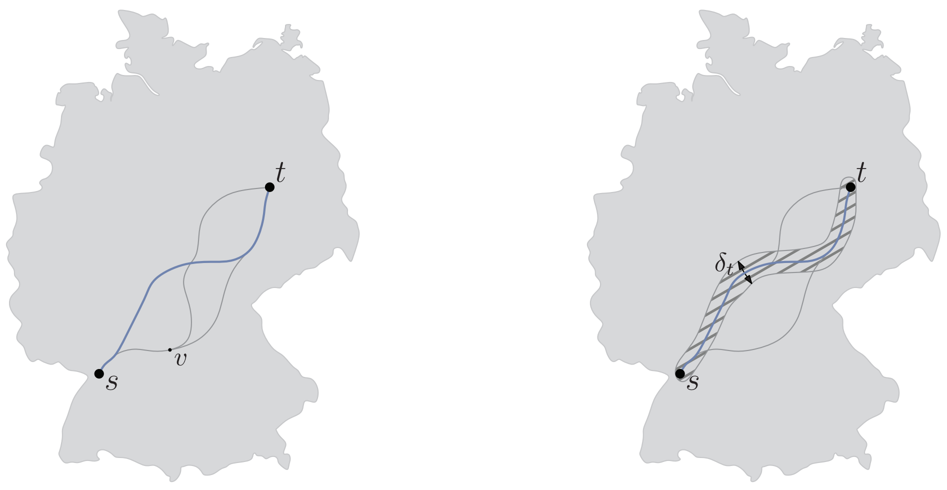


Figure 11.4: *Schematics of detour and turn corridors.*

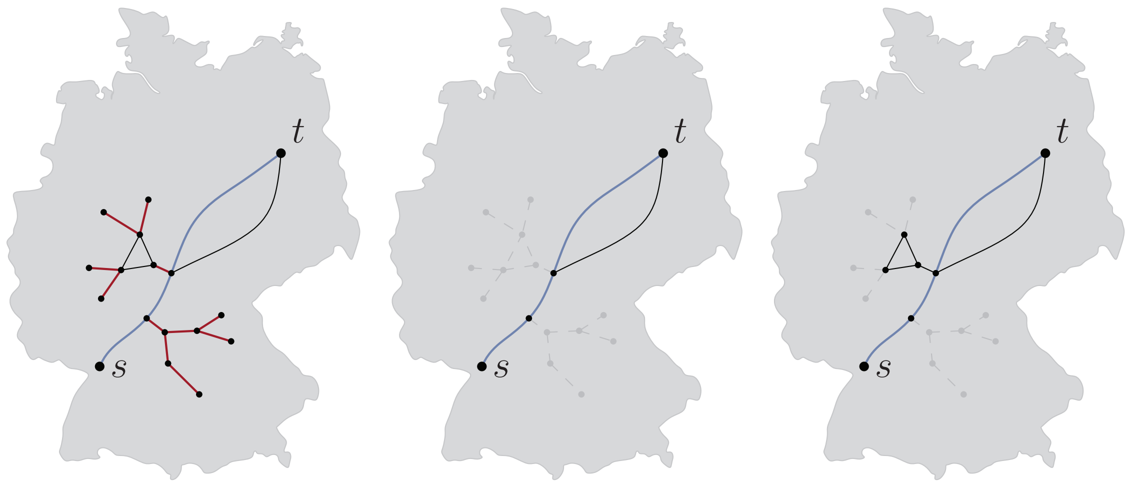
difference between the two is that the number of deviation vertices to a vertex is not limited by the maximum degree but rather by $\pi\delta_t^2$ (which is within $\mathcal{O}(1)$, given our prior assumptions). We can even lower this bound to $\frac{\pi\delta_t^2+\sqrt{n}2\delta_t}{\sqrt{n}} \approx 2\delta_t$ on average, assuming a density of one of the vertices over the plane.

11.4.1 Attached Components

Both of the discussed methods have the tendency to generate attached components. No matter how exactly the component looks, these components add little in terms of alternative routes, as they are entered and exited via the same arc. Therefore, testing for both bounded stretch and local optimality cannot yield a positive result. We propose two methods to filter out these kinds of components.

Bridge-Filtering. Bridge-filtering is the stronger filtering method of the two; a bridge in graph-theory is an arc that, if removed, splits the graph into two components. We employ the standard DFS-based algorithm to detect bridges in an (undirected) representation of the corridor. To avoid side effects regarding the source and the target, we add an additional arc between these two vertices to ensure that they are located on a circle. The bridge detection algorithm finds all arcs that are not part of a circle. In a second step, we scan the graph and remove all vertices and arcs that can only be reached using such a bridge. We illustrate this method in Figure 11.5b.

Topological-Tree-Filtering. The bridge-filtering method removes all attached components. If we want to provide correct route requests for all possible turns up to a certain limit, this removal of components can be seen as an extreme choice. A less



(a) *Corridor* **(b)** *Bridge Filtering* **(c)** *Top. Tree Filtering*

Figure 11.5: *Visualization of our methods for attached component filtering. The gray arcs and vertices visualize the part that is filtered by the two methods. The red arcs in Figure 11.5a indicate the location of bridges.*

aggressive technique would be to use a method based on a topological sort. Our topological-tree-filtering only removes attached trees, as illustrated in Figure 11.5c, starting at their leaves. Again, we prevent possible side-effects on s and t by adding an additional arc between them in an undirected interpretation of the corridor. Afterwards, we perform a topological sort, to remove all vertices that are connected to the corridor by exactly a single arc.

Drawbacks of Filtering. Filtering reduces the amount of unnecessary information at the cost of corridor robustness[2]. When considering turn cost, it is unlikely that we would have to ask the driver to perform a u-turn in a local area. By removing these attached components, we lose some knowledge as well. Especially the bridge-filtering method removes information on all attached circles. It would be possible, however, to implicitly remember u-turns using the topological-tree-filtering on a turn corridor. Comparing the maximal number of wrong turns allowed and the number of wrong turns taken, we can detect whether a vertex would normally be in the corridor. If the number of wrong turns taken is lower than the number of wrong turns during the calculation, we know that the vertex is part of an attached tree and can safely request a u-turn.

[2]We define our notion of robustness in the next Section.

11.5 Robustness

The decision whether a corridor is robust is hard to make as it is near impossible to describe what we would expect in terms of erroneous behavior from a driver. Next to the complexity of the respective intersections, the various drivers using a navigation device cannot be pressed into a simple model. We still chose to simulate a series of different potential drivers to evaluate the corridor methods for their robustness.

11.5.1 The Random Drive Concept

Our concept to test a corridor is based on the general notion of random walks [Pea05]. In our interpretation of the random walk, we take a look at random drives in a plane, limiting the drive to a reasonable setting. For example we assume that a driver within a corridor does not to suddenly turn around in the middle of a road and heads into the opposite direction. As a direct result of this assumption, we can limit the randomness to events that occur at intersections.

After an initial error, this assumption might be invalid. If the driver can perform a u-turn on the current arc, he can return to the shortest path on its own. If the u-turn is not possible, our corridor provides the correct next vertex to travel to.

The moment the GPS detects a deviation from the initial route – ideally this should happen prior to a driver realizing its mistake, even though it is the other way around for the most time – our algorithm can instantly supply the driver with an updated route information and accompanying driving directions. This still holds true, if the driver turns around in the middle of the road. Due to the driver only being able to reach two locations for which the correct driving directions are already known, we can argue that the driver will either have settled for the new and updated route information or return to his or her original trajectory. In both cases, further random u-turns are not to be expected.

For our experiments, we tested a range of drivers, each of which is described in one of the following paragraphs. At this point, we present a selection of the tested drivers. Further studies can be found in Appendix E.

The Adjusted RD Driver. We describe an *adjusted deviation driver*, or aRD, using two probabilities p and $p' > p$; p defines the probability of an initial wrong turn. The additional parameter p' describes the probability of taking another wrong turn after an initial mistake. This heightened probability of a wrong turn emulates a potential panic-driven reaction of a driver after discovering the initial mistake. After following the (updated) driving directions again, the driver switches back to the initial probability p. In our experiments, we use $p' = 2p$.

Complexity Driver. We do not expect a uniform probability of taking a wrong turn. For example, as long as we continue on a road, we would not expect a driver to randomly make a left turn. To account for this, we study the complexity-driven random deviation driver (cRD). It is described in a single parameter p, which gives the probability of a wrong turn. We scale this probability by $\frac{|\angle(u,v,w)-180|\cdot k}{90\cdot 2}p$ with k specifying the number of potential wrong directions to travel in (except for u-turns).

Angular Resistance Driver. Bot previous drivers use wrong turns as their method of deviation. While every deviation is initiated by a wrong turn, only considering turns would result in a bias towards turn corridors. Similar to the model of corridors that allow for a certain time of error, we model a driver based on an error period. The angular resistance driver (AR) is parameterized by an error probability p and a time budget t. Extending the cRD, we analyze the path for the angle present in the current turn descriptions. An error occurs at v with probability $\frac{|\angle(u,v,w)-180|\cdot k}{90\cdot 2}p$ (compare cRD). After such an initial error, we switch to a deviation mode for a travel time of t. When the driver's state is in this deviation mode, rather than following the driving directions specified by the corridor, the driver automatically follows the path of least resistance. To be exact, we calculate the angle between the previously taken road segment and the potential candidates. Among all candidates, we choose the one which is closest to continuing straight; in other words, for a road segment u, v we choose w to minimize $\angle(u, v, w) - 180$.

None of the presented drivers can be described as a very realistic model. Nevertheless, the tested drivers give a general idea of the entire concept and show the viability of corridors. A random wrong turn along a path seems rather unlikely. For a better model, we could even try to improve the corridor calculation to consider difficult intersections instead of all intersections. This would allow for smaller corridors that are directed at correcting only a specific kind of mistake. We would require some information on the *difficulty* of different intersections, though. An angular analysis as done for the AR or cRD could provide a starting point.

Corridor Robustness We evaluate a corridor by simulating a series of random drives for a given source and target pair. If the driver reaches an intersection that is not within the corridor, we count the try as a failure, otherwise as a success. The only exception from this rule is the budgeted driving period of the angular resistance driver; here we check the inclusion property at the end of the driving period. The success rate of a corridor gives an intuition on how well the corridor can handle different rates of errors.

11.5.2 Evaluation

Before we consider the time necessary to compute one of our corridor models, we focus on their general properties. We evaluate them for both their size and the achievable success rates in relation to the previously introduced random drivers. Following the extensive evaluation, we discuss an efficient implementation.

General Properties. In a hybrid scenario, we require a maximum quality of service at minimal expense. The expense is divided into two factors: the number of vertices in the corridor and the time we have to invest in the computation. The latter is subject to factors considered in the upcoming sections. At this point, we focus on the former. In the appendix, we present Figure E.5 and Figure E.6, to visualize the growth of the respective corridor models. In general, the detour corridor offers a lot of information very close to the shortest path and no robustness on longer alternative segments. The turn model, on the other hand, offers the natural recursive extension and in extension robustness along most of the contained paths.

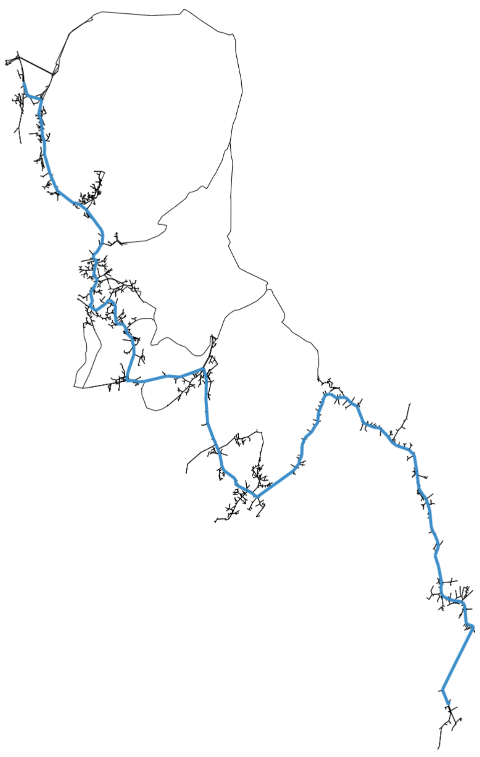
Judging purely by the visual representation (compare Figure 11.6), the recursive variant of the detour corridor (Figure 11.6b) seems the most robust. In our experiments, we see the merits of both the turn variant as well as the recursive detour variant. In general, a high degree of robustness comes at the cost of a high number of vertices within the corridor – potentially too high for practical purposes.

We visualize the size of different corridor methods in Figure 11.7. Already relatively short random detours result in rather large corridors. A detour corridor for a ten second detour ($\delta_t = 100$) is comparable in its size to a turn corridor of degree three. Figure 11.6 indicates the turn corridor to be better suited to our cause, though. Rather long detours – i.e. longer than ten seconds – seem to generate corridors that are too large.

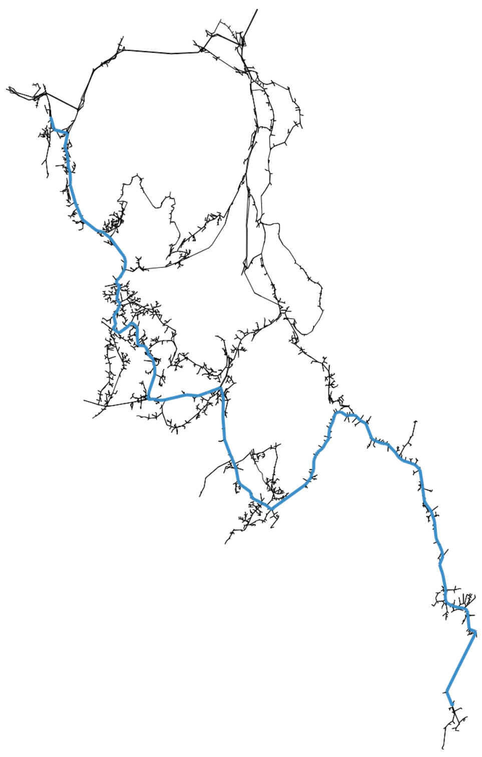
The corridors consist of, in comparison to the theoretical bound, far fewer vertices. Perhaps the most obvious reason behind this behavior is given in the low number of distinct reasonable paths between any given source and target. As a result, we expect a corridor to direct a driver directly back to the initial shortest path in the most cases.

As our model does not support turn penalties, one could offer criticism that the high number of expected u-turns effectively reduces the corridor to the shortest path and the directly connected deviation vertices. To counter this argument, we provide the same information for the arc-expanded version of the graph, a version in which u-turns are penalized by a hundred seconds. We give a juxtaposition of corridors sizes for both versions in Figure 11.8. While the switched setting does increase the corridor growth, the size remains manageable.

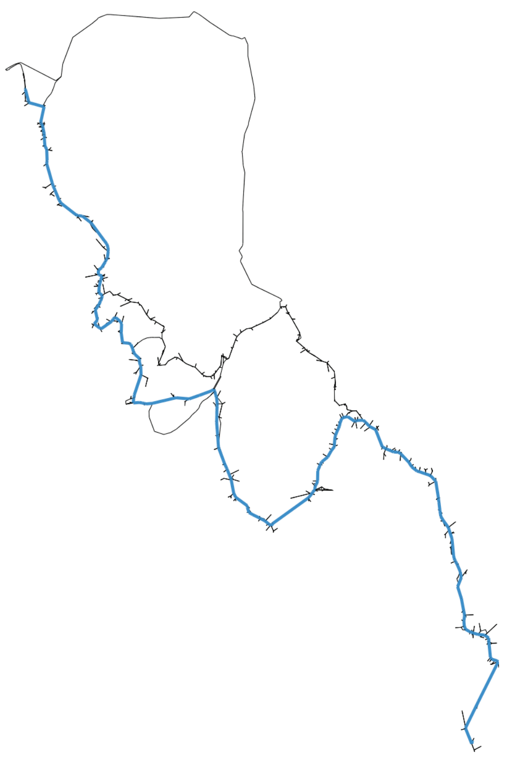
While the general size of the corridor gets far larger in the arc-expanded version of the graph, the relative growth between a corridor of degree i and one of degree $i + 1$ remains at a relative factor of about two. This indicates that our methods do not benefit in an unfair way from using the standard graph model over the arc-expanded



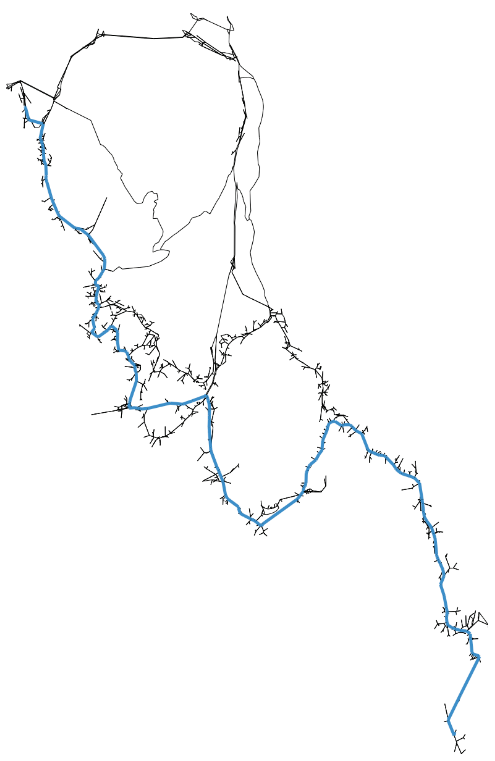
(a) $\delta_t = 100$



(b) $\delta_t = 100$, *recursion depth: 3*



(c) *Two turns*



(d) *Four turns*

Figure 11.6: *Juxtaposition of a detour corridor, a recursive detour corridor, and two different turn corridors on a short range query (Dijkstra rank* 2^{14} .

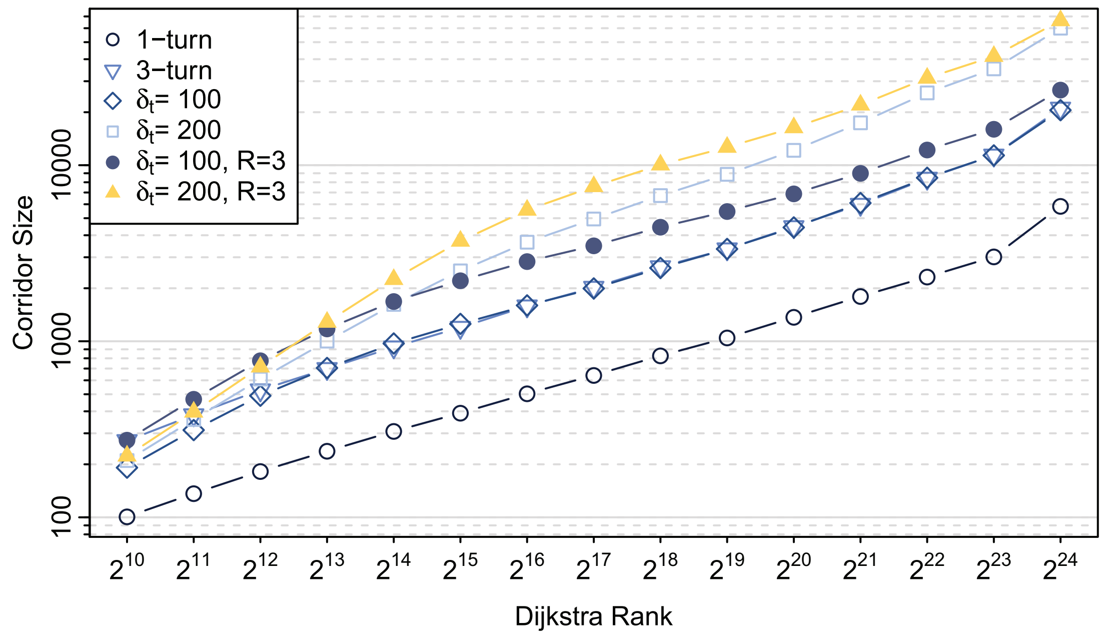


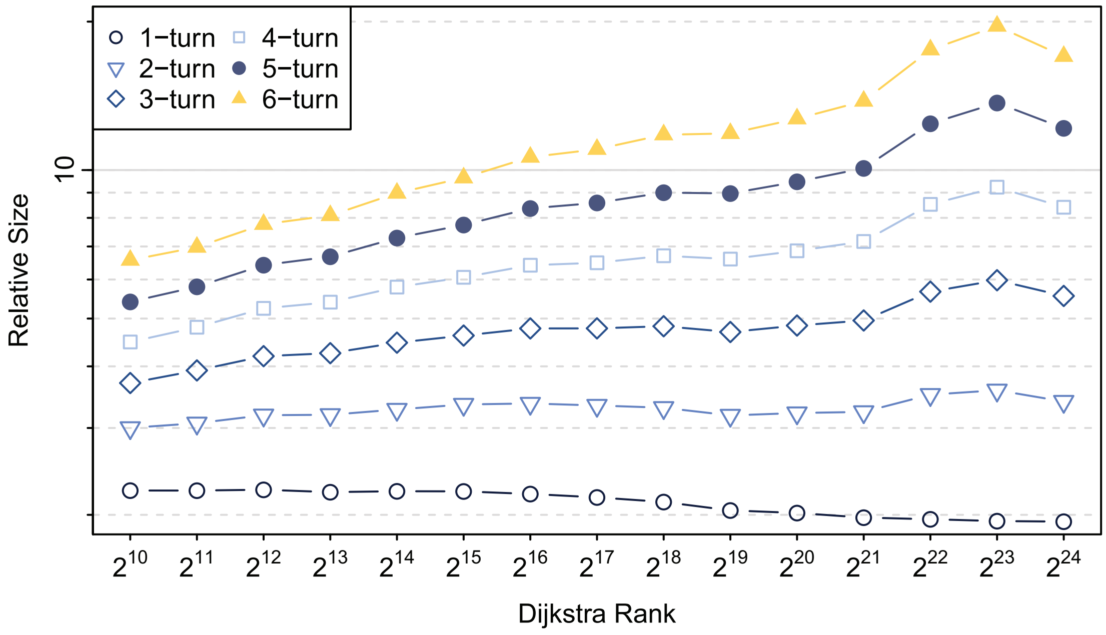
Figure 11.7: *Corridor size in relation to Dijkstra rank for various corridor methods. $\delta_t/10$ specifies the allowed detour in seconds and R the recursion level for the detour corridor.*

one. In addition, the methods that we are introducing in the upcoming sections perform well enough to handle even the arc-expanded graph fast enough for interactive settings, remaining within a hundred milliseconds for a six-turn corridor.

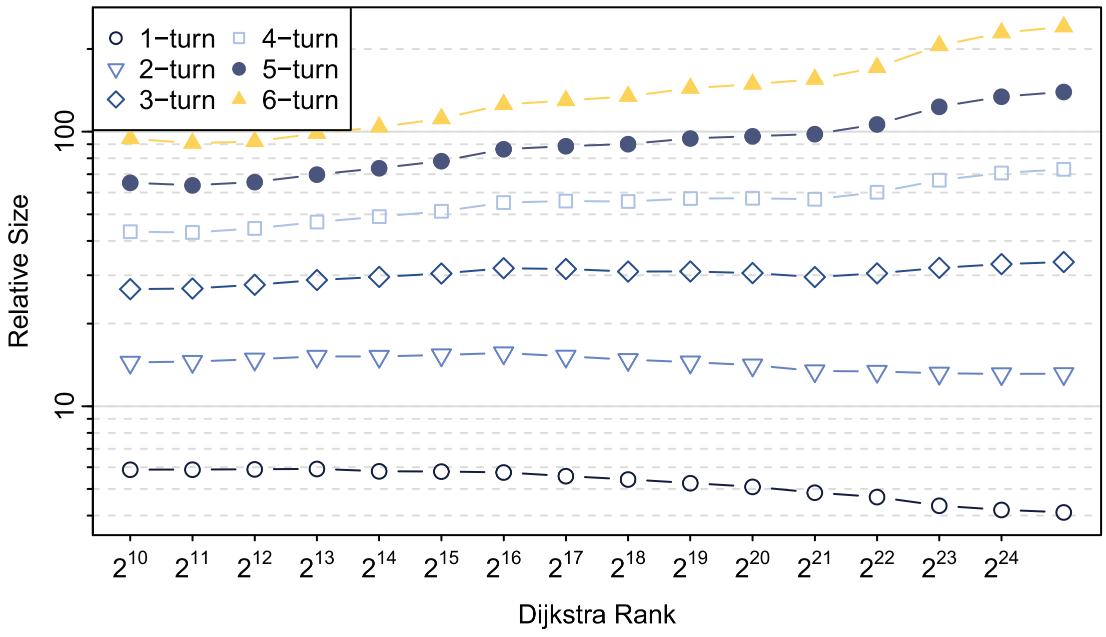
Success Rate. The different driver models favour different types of corridor models. The figures 11.9-11.12 illustrate the different corridor models and the experienced success rates. In every random drive we performed, we limited the time spent on the drive to at most two times the length of the shortest path. Whenever a driver did not leave the corridor within this time-frame, the resulting drive is considered a success.

The detour corridors for ten and twenty seconds of allowed deviation ($\delta_t \in \{100, 200\}$) only show disappointing success rates. A single exception is the cRD driver who is expected to take far less wrong turns than the other models and favours errors along the dense parts of the graph. In these sections, we expect far shorter arcs and as a result a good coverage in the detour model. For lesser error probabilities, the success rates gradually improve, though.

The recursive variant of the detour corridor – both plots show a recursion depth of three – increases the chances of staying within the corridor. All success rates for both the turn-by-turn driver models as well as the AR variants of the drivers stay rather poor, though. We can increase the recursive properties of the approach by not

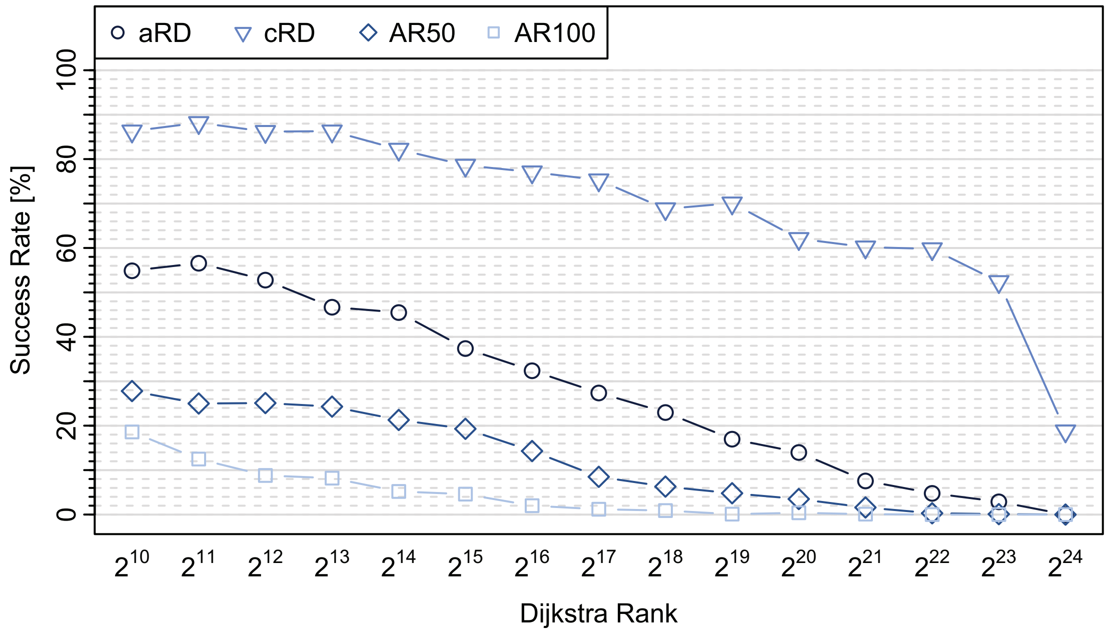


(a) *Standard Graph Model (Graph 1)*

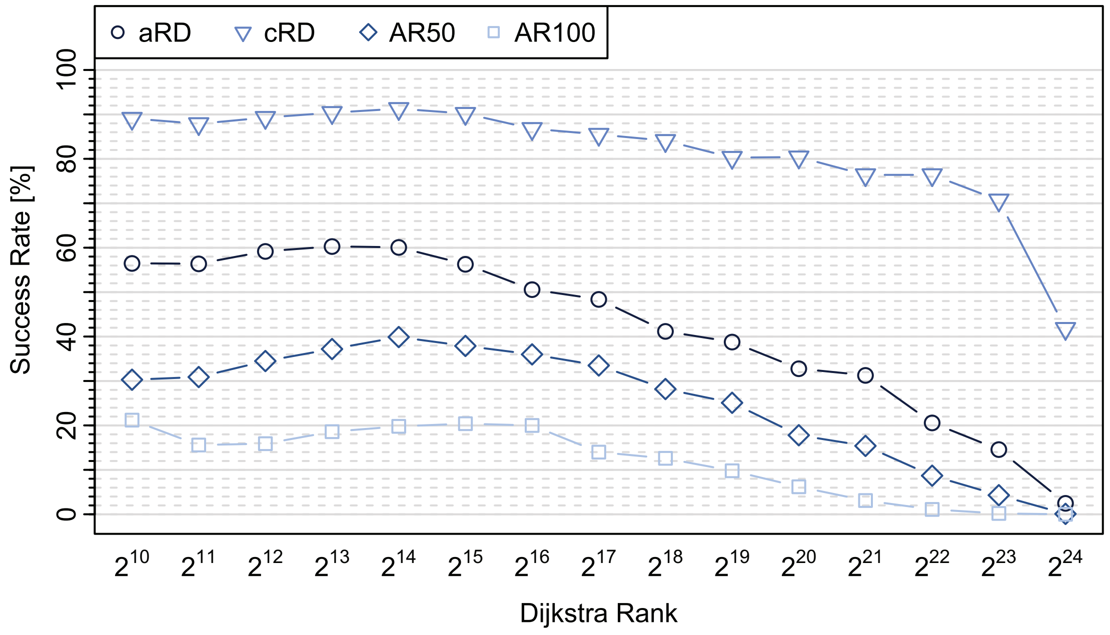


(b) *Arc-Expanded Graph Model (Graph G1e)*

Figure 11.8: *Corridor size (turn model) measured as a relative factor to the shortest path.*



(a) $\delta_t = 100$



(b) $\delta_t = 200$

Figure 11.9: *Success rate in relation to the length of the shortest path for a detour corridor; error probability set to ten.*

reducing the size of δ_t in every recursion. This change results in far higher vertex counts, though.

As to be expected, the turn model excels for the turn-based driver models. The close relation between the corridor model and the driver almost guarantees this. We do require some turns for the method to work, though. A three-turn corridor already exhibits quite favorable success rates for most of the driver types. Only the AR50 and AR100 drivers provide an exception to this. The long reaction times of five and ten seconds respectively can be seen as unrealistic, though. A AR10 driver performs very well (not shown), although a reaction time of one second should be equivalent to a single turn anyhow.

Turning to rather extreme parameters (six turns and fifty seconds of random deviation c.f. Figure 11.12), we can see that corridors can handle even the most unrealistic scenarios rather well; the turn model outperforms the detour model (without recursion), except for very long routes in combination with the AR50 and AR100 driver.

Based on our experiments, we deem the turn model to be the most promising one. While the turn model performs perfectly for the turn-by-turn drivers, it even provides a good quality for the detour based drivers (AR model).

The detour models (both recursive and non-recursive) perform slightly better on the detour-based drivers than the turn model, which is to be expected. For some parameter choices, however, the turn model of a corridor can already outperform the detour model even for those drivers. The turn-by-turn drivers perform significantly worse in the detour model. At the cost of a high increase of vertices, we can switch this around by including all vertices that are connected to a road that is reachable within δ_t. We can, however, hardly justify this increase in vertices as the turn model already performs very well.

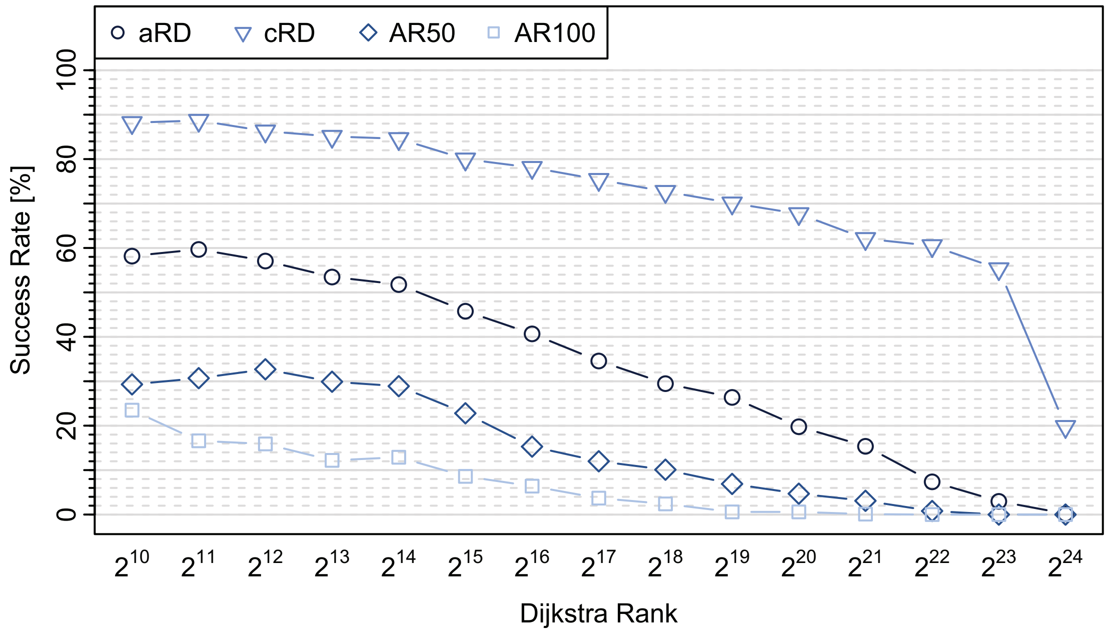
The computational methods of both variants are closely related and we invite the reader to make their own selection of models and parameters that best fit the desired purpose.

11.6 Attached Component Filtering

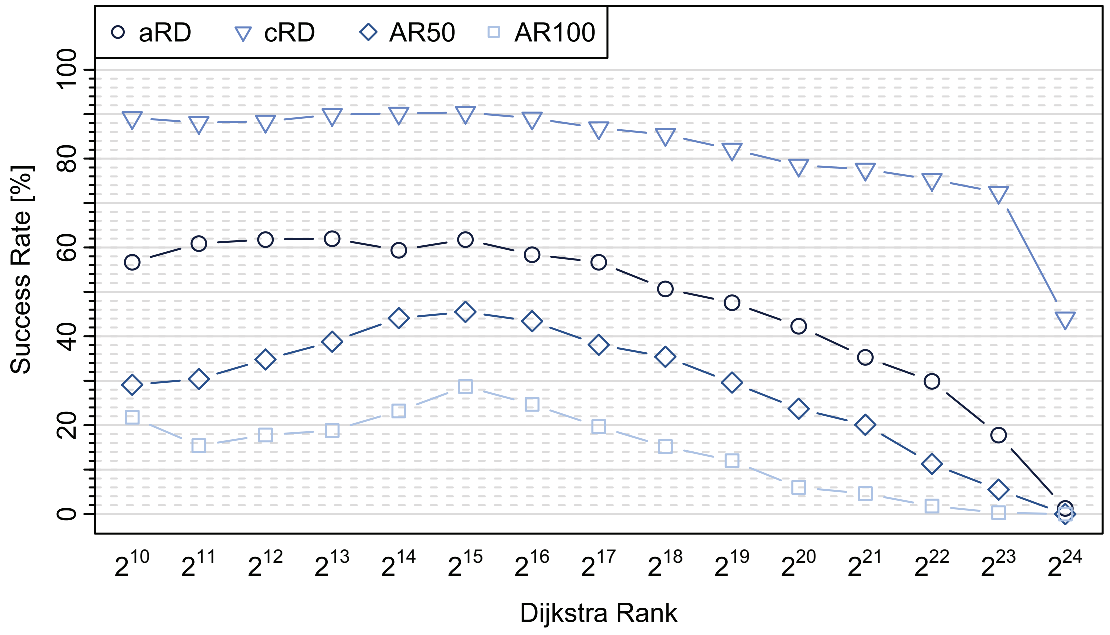
In the following sections, we present an experimental evaluation that shows the effects of our filtering methods for attached components.

11.6.1 Corridor Size Reduction

The reduction of corridor size severely depends on the region that the path is located in. For example in cities, we can barely see any effect. The reason for this is the graph density. For longer range queries, the fact that a highway exit and highway entry are two distinct arcs prevents us from detecting them in our filters. Nevertheless, we can

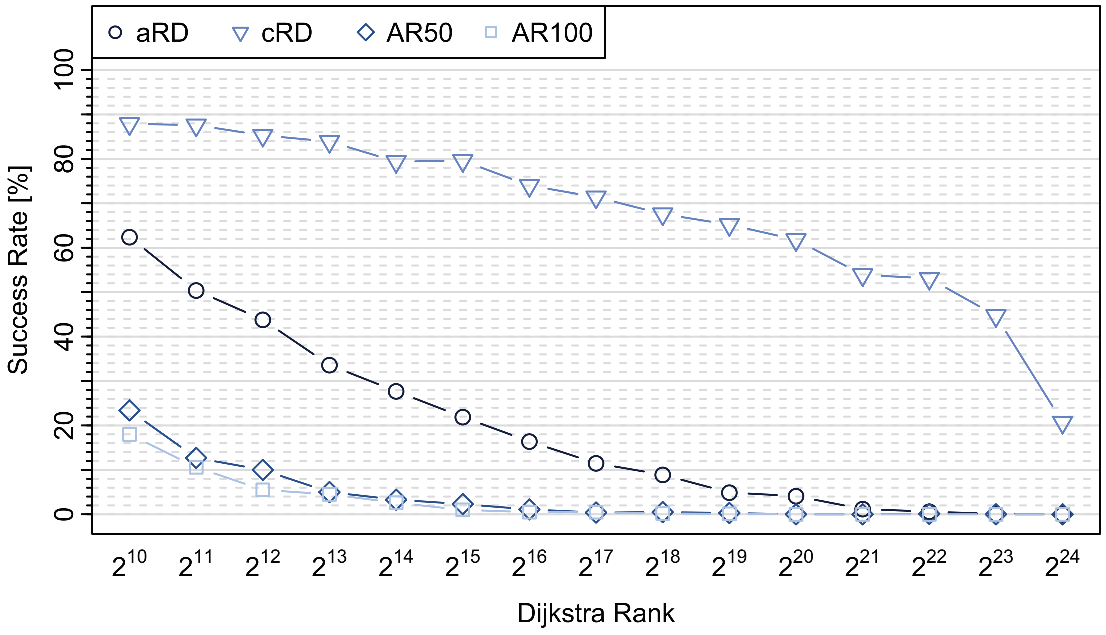


(a) $\delta_t = 100$

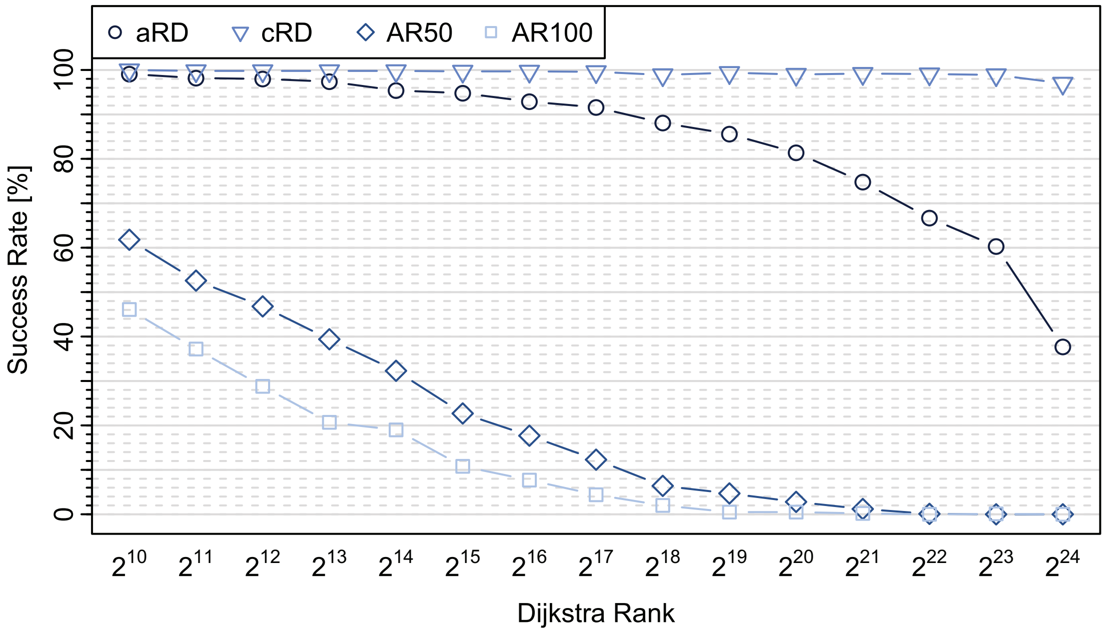


(b) $\delta_t = 200$

Figure 11.10: *Success rate in relation to the length of the shortest path for a recursive detour corridor (depth of three); error probability set to ten.*

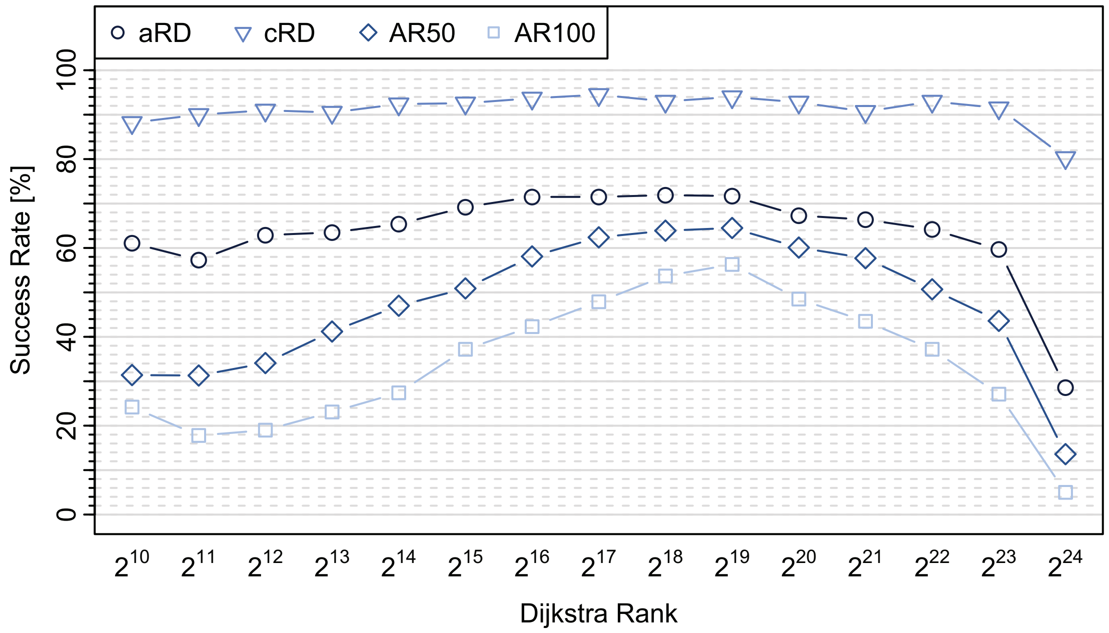


(a) *One turn*

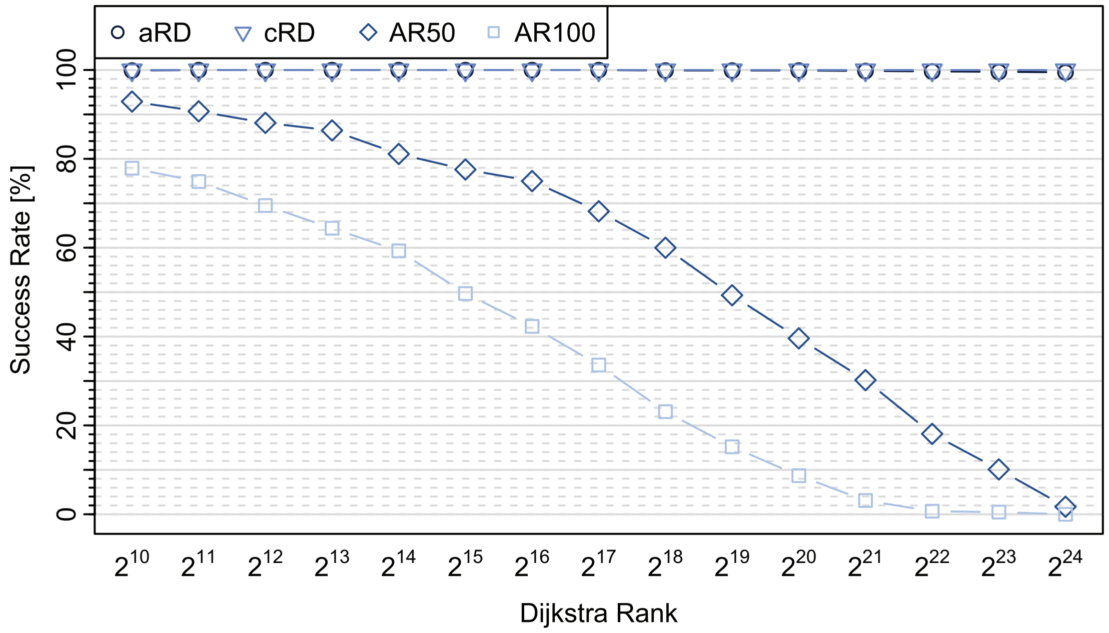


(b) *Three turns*

Figure 11.11: *Success rate in relation to the length of the shortest path for a turn corridor; error probability set to ten percent.*



(a) *Detour corridor;* $\delta_t = 500$



(b) *Turn corridor; six turns*

Figure 11.12: *Success rate in relation to the length of the shortest path for a detour corridor; error probability set to ten percent.*

see a large reduction of vertices, when we apply either one of our filtering methods. We visualize the results in Figure 11.13 for both methods.

If we only desire the calculation of an alternative graph, further filtering could target deviations that only offer little variation or very high stretch. Both kinds of deviation cannot be detected using our implemented filtering methods, though. We present an overview of our filtering results in Figure 11.14 and Figure 11.15. The turn corridors can be reduced to about three-quarters of their unfiltered size. For a larger number of turns, both the general reduction and the benefit of using bridge-filtering over topological-tree-filtering increase. As expected, the detour model holds a lot more unwanted information. This results in a far greater reduction rate when we apply either one of our filtering methods. The differences between bridge-filtering, which can achieve a reduction of forty percent, and topological-tree-filtering, which only reaches around thirty percent, are far greater in the detour model. This further confirms that long detours in rural areas often contribute little to the alternative graph.

11.6.2 Robustness

The large reduction in size comes along with a reduced robustness as well. Unless we can get hold of actual data that indicates where wrong turns happen, it is hard to argue how to reduce a corridor in size. For example, we would expect a bridge-filtering to be fully reasonable close to a turn that most likely will never happen. If a mistake were common at the given intersection, however, we would potentially only want to filter using the topological-tree removal. Since both techniques remove a lot of the u-turn information along a path, the success rates drop significantly, as shown in Figure 11.16, when compared to Figure 11.11 and Figure 11.9.

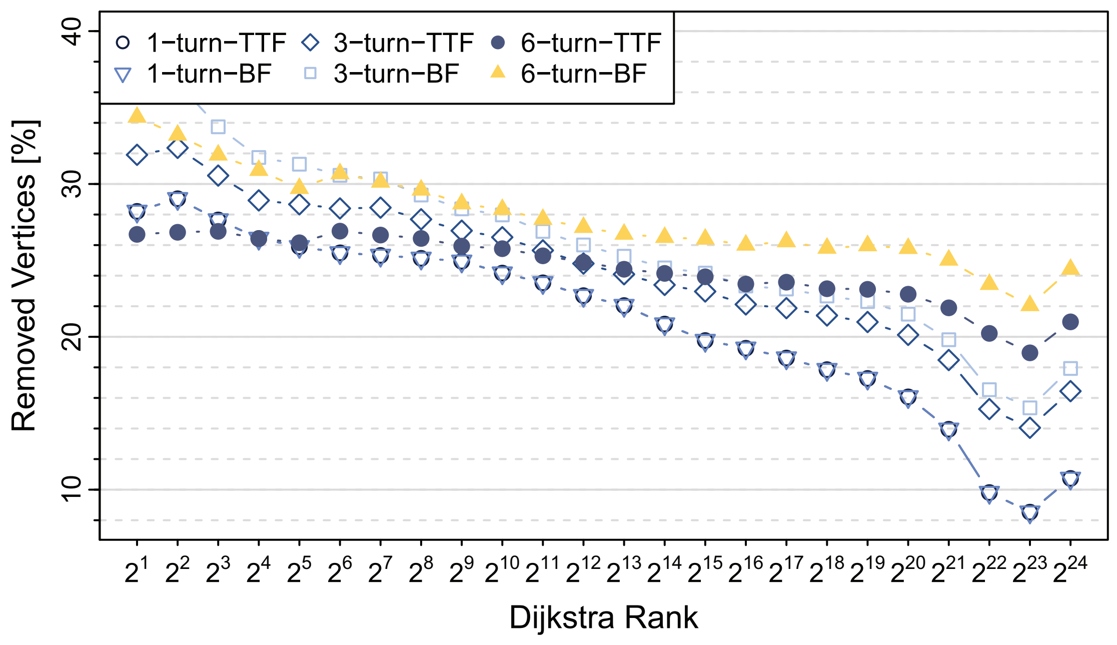
The filtering has its own justification, when considering corridors as a method for the computation of alternative graphs, though. Attached components cannot contribute to an alternative path in a reasonable way. Depending on the cost of the alternative route extraction mechanism, applying a filter beforehand to reduce the workload can be beneficial. Especially as our filtering methods both offer linear asymptotic guarantees.

11.7 Corridor Computation

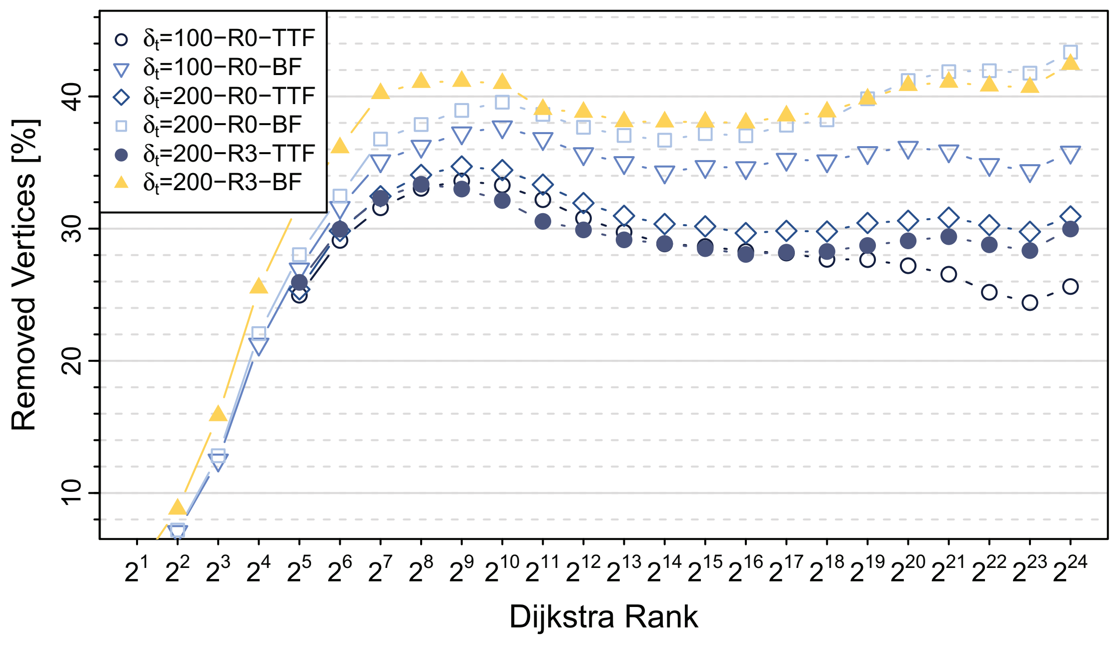
For our considerations about an efficient implementation, we focus on the turn corridor model. The ideas and methods that we develop in the following paragraphs can be applied to the detour paradigm in the same way, though.

11.7.1 Conceptual Approaches

Computing turn corridors, especially if we consider their potential size, seems like a difficult task at first. The upper bound on their size, however, is rather large and we

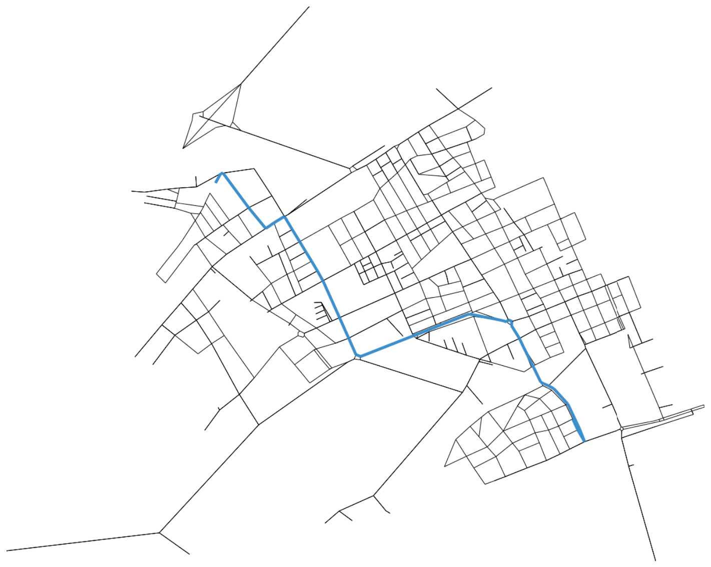


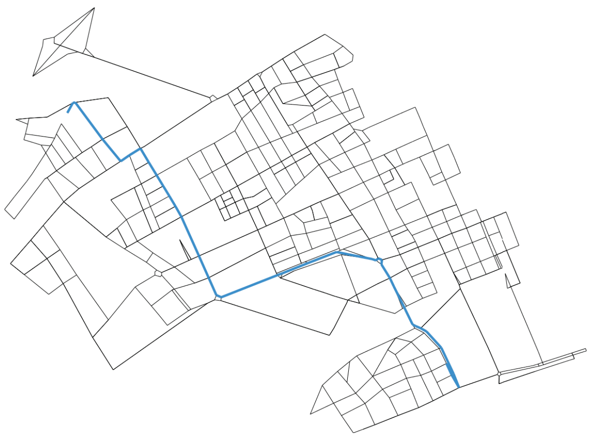
(a) *Turn Corridors*



(b) *Detour Corridors*

Figure 11.13: *Reduction of corridor size for a selection of corridor types (TTF=Topological-Tree-Filtering, BF=Bridge-Filtering). The detour corridor is specified with respect to the allowed detour (δ_t/10 s and the recursion depth (R).*


(a) *Unfiltered*


(b) *Topological-Tree-Filtering*

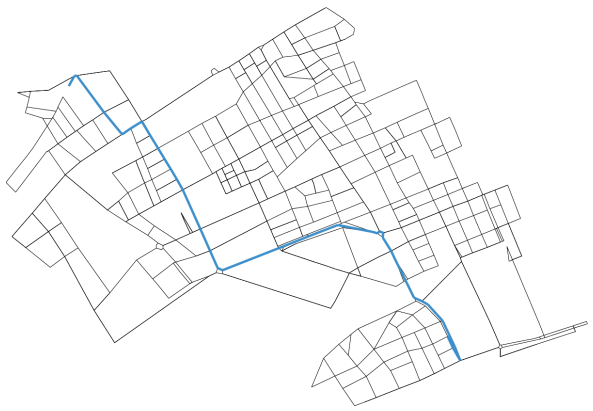
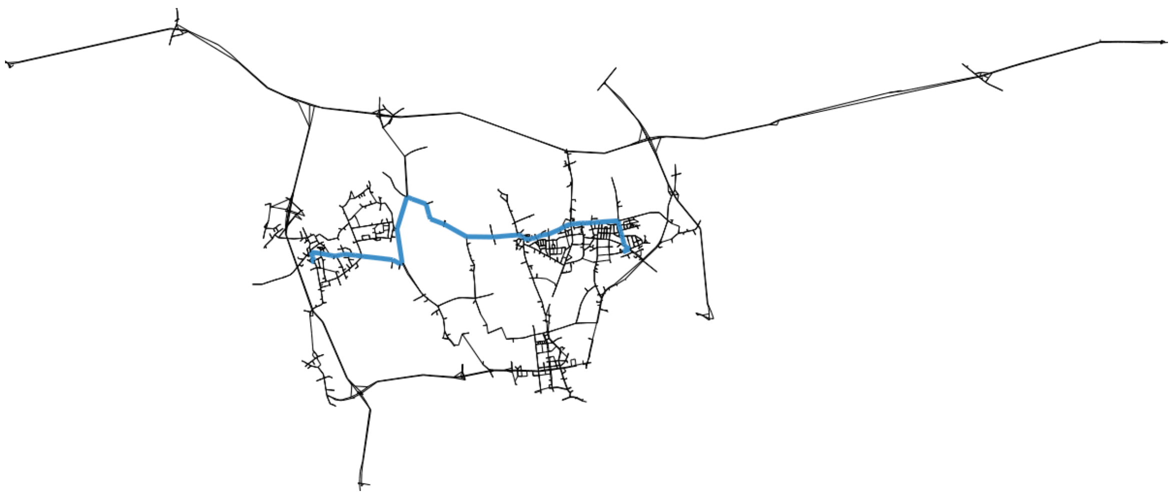
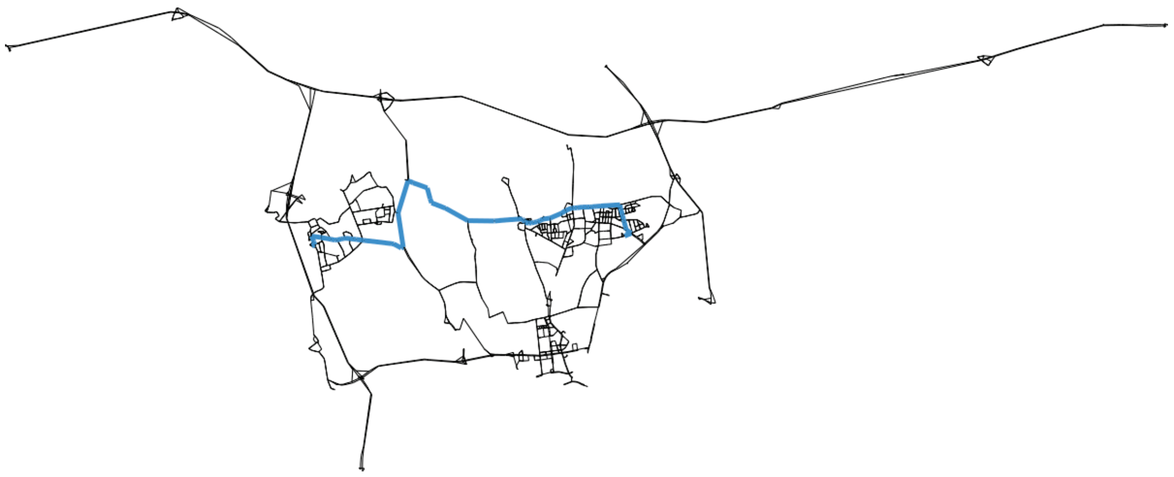

(c) *Bridge-Filtering*

Figure 11.14: *Visualization of our filtering methods. The example shows a five turn corridor on a Query of Dijkstra rank two to the power of nine.*


(a) *Unfiltered*


(b) *Topological-Tree-Filtering*

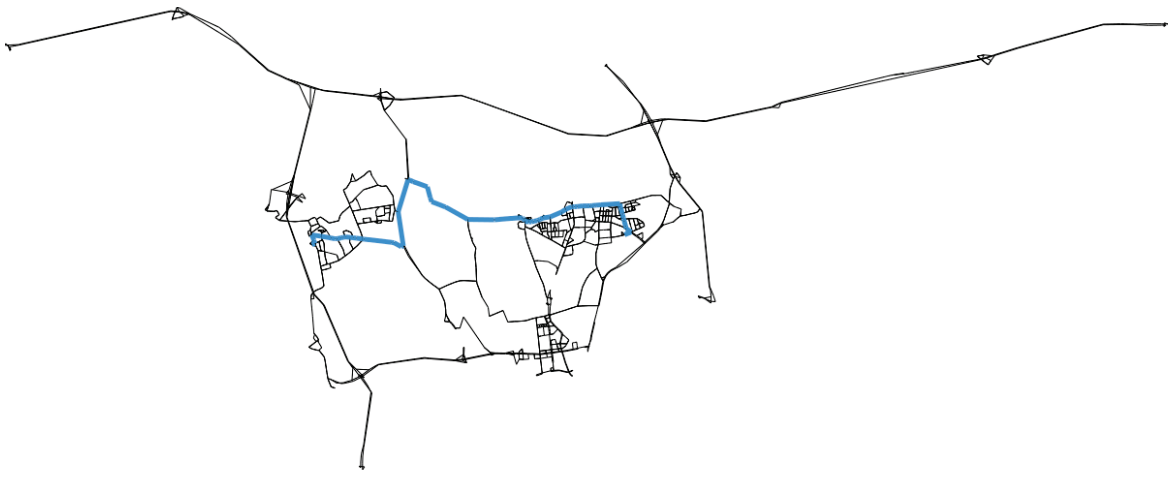
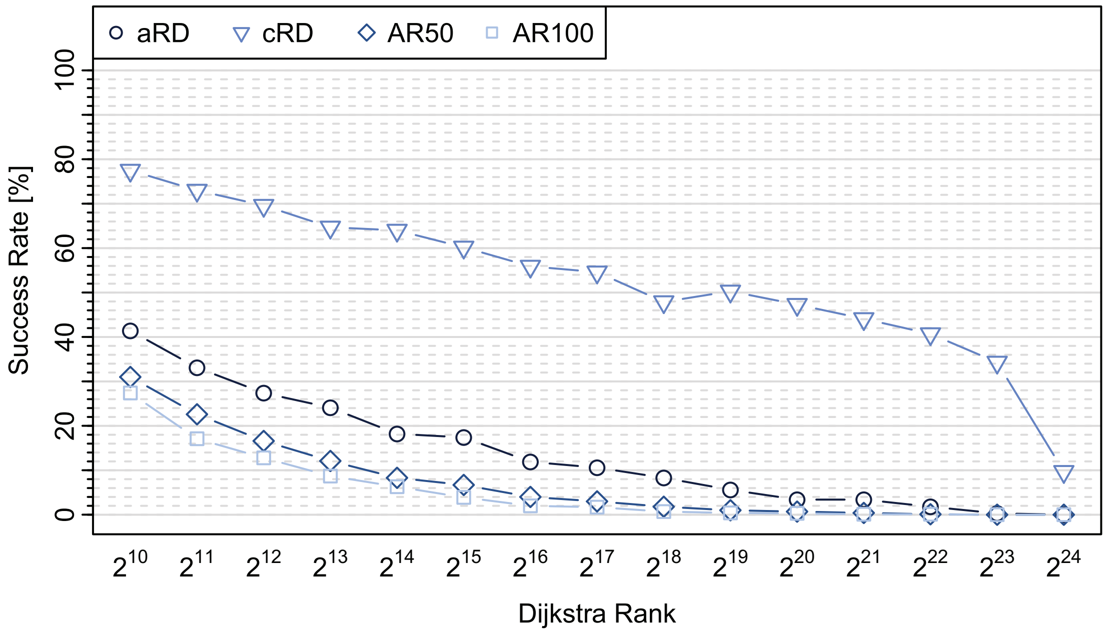
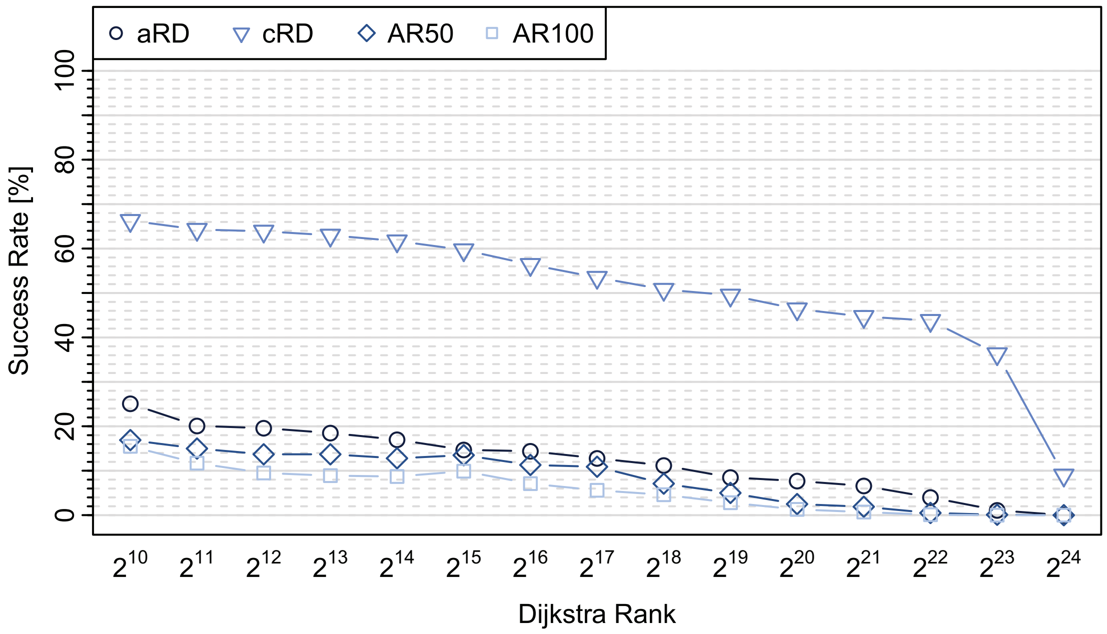

(c) *Bridge-Filtering*

Figure 11.15: *Visualization of our filtering methods. The example shows a detour corridor of recursion level three with a initial detour of 20 seconds ($\delta_t = 200$). The used search is a query of Dijkstra rank two to the power of thirteen.*



(a) *Three turn corridor*



(b) *Detour corridor ($\delta_t = 200$, three recursions)*

Figure 11.16: *Success Rates on a bridge-filtered corridor. The topological-tree-filtering method yields very similar results.*

Algorithm 11.1 Simplistic Turn Corridor Computation

Input: A corridor $\Gamma$of degree d to a target t
Output: A corridor of degree $d+1$

```
1: $D := \{v \in V | \exists a = (u,v) \in A, v \notin \Gamma, u \in \Gamma\}$
2: for all $v \in D$ do
3:     $\Gamma = \Gamma \cup \Pi_{v,t}$
4: end for
5: return $\Gamma$
```

have already seen that corridors contains only a fraction of the theoretically possible vertices. A main reason for this is the limited number of alternative routes we can find even with techniques like X-BDV, as – for the most part – the directions guide the driver back to the initial shortest path.

For the computation of a turn corridor, we first present a preliminary algorithm (see Algorithm 11.1) that describes the general concept of our approach.

Though very simple in principle, the straightforward implementation exhibits a lot of optimization potential that we can exploit for an efficient algorithm for turn corridors. The expected form of a turn corridor exhibits some optimization potential. For the most part, we expect a small set of different shortest paths to the target and very short detours along these paths. So, let us first take a step back and examine a turn corridor algorithm based on Dijkstra's algorithm:

Dijkstra-based Corridor. To compute a turn corridor with Dijkstra's algorithm, we instantiate an implementation of the Algorithm searching backwards from the target, searching for the source s. As depicted in Algorithm 11.2, the further progress extends this potentially incomplete shortest-path tree whenever it should become necessary.

Computing the deviation vertices is straightforward. We traverse all vertices that have just been added to the corridor Γ (initially this set is comprised of the shortest path itself) and their adjacent arcs. All vertices that can be reached from one of the vertices of the last iteration and have not been added to the corridor are stored on a stack for further processing[3].

For every such vertex v, we need to find the shortest path to the target. To do so, we check whether v has already been settled by our instance of Dijkstra's algorithm. If this is not the case, we first continue the initial instances of Dijkstras' algorithm until we settle v. Afterwards, we track the parent pointers until we reach the first vertex located within the corridor. By adding all the traversed vertices to the corridor, we compute the shortest-path hull of $\Gamma \cup \{v\}$.

The set of vertices extracted in this way forms a valid corridor which directly follows

[3]The order we use here is not relevant for the correctness and ignored in Algorithm 11.2. It offers better locality, though.

Algorithm 11.2 Dijkstra-based Turn Corridor Computation

Input: A corridor $\Gamma$of degree d to a target t in the form of a tree pointing at t, a partially completed instance of Dijkstra's algorithm
Output: A corridor of degree $d+1$ in the same form

```
p[·] =⊥                                              ▷ Parent pointers to every vertex
$D := \{v \in V \mid \exists\, a = (u,v) \in A, v \notin \Gamma, u \in \Gamma\}$
for all $v \in D$ do
    while not settled[$v$] do
        step_dijkstra()
    end while
    for all $(u,w) \in \Pi_{v,t}, u \notin \Gamma$ do
        $\Gamma = \Gamma \cup u$
        $p[u] = w$
    end for
end for
return $\Gamma$
```

from the correctness of Dijkstra's algorithm. Every vertex v in the corridor is only added once and examined at most $\delta(v)$ times, each arc at most looked at from both its origin and destination. As a result, the running time of Dijkstras' algorithm dominates the execution of the corridor method.

For small distances between s and t, this algorithm might even perform well enough, losing its practicality on larger instances, though, as the running time of Dijkstra's algorithm gets longer. Nevertheless, we utilize it as a comparison algorithm and refer to this simple version as `dijkC`.

The most obvious approach at this point is to turn to speed-up techniques. A direct variation of the shortest-path tree approach is to accelerate its computation (e.g. by employing the PHAST algorithm).

The PHAST algorithm only shifts the general problem towards larger graphs. The execution still requires a considerably long time and the full benefit of the algorithm only shows in an amortized setting in which many trees can be computed in parallel. In addition, unless we compute the search space for every source, we compute the full shortest-path tree in every iteration. We implemented it as `phastC` for a comparison, though.

For our own method, we employ a different algorithm that, making use of a point-to-point technique, might seem to offer a strange choice at first.

11.7.2 Contraction Hierarchies-Based Corridors

Opposed to the single (iteratively extended) one-to-all query, CH is a point-to-point technique which implies the requirement for a different calculation method for our corridors. Even though a CH provides an extremely fast point-to-point shortest-path query, the experienced size of a corridor (c.f. Figure 11.7) already indicates a still high computational cost due to the large amount of necessary queries. We gradually increase the performance as shown in the next paragraphs until we arrive at our proposed solution.

The straightforward implementation of a CH-based corridor algorithm would be to follow the design presented in Algorithm 11.1. We refer to this version as `cCH`. Every computed shortest path is calculated to its full length. During the traversal, however, we stop at the first vertex already in the corridor. The larger the corridor grows, the greater the chance gets to encounter such a vertex early. The nature of the point-to-point algorithm requires us to compute the full path in every query, though. We can only limit the extraction process to consider only a prefix.

We can use the technique introduced in the variant of a many-to-many version of the CH, to reduce unnecessary overhead. This variant operates differently for the forward and backward parts of the query algorithm. In the many-to-many variant, we are interested in a $S \times T$ sized table, containing the distances between all $s \in S$ and all $t \in T$. The algorithm first performs the backwards search for all t vertices to exhaustion, computing a list of tentative distances to all vertices reachable in $G^{\downarrow}$. During the forwards search, all these values are scanned to compute all distances in $\mathcal{O}(S + T)$ passes over a CH search space rather than $\mathcal{O}(S \cdot T)$[4]. The same thing can be done to improve the performance of the initial algorithms in only a slight modification, referred to by `cBCH`. In `cBCH`, we eliminate the requirement of traversing the backwards graph multiple times. By traversing it completely, we add some additional cost to for short-range queries, though.

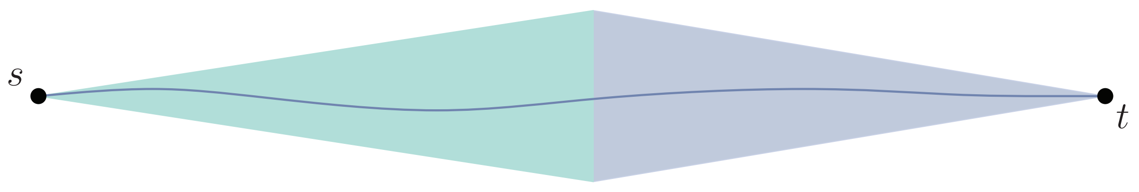
What holds for the part in $G^{\downarrow}$ does also hold for $G^{\uparrow}$. Computing a large series of shortest-path queries, we are likely to traverse major parts of the search space in $G^{\uparrow}$ a repeated number of times. Borrowing ideas that we have already discussed in Chapter 9, we present a third version of a CH-corridor in the upcoming Section.

11.8 Turn Corridor Contraction Hierarchies

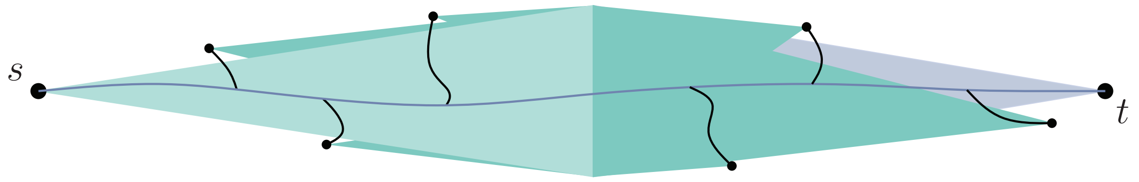
Our method of choice, initially developed for the turn corridor and therefore referred to as `tcCH` or turn corridor CH, starts of in the same way as `cBCH`. In an initialization phase, we perform an exhaustive upward search in $G^{\downarrow}$, starting at the target t.

The step towards the `tcCH` is made through the introduction of a method to prune vertices during the search in $G^{\uparrow}$. Conceptually, the process is divided into three different

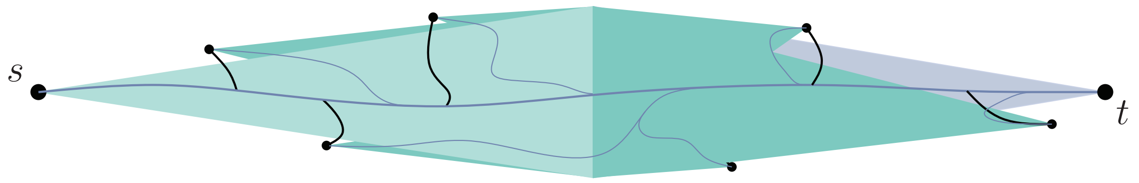
[4]The different scans of distance values of course require time in the order of $\mathcal{O}(S \cdot T)$.



(a) *Initial Search*



(b) *Deviation Vertices and Second Search*



(c) *Tree Extension*

Figure 11.17: *Schematic of `tcCH` progress.*

phases during which we keep track of a few flags for every vertex: the `finalized` flag, which describes whether a distance value is tentative or final, and the `extracted` flag that indicates whether a vertex has been extracted for the corridor already.

1 - upward search: The upward search is comparable to the approach already taken in Chapter 9. We locate all vertices in $G^{\uparrow}$ that are reachable from any of the deviation vertices. Since we do not require any distance information in this phase, we can perform this search for all vertices at the same time. Our experiments show close to no difference between a sequential and a simultaneous generation of the search space, though. For pruning purposes, we can stop the upward search at every vertex with an already finalized distance value.

2 - sweep: This phase propagates distances along the partial shortest-path tree. The process is identical to the general approach of PHAST or its localized variant [DGNW13, DGW11]. Vertices are processed in reverse contraction order (incomparable vertices – vertices that are on the same level – ordered arbitrarily). Every vertex processed at this stage gets its `finalized` flag assigned to `true`.

3 - path unpacking: The previous stage results in a (set of) shortest path(s), partially

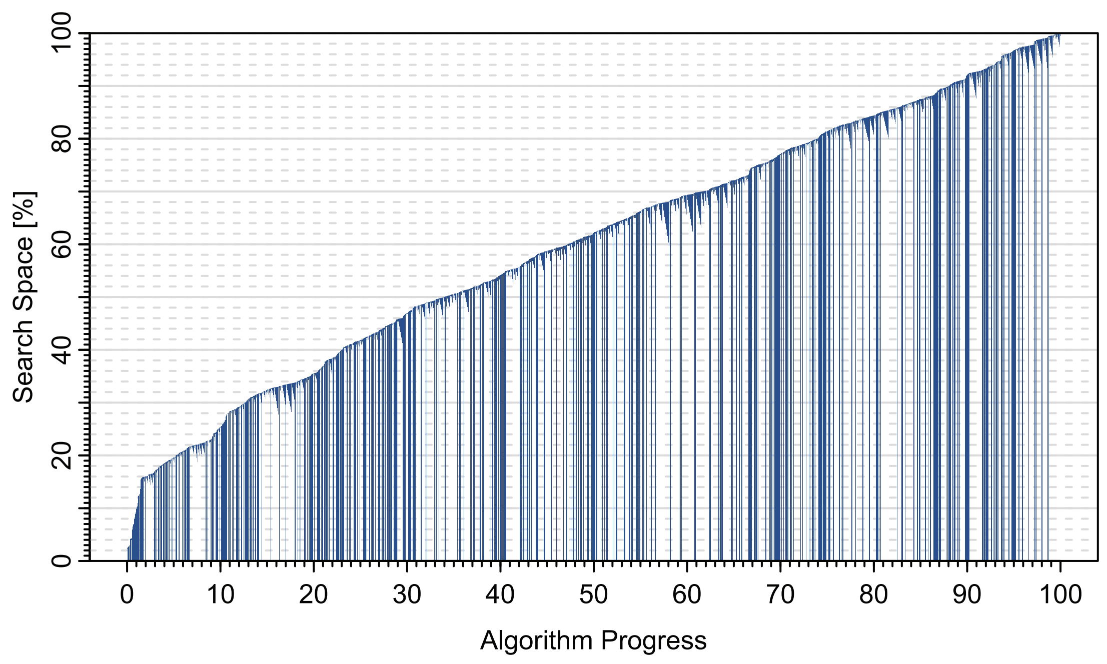


Figure 11.18: *Covered search space size over the course of a six turn corridor generation.*

consisting of shortcuts. The final stage of our algorithm unpacks these shortcuts to extract the actual shortest-path information and computes the deviation vertices (if necessary) for the next cycle.

We describe the different phases in more detail in the following paragraphs.

Upward Search. The upward search starts with an initial set of deviation vertices. In the first round that computes the corridor of degree zero, this set only contains the source s of the query. For all the deviation vertices, we perform the upward search in $G^{\uparrow}$, using the bucket approach already described in Chapter 9 (compare Algorithm 9.1 and Algorithm 11.3).

Due to previously executed queries, we can prune the query at all vertices for which the distance to the target has already been finalized. This is the case, after the initial search, for all vertices that can be reached from s in $G^{\uparrow}$. In addition, this also holds for all vertices on the shortest path, especially the part in $G^{\downarrow}$. The searches starting at the deviation vertices, c.f. Figure 11.17, can most often be pruned early on (compare the near linear growth in Figure 11.18).. A majority of the different queries requires only a single additional vertex or even none at all.

Algorithm 11.3 Compute Sweep Space

Input: A graph processed into a CH, a set of deviation vertices D, a set of `final` flags, and a target t

Output: The set of vertices to sweep in the next step of `tcCH`

```
1: $\mathcal{Q}[\cdot] := \emptyset$
2: for all $v \in D$ do
3:     $\mathcal{Q}[\ell[s]] = \mathcal{Q}[\ell[s]] \cup s$
4: end for
5: $\mathcal{S} = \emptyset$
6: for all $l \geq \min_{v \in D} \ell[v]$ do
7:     for all $v \in \mathcal{Q}[l]$ do
8:         $\mathcal{S} = \mathcal{S} \cup v$
9:         for all $(v, w) \in G^{\uparrow}$ do
10:            if not finalized[$w$] then
11:                $\mathcal{Q}[\ell[w]] = \mathcal{Q}[\ell[w]] \cup w$          ▷ $\ell[v] < \ell[w]$
12:                finalized[$w$] = true    ▷ even though the distance is still tentative
13:            end if
14:        end for
15:    end for
16: end for
17: return $\mathcal{S}$
```

Sweep. The sweep phase is identical to the one of HiDAR and R-PHAST [DGW11]. From the topmost vertices to the lowermost, we scan arcs in $G^{\uparrow}$ and set the distance value. This process extends the tree directed at the target by adding the newly discovered vertices. During the process, their respective final flag is set to `true`. Some of these vertices are connected only via shortcuts, though. Since we have to extract them for the discovery of deviation vertices anyway, we can also finalize the distance values along them.

Path Unpacking. To compute the deviation vertices for the next step (and the final corridor in the last iteration), we need to extract all prefixes of the shortcuts. The part we are concerned with is given by all vertices in the beginning of the shortcut that are not yet part of the corridor.

For finding out whether a vertex is part of the previously extracted ones, we keep a flag (`extracted`) for every vertex that indicates the participation of said vertex in the shortest-path tree directed at the target. If this flag is set to `true`, it also displays that the connection to the tree is in the form of ordinary arcs and not in the potential form of a shortcut. Any vertex still connected via a shortcut (`extracted` equals `false`) triggers an unpacking process. During the extraction, we can either use the recursive method or access pre-unpacked shortcuts. For the recursive method, we can discard every suffix of a shortcut for which the middle vertex has been extracted. Using pre-unpacked corridors, we can stop traversal at the first vertex that is already part of the corridor. The latter method ensures the avoidance of unnecessary workload due to the repeated unpacking of shortcuts.

During the extraction, or traversal, of the path, we can set the respective distance values and also assign the `finalized` flag a value of `true`. The sub-path optimality, c.f. Section 2.4.2, guarantees that all encountered distances are optimal. This assignment allows for earlier pruning in the search space generation phase.

11.9 Experimental Evaluation

Our algorithm for computing a turn corridor performs well, even though its size can grow rather large. We are able to compute a turn corridor of reasonable degree (up to six turns) fast enough for any interactive application (less than fifty milliseconds). We show the validity of our optimizations in the comparison to the conceptual methods that we introduced during the development of our `tcCH` method.

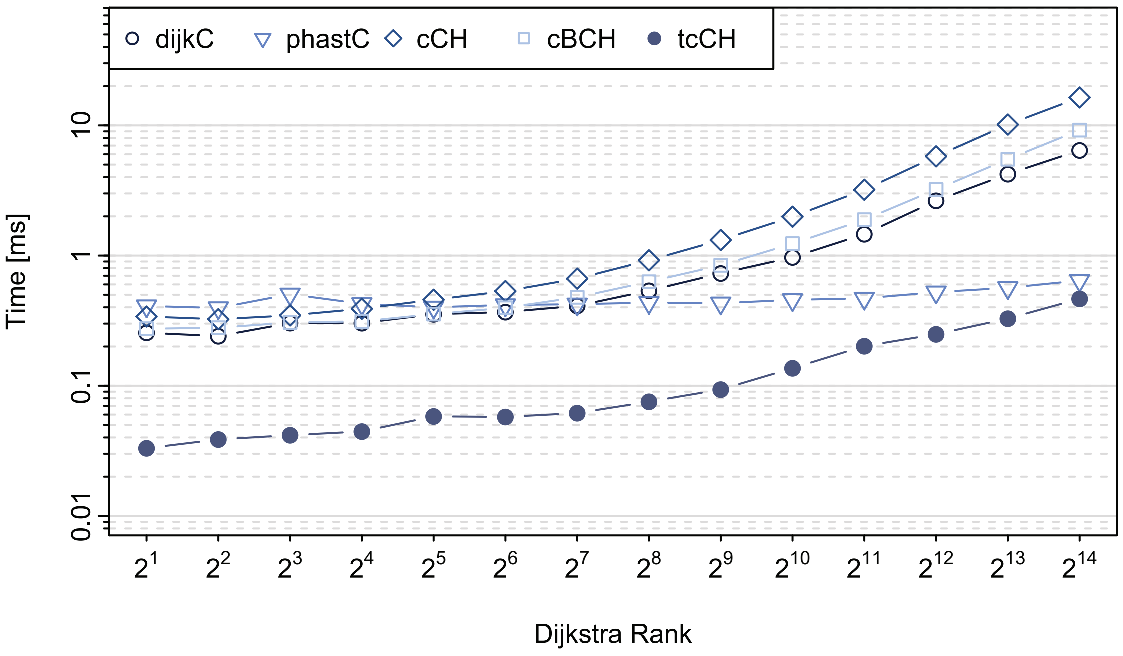
The validity of the different corridor methods strongly depends on the size of the used graph. Whereas Dijkstra's algorithm is fast enough for interactive applications on small graphs, the running times of networks like the German road system are forbiddingly high. Our evaluation on the full selection of inputs specified in Table 6.3 paints an interesting picture.

For smaller instances, due to its simple extension to search for a deviation vertex without starting anew, Dijkstra's algorithm can even outperform `cCH` and `cBCH` for a medium to high number of turns (e.g. Figure 11.19a). This is founded in the need of a CH based algorithm to recompute many parts of the different queries again and again. The larger the cost for an instantiation of Algorithm 2.2 gets, however, the better the CH versions perform (c.f. Figure 11.19b).

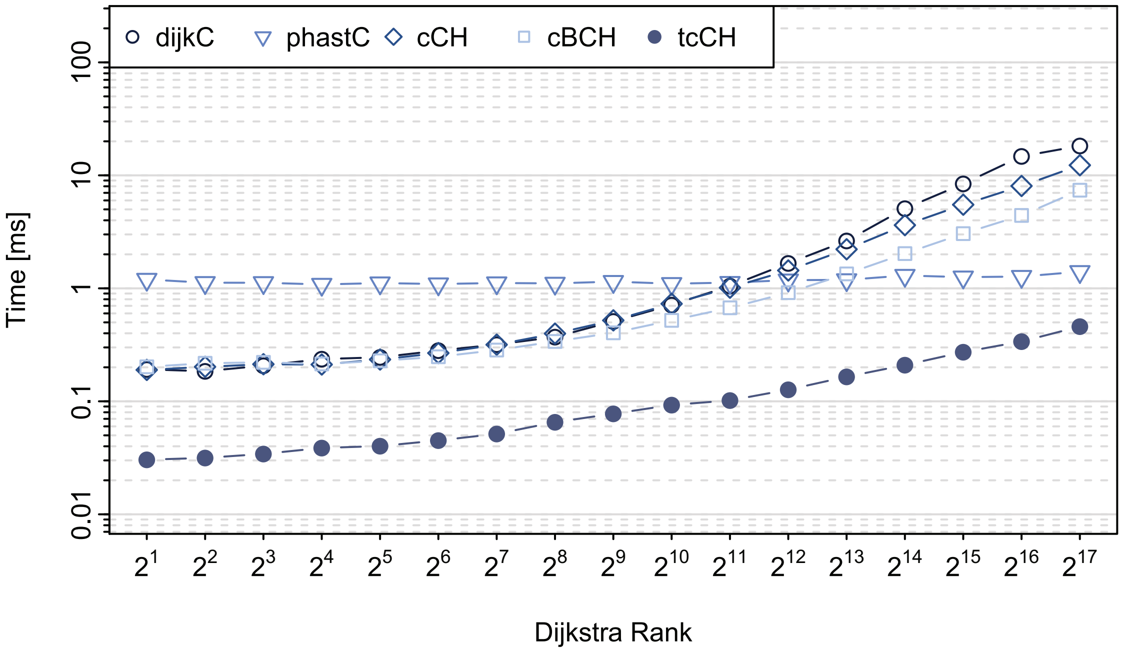
In general, `cBCH` is better than `cCH`, except for very short-ranged queries on large graphs. For graphs the size of Baden-Wuerttemberg and larger, `dijkC` is outperformed by `cBCH`, even if we consider six turns. The `phastC` version remains faster for the longer paths, though.

Considering a full shortest-path tree does not offer a viable implementation choice if the graph gets much larger. Due to its cache efficient-scan over the data and the good query behavior, `phastC` offers a viable choice for medium sized graphs. It is, however, only competitive for long-range queries and a high number of turns. In other words, `phastC` can become competitive if we cover a major part of the graph with the corridor. Starting from road networks the size of the Netherlands, `cBCH` begins to outperform the `phastC` algorithm.

None of the techniques can compete with `tcCH`, though. Our method of choice performs best in all tested scenarios and should be employed when solving the corridor



(a) *Luxembourg network (G9); six turn corridor.*



(b) *Adm. Karlsruhe network (G7); three turn corridor.*

Figure 11.19: *Plots showing exemplary running times of the various corridor algorithms.*

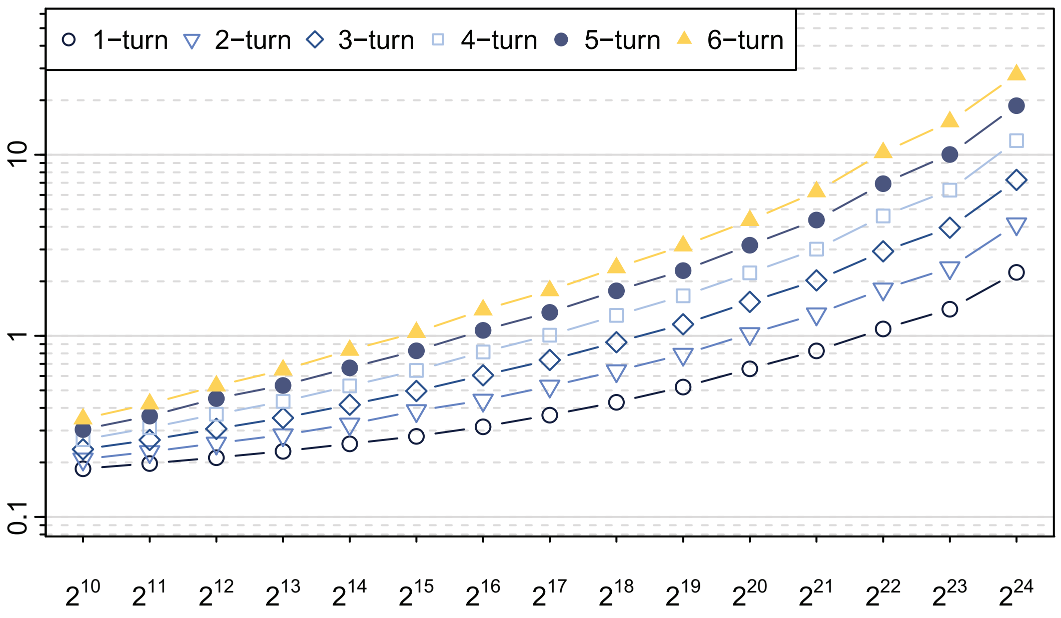


Figure 11.20: *Running times of* `tcCH` *for a different number of turns on the Western European road network (G1).*

calculation problem.

Detour Corridor Performance. Even though we prefer the turn model as our corridor method, we also provide some numbers (Figure 11.21) for the detour model. We can see that for reasonably small values of δ_t, the computational overhead is more than manageable. Even with a recursion depth of three (see Figure 11.21), all queries on the European road network (G1) stay within an acceptable bound of a hundred milliseconds.

Many-to-One Performance. The base of our `tcCH` can also be used as a general many-to-one query algorithm. As such, the algorithm is comparable to the many-to-many technique already described in [GSSV12]. The extension of the shortest-path tree down the hierarchy allows for earlier pruning and an overall good query performance.

Figure 11.22 indicates that, even though we have to actually traverse the found shortest path to extend the backwards tree, our method requires only a minimal number of different sources to perform faster than the standard one-to-one queries; for the one-to-one queries, we do not even extract the full path in this setting. In fact, searching from more than ten targets, we outperform the standard CH query. This method can, of course, be used to find many targets, starting at a single source, as well.

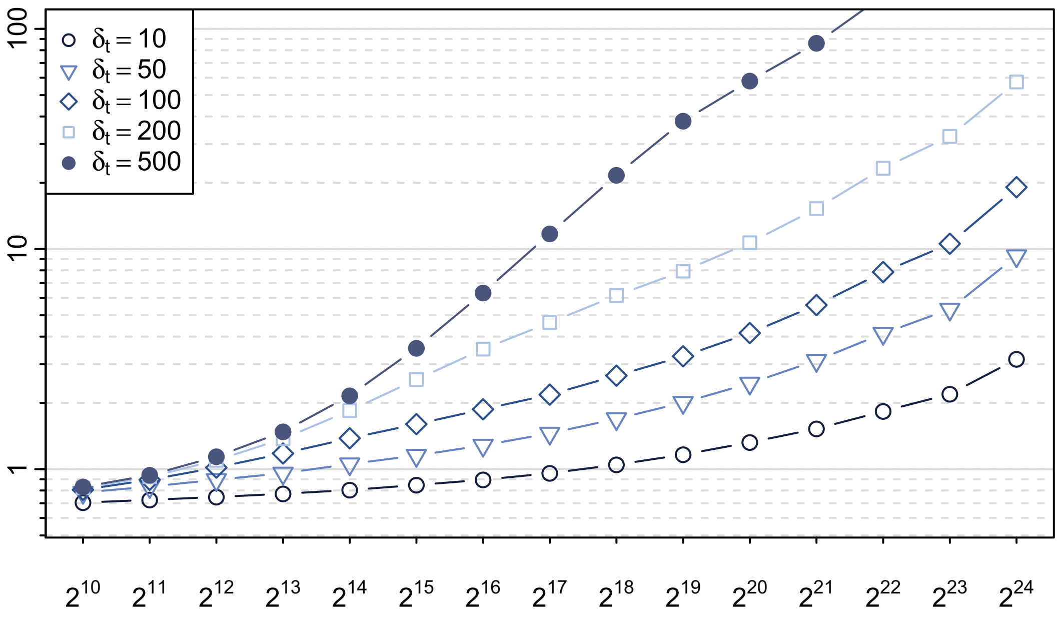
11.9.1 Corridor-based Alternative Routes

In terms of alternative routes, corridors seem to lack behind most of the other techniques, at least for long-range queries. As a alternative graph they are not presentable to a human, due to the explosion in decision arcs as well as total-distance ($\mathcal{D}_{tot}$) that we can see in Figure 11.23. This should not affect their use as an alternative graph, though. The general success rate is not high enough for long-range queries, when compared to other methods. For short-range to mid-range queries, they can actually be considered reasonable.

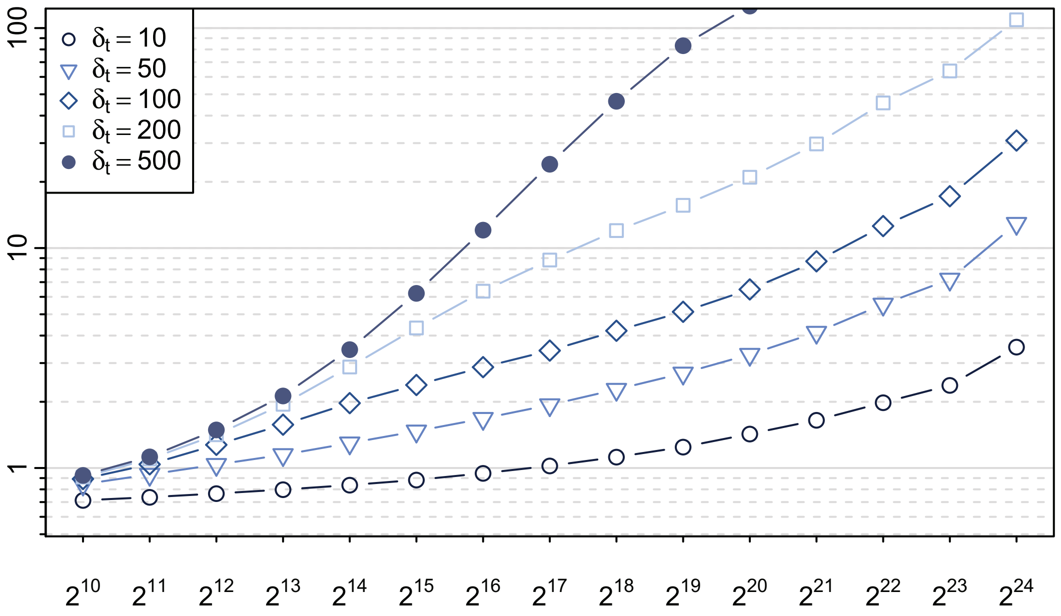
The quality values for the criteria of Definition 5.3 are comparable to via-vertex alternative routes.

11.9.2 Conclusion

Corridors, as we introduced them, offer a wide range of applications. Next to their original design, they offer high quality alternative routes for short-range to mid-range queries. Their main area of application still remains in a hybrid navigation scenario to offer robustness against deviations. While the models chosen for the different drivers are far from being realistic, we think that the number of drivers and the variation between them offers a good impression, nevertheless. The method proposed for their computation can also be used as a general many-to-one/one-to-many technique that only requires a very small load before the initial start-up overhead is amortized. In terms of sizes, we think that a three-turn corridor offers the best trade-off between robustness and size. Given a better model of wrong turn probability, it would be interesting to study tailored versions of corridors, maybe even offering some robustness guarantees.



(a) *Without recursion*



(b) *Recursion depth three.*

Figure 11.21: *Running times of the detour corridor paradigm over various choices of δ_t .*

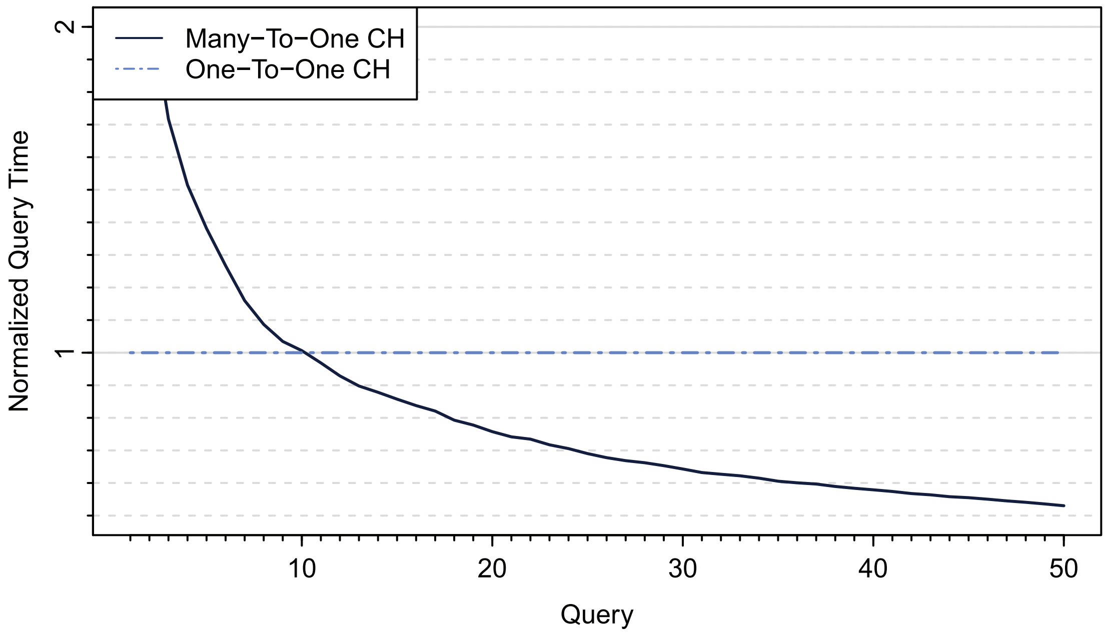
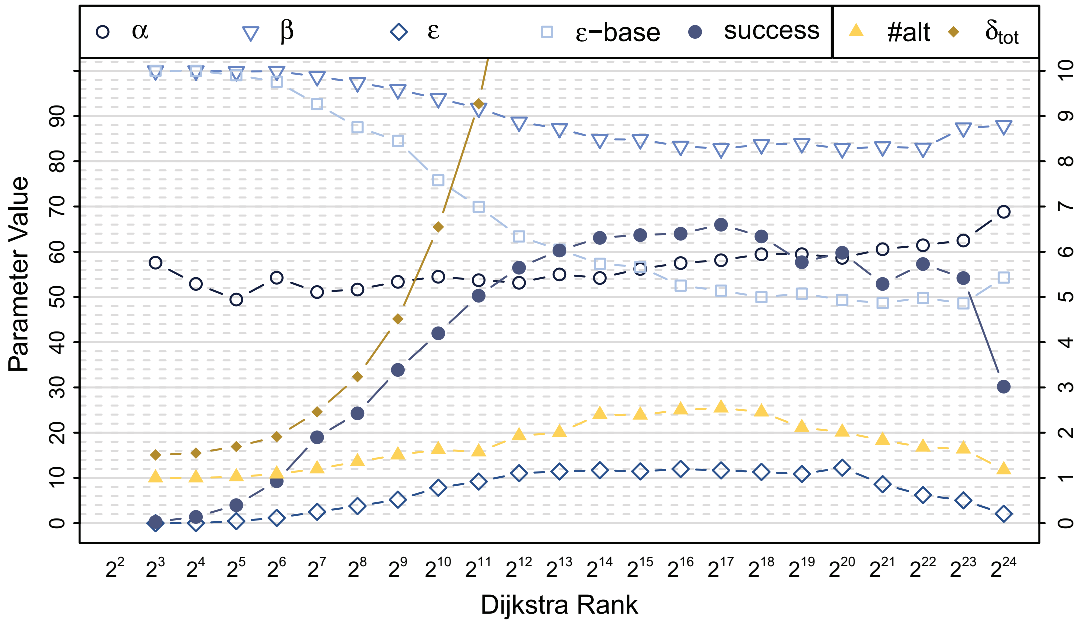
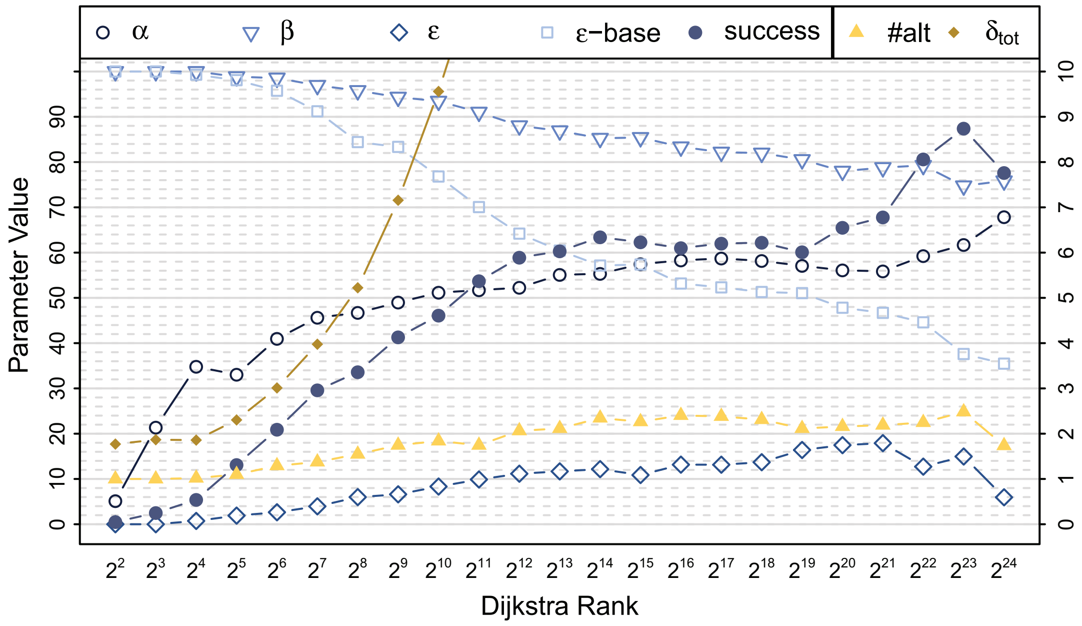


Figure 11.22: *Normalized running time of the underlying Many-To-One CH of our* `tcCH` *algorithm in comparison to a standard one-to-one CH query algorithm. The running times are measured over a series of different shortest-path queries, starting at different source vertices (chosen at random) and directed at the same target. The graph is the Western European road network (G1). Times are normalized in relation to the standard CH query.*



(a) *Detour Corridor*



(b) *Turn Corridor (six turns)*

Figure 11.23: *Development of the quality parameters, according to Definition 5.3, for the corridor approach.*

Chapter 12

12 Method Comparison

Alternative-route techniques, just as the paths they compute, cannot be directly compared as to being better or not. We present a selection of metrics and parameters to highlight their different properties. A technique being worse in a metrics than another one presented does not mean that the approach is useless, though.

12.1 Analysis

Based on the graph and path measures first introduced in Chapter 5 and the additional measures of Section 6.5, we present a detailed quality analysis and comparison of our alternative-route techniques in the following sections. We start off by comparing our newly introduced techniques to the methods found in the literature and then present some additional details.

12.1.1 Running Time Analysis

Table 12.1 shows that our techniques are slower than the fastest method for via-vertex alternative routes. The fastest method requires a far larger amount of preprocessing, though. These cost are a major concern in a production environment if we want to incorporate further parameters or perform the preprocessing on a regular basis.

Using only the ordinary preprocessing of CH and CRP, our techniques require far less computational overhead than the multi-level [LS12] approach. Penalty-based techniques as well as the corridors show a far longer running time than the via-vertex approach. All techniques are within the limits required to be handled successfully during a *http* cycle[1].

[1]While we require less than a hundred milliseconds for an interactive feel, running times of around 200 milliseconds are still below reasonable time-out thresholds.

Table 12.1: *An overview of the average success rates of the alternative-route techniques, as well as the time required to calculate the respective alternative routes. We present success rates in percentage and time in milliseconds. The values for the methods from the literature are taken from [ADGW13] as well as [LS12]. 3-CHV and 3-REV were evaluated on an AMD Opteron 250, clocked at 2.4 GHz using* $2 \times 1MB$ *of L2 cache. The running times of the multi-level approach are only specified to show in how far further preprocessing can be used to speed up the computation even further. For the turn corridor, we used six turns. For the detour corridor, we used the recursive method with three recursion levels and an initial detour of twenty seconds* ($\delta_t = 200, R = 3$)*. Our results are measured on M2, using up to 16 threads when running in parallel. Running times are specified in* $[ms]$*, success rates are given in percent.*

method	first		second		third	
	success	time	success	time	success	time
HiDAR	92.4	16.6	80.1	16.6	66.1	16.6
CRP-π	96.5	113.5	90.2	113.5	80.0	113.5
CRP-∞	95.2	92.2	85.8	92.2	69.6	92.2
turn-corridor	78.9	61.3	58.3	62.0	42.2	62.4
detour-corridor	51.0	87.8	28.6	88.4	14.9	88.6
X-BDV	94.5	∞	81.1	∞	61.6	∞
3-CHV	90.7	16.9	70.1	20.3	42.3	22.1
3-REV	94.2	55.3	80.6	73.0	61.0	91.7
multi-level	90.0	0.1	70.4	0.3	44.2	0.5

All but one of our query types, namely CRP-π (see Chapter 8), remain within a hundred milliseconds on average for random inputs. While the machine used in the experiments for 3-CHV[2] is slower, our general running times are comparable. At the same time, we can provide a far better success rate with HiDAR (Chapter 9). A similar success rate for X-CHV[3] requires a stronger relaxation and results in far longer running times. CRP-∞ (Chapter 10) remains within the postulated bound of one hundred milliseconds and offers the highest success rates of the techniques that remain within the bound, even surpassing X-BDV[4]; this is impressive as X-BDV is usually accepted as an upper bound on how many alternative routes can be found [ADGW13], even though some expensive algorithms can find more paths. In surpassing X-BDV with CRP-∞ and especially CRP-π, we can see that high quality alternative routes are not necessarily describable in a single via-vertex. The general quality of CRP-π does not stand back in comparison to the via-vertex approach. HiDAR offers a higher success

[2][ADGW13], the relaxed version of via-vertex on CH.

[3]Using X levels of relaxation.

[4]The full plateau method on Dijkstras' algorithm.

rate than X-BDV for the third alternative. This effect can be attributed to the order in which routes are chosen, though, as the previously selected routes impact the overlap for further candidates. We study the effects of different route extraction strategies in the next section.

Both corridor methods remain well within the postulated bound. The success rates for first and second alternative paths are too low to justify their usage as an alternative-route technique, though. Since our techniques are all based on the creation of a graph and run X-BDV on it, extracting a second and third alternative comes cheap. Only the graphs created by the corridor methods are large enough to impact upon the running time in a noticeable way.

A Note on Possible Combinations. Some combinations of the techniques presented here seem rather logical. Bader et al. have already shown that the combination of the plateau method and the penalty method [BDGS11] offers better results than the methods on their own. The different underlying speed-up techniques prevent an combination apart from uniting the final alternative graphs. While CRP-π and CRP-∞ use the same technique, their results are too similar to show any significant benefit. The benefit of combining the plateau method, in form of HiDAR, with CRP-π has already been discussed by Bader et al. and is left out for this reason.

12.1.2 Path Extraction Strategies

As previously shown in Figure 6.3, the strategy of how to select paths could potentially have a large influence on the success rates and the quality of the found alternative routes. To check this influence, we performed a series of experiments, calculating success rates for the first three alternative routes, as well as their quality. For strategies, we ordered the candidates in order of their quality with respect to the different parameters in Definition 5.4. Finally, we also considered a linear combination of the different quality parameters, as used in the publications of Abraham et al. [ADGW13] and Luxen and Schieferdecker [LS12]. For each of these strategies, we performed an extraction of up to three alternative routes on a thousand random pairs of source and target vertices. We summarize the results in Table 12.2.

Interestingly enough, the general selection strategy does not seem to impact upon the success rates for finding a second or third alternative route. The strategy can have a large impact on the perceived quality of a path, though. In most cases, the results for a given method are consistent, offering similar results for all parameter values. Large variations are possible, though, and sometimes are in the order of close to ten percent. One might say, though, that any reasonable strategy does not change the quality in an unreasonable way. Focusing on the optimization of a single value can result in a loss of quality for later alternatives; the overlap with selected routes may prevent the discovery of other high-quality routes. Optimizing for a good BS for example yields a worse result on the third alternative than the optimization for limited sharing.

Table 12.2: *Success rates and parameter quality over different selection strategies. The strategies order candidates by a single parameter or a linear combination ($\mathcal{F}$, equally weighting all three) of them. Depicted are the values for up to three alternative routes. The quality values of ML are inconsistent with the provided success rates, as Luxen and Schieferdecker fail to provide a quality evaluation for their best success rates. The success rates for which the specified quality values are presented in [LS12] are considerably lower (*81.2, 51.2, *and* 25.0 *percent). For the corridors experiments, we used a six-turn corridor and a three-recursive detour corridor with an initial detour of twenty seconds ($\delta_t = 200$). The quality values for 3-CHV and 3-REV are only specified for the first alternative. The ones for the second and third alternatives are taken from the non-relaxed versions.*

method	S	first				second				third			
		rate	α	β	ϵ	rate	α	β	ϵ	rate	α	β	ϵ
CRP-π	α	**96.5**	**30.6**	30.3	18.4	**90.2**	35.7	30.5	14.1	**80.0**	37.7	31.6	11.2
	β	96.5	50.1	50.6	11.4	90.2	51.8	47.7	11.5	80.0	42.4	32.7	12.5
	ϵ	96.5	52.0	61.0	6.7	90.2	44.8	39.5	10.9	80.0	38.3	26.5	12.4
	$\mathcal{F}$	96.5	43.4	55.8	8.0	90.2	45.4	42.5	11.4	80.0	39.4	28.6	12.6
CRP-∞	α	95.2	44.6	58.7	17.4	85.8	50.2	57.2	13.1	69.6	46.2	49.4	11.3
	β	95.2	59.9	83.0	9.2	85.8	55.2	65.4	10.7	69.6	41.1	43.7	12.4
	ϵ	92.5	56.8	82.3	5.8	85.8	50.6	58.6	10.8	69.6	42.0	42.1	14.6
	$\mathcal{F}$	92.5	52.6	83.6	6.5	85.8	52.1	61.3	12.4	69.6	43.4	43.2	12.0
HiDAR	α	92.4	37.5	61.7	12.2	80.1	40.9	56.6	9.8	66.1	39.2	49.6	7.9
	β	92.4	55.1	**85.8**	7.0	80.1	48.1	66.5	9.2	66.1	37.9	47.1	8.5
	ϵ	92.4	53.5	81.7	5.8	80.1	44.1	58.8	8.1	66.1	37.0	46.7	8.3
	$\mathcal{F}$	92.4	48.5	82.5	6.3	80.1	43.3	60.1	8.2	66.1	37.0	46.0	8.4
turn	α	78.9	42.8	50.9	11.6	58.3	36.2	40.0	8.8	42.2	28.5	30.0	6.2
	β	78.9	50.1	69.2	6.5	58.3	37.1	4.3	9.3	42.2	26.2	24.5	8.1
	ϵ	78.9	48.9	67.0	6.4	58.3	35.3	3.9	8.4	42.2	27.4	26.5	6.2
	$\mathcal{F}$	78.9	47.3	68.0	6.4	58.3	36.3	41.0	8.6	42.2	26.9	25.6	6.1
detour	α	51.0	29.7	38.1	5.2	28.6	18.3	22.4	2.5	14.9	10.0	11.4	**1.1**
	β	51.0	32.2	46.2	2.4	28.6	17.9	20.3	5.0	14.9	9.2	8.9	1.4
	ϵ	51.0	31.6	44.8	**2.4**	28.6	**17.8**	20.6	**2.1**	14.9	**9.5**	9.6	3.7
	$\mathcal{F}$	51.0	31.0	45.6	2.4	28.6	18.1	20.3	4.5	14.9	9.5	9.3	1.5
X-BDV	$\mathcal{F}$	94.5	47.2	73.1	9.4	80.6	62.4	71.8	11.8	59.6	41.2	68.7	13.2
3-CHV	$\mathcal{F}$	90.7	45.4	67.7	11.5	70.1	55.3	77.6	10.8	53.3	59.3	79.0	12.0
3-REV	$\mathcal{F}$	94.2	46.7	71.9	9.5	80.6	60.3	71.3	12.2	61.0	66.6	74.9	12.8
ML*	$\mathcal{F}$	90.0	48.6	75.8	9.9	70.4	57.0	**80.4**	10.7	44.2	59.8	**82.6**	10.7

12.2 Quality Analysis

Next to the values for random queries, we also present information with regard to different Dijkstra ranks. This analysis discusses the quality of the different approaches for different query types.

12.2.1 Alternative Graph Parameters

In addition to the considerations we already presented within the chapters, we now compare the different approaches with regard to their respective quality parameters. We omit the values for average distance, as they show very little variation. The values reach from 1.02 to 1.06, except for the corridors, which reach 1.1 for medium-ranged queries. We depict values for total distance ($\mathcal{D}_{tot}$) and the number of decision-arcs in Figure 12.1. Except for the corridors, which offer a very large number of vertices and arcs, HiDAR offers the best $\mathcal{D}_{tot}$. We might extract alternative routes from the corridor to get reasonable values for $\mathcal{D}_{tot}$. The next section shows that the corridors do not offer better values than the other techniques, though. They do reach these values, but only by ignoring the limit on decision-arcs, though. CRP-∞ can compete with CRP-π for most Dijkstra ranks, making it a viable replacement for low-range to mid-range queries. For ranks 2^{21} through 2^{24}, $\mathcal{D}_{tot}$ remains far lower than with CRP-π. Given a similar number of decision-arcs, we can see that CRP-∞ finds shorter detours.

12.2.2 Route Quality

Next to the alternative graph parameters, we study the quality of extracted routes as well. Here, we present the success rate for the first alternative route as well as its quality. The selection considers the first viable route according to a strategy that selects the longest plateau first.

Success Rate. In terms of success rate, most methods show a very similar behavior. Figure 12.2 shows all but the corridor methods to offer close to the same success rate for all Dijkstra ranks. For medium-ranged queries, the via-vertex approach is a bit less good than the other approaches. It surpasses the corridors starting at Dijkstra rank 2^{18} . As we stated in Chapter 6, this rank roughly represents queries of three hundred kilometers in length. In general, the penalty-based approaches offer the best success rates. They also require the largest computational effort, though.

The number of different choices that are available as a first alternative routes offers a more distinct picture. For short-ranged and mid-ranged queries, the general picture remains the same. For long-ranged queries, however, HiDAR now surpasses CRP-∞. Especially the corridors methods offer only a few possible alternatives. Still, the most costly method offers the best results. CRP-πs offers up to two additional choices for Dijkstra rank 2^{24} .

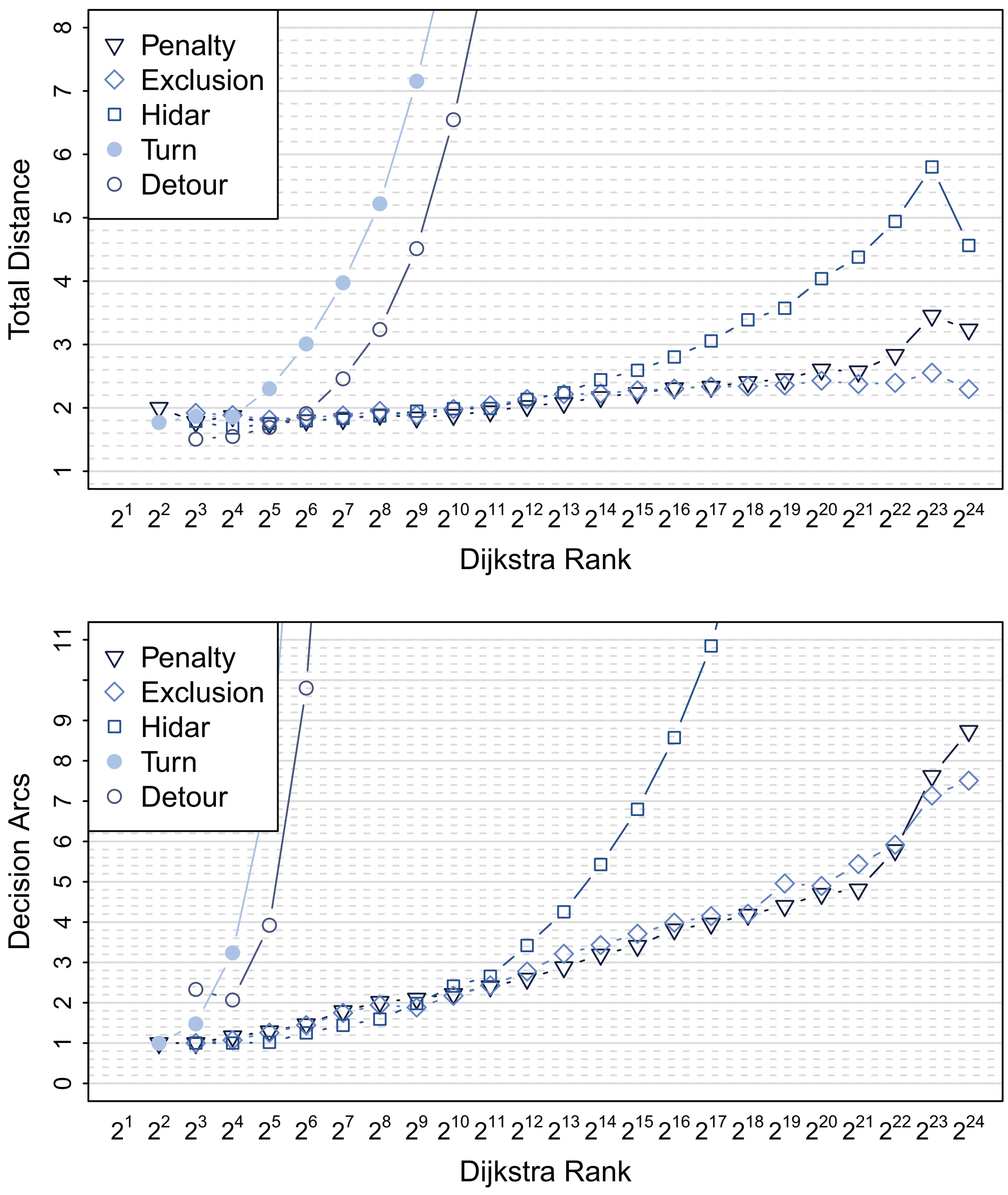
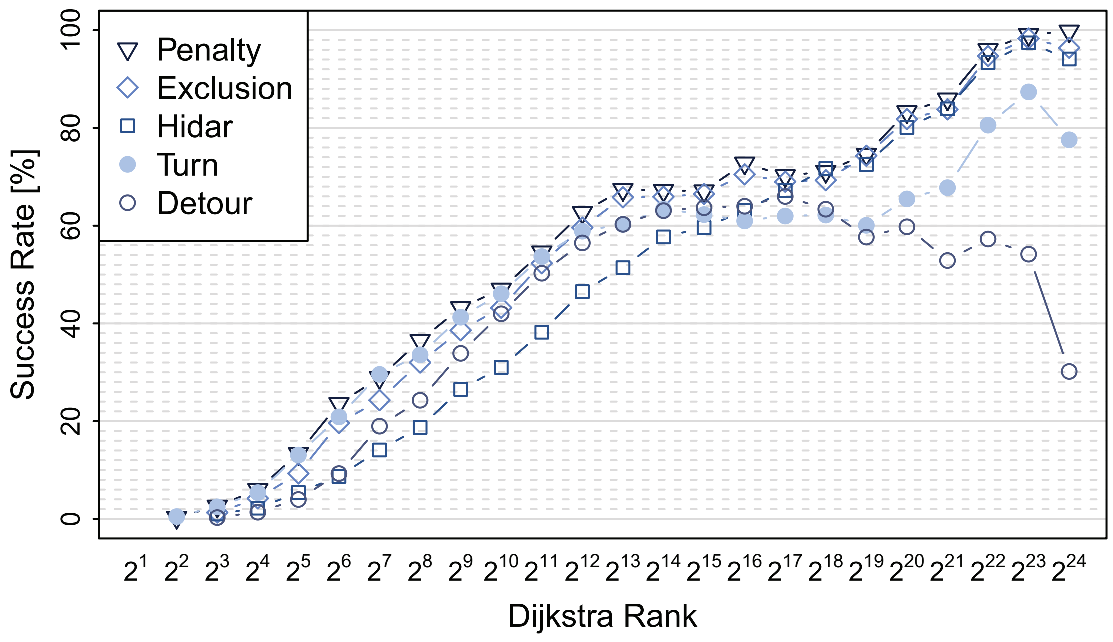


Figure 12.1: *Alternative Graph Measures*



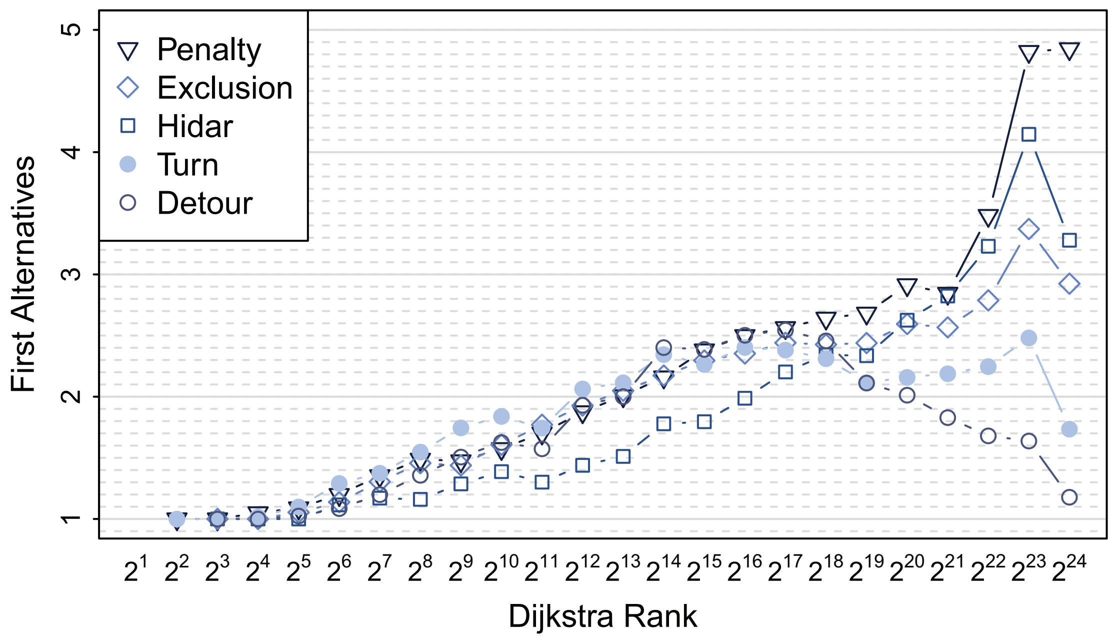


Figure 12.2: *Single Routes*

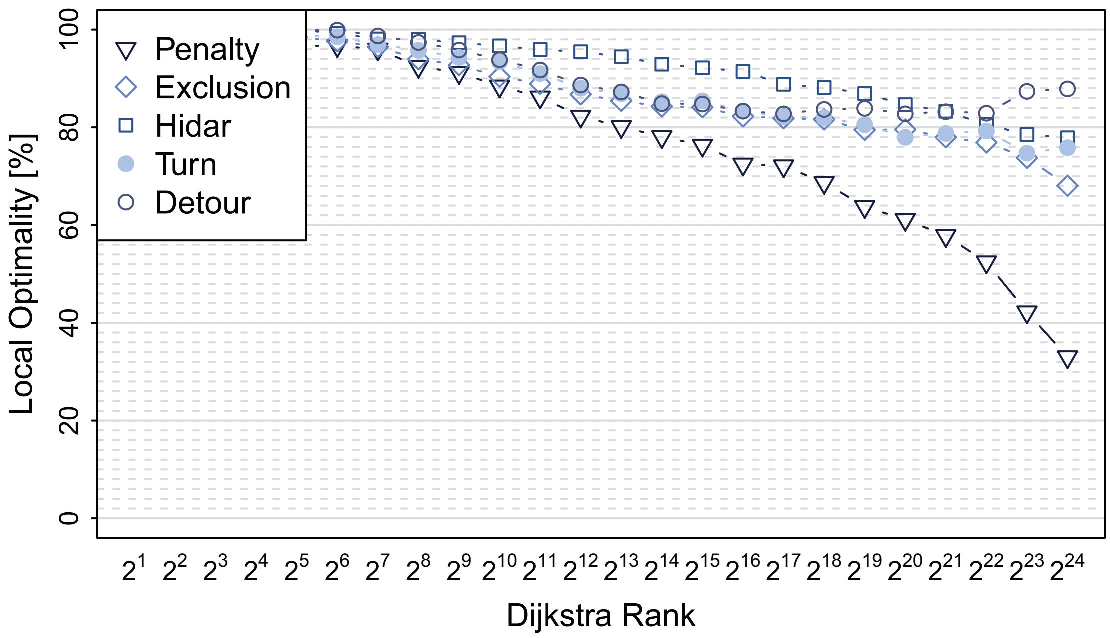


Figure 12.3: *Quality according to Definition 5.3.*

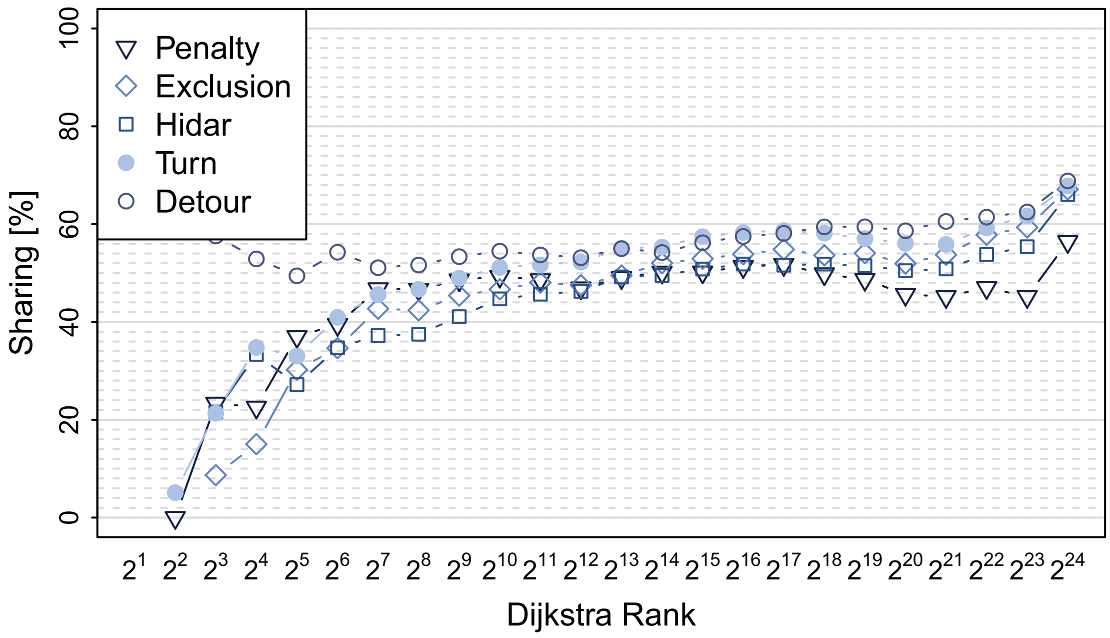


Figure 12.4: *Quality according to Definition 5.3.*

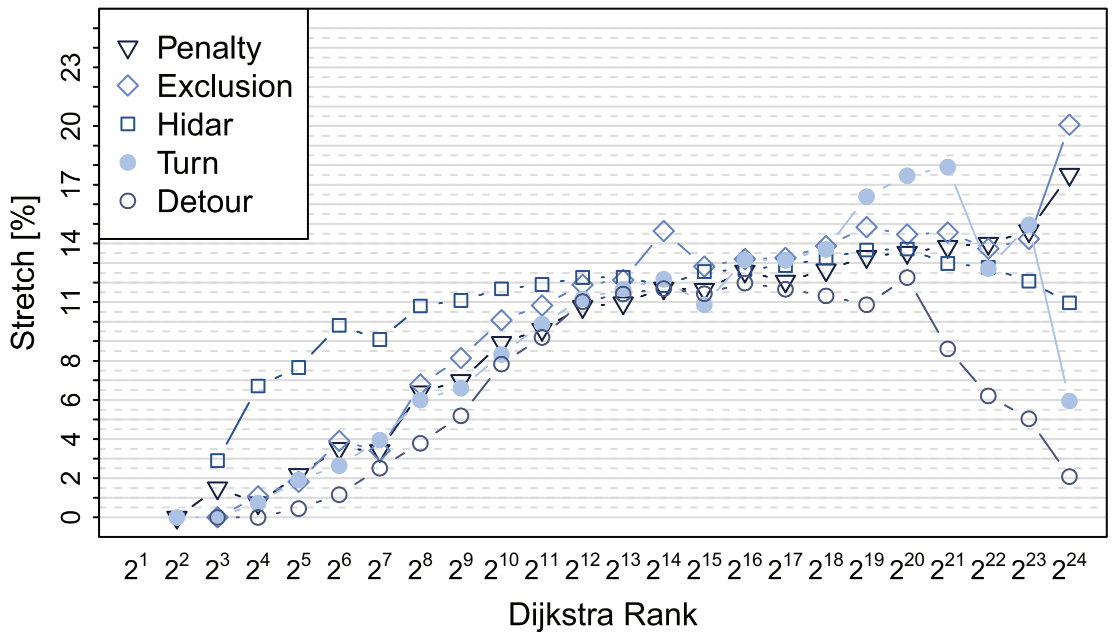


Figure 12.5: *Quality according to Definition 5.3.*

Stretch. Similar to the values in Table 12.2, the values for bounded stretch (BS) are comparable for most of the Dijkstra ranks. For long-range queries, penalization methods offer larger BS, though.

Local Optimality. Optimizing for long plateaux means essentially optimizing for local optimality (LO). As a result, we see the largest differences between our methods in Figure 12.3. Penalization tends to find paths that offer far lower values for LO. CRP-∞ is closer to HiDAR, though, than the classic form of penalization. This indicates routes calculated by CRP-π might try to avoid the shortest path too strongly.

Sharing. On average, most techniques offer very little variation in terms of sharing. This depends on the selection strategy, though, as we can see when comparing to the values in Table 12.2. The wide range of possible strategies make it impossible to give a complete overview.

12.3 Conclusion

We have discussed five different methods of computing alternative routes for static graphs. These methods offer an interesting selection of possible routes that can be calculated using speed-up techniques or parallel processing. As with the general underlying problem itself, none of the techniques can be claimed as superior in all aspects. While corridors seem to be reasonable with respect to their initial goal, the other techniques in form of CRP-π, CRP-∞, and HiDAR offer their own benefits. All of them can be considered as alternative algorithms to compute alternative routes.

Our algorithms offer very good success rates and can also be used to discover routes of higher quality. For commercial systems, there exist a lot of topics for further studies. All CH-based methods offer a problem in terms of consistency. Whenever the preprocessing is redone, the order of contraction may change. As a direct result, the discovery of via-vertex may change and offer non-deterministic results.

A penalization approach, as in CRP-π or CRP-∞ – a variant that may be possible to implement on a CH as well – can alleviate this problem. A via-vertex, given the restrictions usually imposed, is arguably a viable choice for an alternative route, though. The same can, however, not be said as easily for a penalization approach. Any small segment on a penalized path can, if it is unreasonable, result in unsatisfied customers.

In addition, all techniques offer less possibilities than we would expect in general. Especially the success rates for low-range and mid-range queries should be increased to better cover the most common queries. Possibly, a combination of several of the proposed techniques might be useful. It is doubtful, however, that a multi-technique set-up is sustainable in a commercial system. In addition, the idea of what might constitute a good alternative route might have to be reconsidered. Current quality criteria are mostly motivated by the techniques they were introduced with first.

In terms of running times, we cannot compete with the currently fastest approach of Luxen and Schieferdecker. Taking preprocessing overhead into account, we simply offer a different trade-off.

Using more time in the query, penalization as done in CRP-π and CRP-∞ offers an interesting addition to the field of alternative routes. Our implementation is the first one to allow for interactive queries on continental networks. The improved success rates over the full plateau method (X-BDV) indicate that more versions of reasonable alternative paths exist than the classical via-vertex approach might make think. A possible example to consider might be to search for multi-hop routes: instead of computing the concatenation of two paths, we might consider a list of intermediate vertices.

Part III

Further Studies

Chapter 13

13 On Stochastic On-Time Arrival

In the presence of uncertainty, optimal routing requires a good strategy. This chapter discusses possible further applications for alternative route techniques in an advanced scenario. The implementation presented in this paper is supported by a wrapper for the `fftw` library, written by Dennis Schieferdecker. The library is used to speed-up the convolutions required in the following setting. The code for the stochastic on-time arrival problem (SOTA) algorithm is a reimplementation of my own, written during fruitful discussion with Samitha Samaranayake.

The most typical metric used in the SPP is the travel time metric, which considers the average expected travel time along a path. This metric should, normally, provide the fastest means of traveling from a source to a target. But Metrics only considering a single value offer only a rather simplified view of the real world, as this standard assumption of a static road network does not represent known dynamic influences. A typical example of such a parameter can be found in the recurrent delays during the rush hour that result in slow-moving traffic or even traffic jams on many important road segments.

In general, we can distinguish between two types of delays that occur in road networks: regular delays and irregular delays. Regular delays can be identified, to a certain degree, via a series of measurements; remember Figure 6.2 for a comparison. At certain times of the day, heavily used roads are crowded due to regularly occurring demand like the commute to and from work; as a result, we can find re-emerging patterns over the course of a week. These patterns manifest themselves in slower travel times for certain road segments during the rush hour in comparison to the ones experienced during free flowing-traffic. These types of changes are subject to the research done on time-dependent route planning. For a thorough discussion of the subject, we refer to [Bat14].

We know of only a single study that discusses the merits of considering times of high or low traffic density in navigation systems [DBKS10]. The improvements claimed by the authors seem a bit high in our eyes, though, and seem to result in a comparison of routes in the time-dependent model against routes calculated for free-flowing traffic. A consideration of the on average expected travel time would probably reduce the improvements gained by the consideration of time-dependence severely. We see a similar effect in our algorithms that show significantly different results if we consider free-flowing traffic instead of expected travel times. In addition, the model used in the study seems to contain simulated data and lacks verification in form of an actual user-study. Besides this study, no further discussions of the merits of time-dependent route planning are known to us.

In the second type of delays, the research community considers a form of irregular delays that cannot be pinpointed to a specific recurring pattern. These irregular delays can result from a car in a narrow road during a parallel parking maneuver, a garbage truck or some other delivery vehicle blocking a road. In addition, many other forms of unpredictable behavior that can affect the road network are possible. These uncertainties can also be used to model effects of traffic lights, stop signs or any other traffic guidance method that can result in a stop. All of these potentially long delays can be expressed as a probabilistic delay, associated with the road segments.

Handling these stochastic delays is a difficult task, though, and an exact computation might not always be required.

Our final contribution is split into two sections. In the upcoming section, we describe the general problem of the stochastic on-time arrival problem problem. We also introduce the algorithms currently used to solve it.

Afterwards, we discuss our own contribution: We postulate that even complex problems in road networks often reduce to a set of meaningful alternative routes. For example, attached components next to a road will never contribute to a viable routing strategy. By reducing the algorithms for difficult problems to work on a smaller sub-graph, we can significantly impact the performance of these algorithms. Even though this method lacks any approximation guarantees, we show in an experimental study that they offer viable approximations nonetheless. We present these results in the context of an algorithmic study, without introducing tailored algorithms.

13.1 Problem Definition and Baseline Algorithm

The literature mainly distinguishes between two different ways of considering unreliable travel times: shortest path with on-time arrival reliability (SPOTAR) and the SOTA[1], the latter being the main topic of interest in this chapter. Both techniques operate on a limited travel time budget. SPOTAR is targeted at an optimal *a priori* path that maximizes the probability of on-time arrival. A solution to this problem is due

[1]Different settings considering reliability of the computed paths exist. Examples of different approaches are [CBWB06, CBB07].

to Nie and Wu [NW09] who consider the first-order stochastic dominance of paths. The optimal solution is not viable due to potentially exponential running times in the worst-case; Nie and Wu present a pseudo-polynomial algorithm for an approximate solution that performs well in practice. For a restricted version that considers Gaussian distributions, Nikolova et al. [NKBM06] show how to find a solution in $\mathcal{O}\left(n^{\Theta(\log n)}\right)$. We discuss the second approach in the following sections.

13.1.1 The SOTA problem

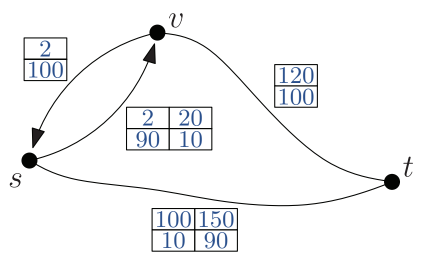
The stochastic on-time arrival problem problem aims at the calculation of an on-line strategy to guide a driver, rather than the calculation of a single a priori path. Such a strategy denotes a next vertex for every budget that the respective vertex can be reached with, given a fixed starting budget. This allows for a dynamic adjustment of routing directions, depending on previously experienced travel times. The resulting (expected) success rate of a driver following the SOTA strategy is guaranteed to be equal to or higher the probability of SPOTAR.

Formally, the SOTA problem is defined as follows:

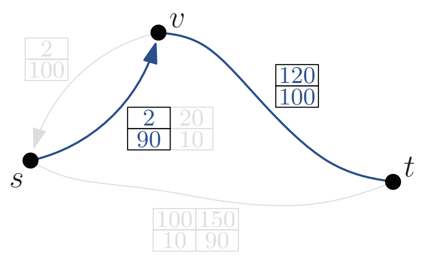
Definition 13.1 (stochastic on-time arrival problem (SOTA) problem)**.** *We consider a graph $G(V, A)$ and for every arc $a \in A$ a cost-function $\mu_a(\cdot)$ in the form of a probability density distribution; the distributions themselves are considered independent of one another. Given a time budget T, a source s and a target t, find an optimal* routing policy *that maximizes the probability of reaching t within the time budget T. Such a routing strategy determines the optimal next vertex to travel to, based on the remaining time budget due to the previously realized travel time. Let $p_{v,t}[T]$ denote the probability of reaching t from v within the budget T. A solution to the optimal routing policy is described in the equations, given $0 \leq \tau \leq T$:*

$$\begin{aligned}
p_{v,t}[\tau] &= \max_{a=(v,w)\in A} \int_0^\tau \mu_a(\omega)\, p_{v,t}[\tau-\omega]\, d\omega, \forall u \in V \setminus \{t\} && (13.1)\\
p_{t,t}[\tau] &= 1\\
succ_{v,t}(\tau) &= \arg\max_{(v,w)\in A} \int_0^\tau \mu_a(\omega)\, p_{v,t}[\tau-\omega]\, d\omega, \forall u \in V \setminus \{t\} && (13.2)
\end{aligned}$$

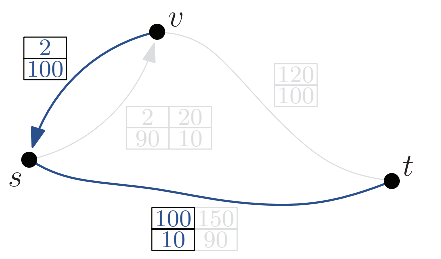
Fan and Nie [FN06] formulate the SOTA problem as a dynamic programming problem, solving it with a Successive Approximation algorithm. Their algorithm cannot be bounded in a finite number of steps, though, when the network contains cycles [SBB12a]. In the SOTA setting, such a cycle can be part of an optimal strategy. An example for such a case is presented in Figure 13.1. Figuratively speaking, in a solution to the SOTA problem we compute a SPOTAR path in advance for any remaining budget and every possible vertex we might end up at during a journey.



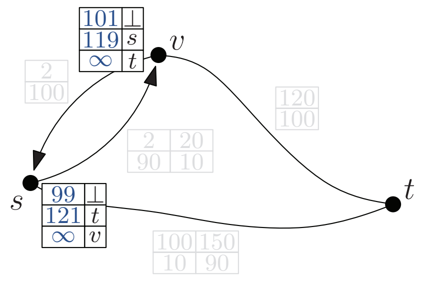
(a) *Graph*



(b) *SPOTAR*



(c) *SPOTAR at v, rem. budget 110*



(d) *SOTA strategy*

Figure 13.1: *A small example that contains a cycle in the optimal strategy. Let the budget for the travel be 130. Arc weights are specified in the boxes, each stack giving a time and the probability of experiencing this time in percent. The SPOTAR paths (13.1b, 13.1c) show the contributing parts of the respective arcs. For the SOTA strategy, we give the next vertex in dependence of the remaining budget.*

To deal with such cycles, Nie and Fan propose a discrete approximation algorithm that converges in a finite number of steps and is running in pseudo-polynomial time [NF06]. Under the reasonable assumption of a strictly positive minimum weight over all arcs ($\mu(a \in A) > 0$), Samaranayake et al. [SBB12a] present a number of optimization techniques like a label-setting algorithm and advanced convolution methods centered on the idea of Fast Fourier-Transformations(FFTs) by zero-delay convolutions [SBB12b].

The structure of the problem prevents the direct application of the techniques discussed in Chapter 4; a main reason for this is the inclusion of loops in an optimal strategy. Nevertheless, some form of pre-processing is still possible. Sabran et al. [SSB14] manage to employ versions of reach and ArcFlags to speed-up the computation. The success of the techniques ranges from little improvement, speeding up the computation by a few percent, to close to an order of magnitude. These techniques require close to a day running time, though, even on very small graphs.

The difficulty of this problem is our main reason to consider it in the connection with alternative routes. Whereas a connection between a reasonable path as considered in the SPOTAR model and a set of meaningful alternative routes seems plausible, the same cannot be said immediately for the connection with an online SOTA strategy. The main difference between a path in an alternative graph and the SOTA strategy is the inclusion of cycles in the latter. For a viable alternative path, we postulate a simple path by employing the criterion of local optimality. In SOTA, the optimal strategy depends on the possibility to turn around at any given location. As a result, the two approaches offer some considerable structural differences. Nevertheless, our study shows that alternative route techniques can offer a viable approximation choice for such a difficult problem.

13.1.2 Baseline Algorithm

Before coming to the discussion of our main topic, i.e. in how far alternative route techniques can be applied to the SOTA problem, we introduce the baseline algorithm and its main principles.

Discretization As the basic problem can be seen as a continuous problem, it does not seem reasonable to consider higher resolutions than the resolution considered in static route planning, especially, as the probabilistic model itself can only be seen as an estimate. A fully continuous interpretation of the problem is not beneficial, as we are limited in the resolution anyhow. We denote the discrete interpretation of the continuous functions in the use of square brackets ($[\cdot]$) over parenthesis in the formulas.

The optimal order algorithm. As we have already mentioned, the equations (13.2) can be solved using Successive Approximation [FN06]. Samaranayake et al. [SBB12a] present an algorithm for finding the optimal solution to the SOTA problem in a single pass through the time-space domain. The only requirement is the previously mentioned assumption of a strictly positive lower bound for the cost of an arc. Let $u_v^k[t]$ denote the Cumulative Distribution Function (CDF) that denotes the probability of reaching the target t from v in at most k steps: Algorithm 13.1 describes the method presented in [SBB12a].

The computation of the CDF cannot be computed analytically; the form of a pointwise maximum over the adjacent arcs and their associated convolutions prevents it. We can improve on algorithm 13.1 by considering not just the minimum realizable travel time at an arc connected to a vertex but also the length of the shortest possible cycle which contains it. The proof can be found in [SSB14], but the idea behind it is rather simple: assume for any link $a = (v, w)$ that we have computed $u_w[\cdot]$ up to some budget τ_w. Due to the minimal possible travel time between v and w of $\delta_{v,w}$, we can update $u_v[\cdot]$ up to the budget of $\tau_v \leq \tau_w + \delta_{v,w}$. Now consider a shortest cycle $v = v_0, \ldots, v_k = v$ of length $\delta_v = \sum_{i=1}^{k} \delta_{v_{i-1},v_i}$ that v belongs to. Stepping along the path, we update each $u_{v_i}[\cdot]$ by a delta of δ_{v_{i-1},v_i} before arriving back at $v_k = v$. Due to the updates along the path, we update $u_v[\cdot]$ by $\delta_{v_k,v_{k-1}}$, completing the circle. The

Algorithm 13.1 Single Pass SOTA

Input: A graph $G(V, A)$, a metric in the form of probability distributions $(p[\cdot])$, a source s, a target t, a time budget T

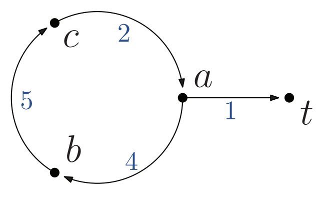
Output: An optimal routing strategy from s to t that respects T

```
1:  $\delta = \min_{a \in A} \tau_a, \tau_a = \min_\tau : p_a[\tau] > 0$        ▷ Minimal realizable travel time
2:  $k_{\max} = \lceil T/\delta \rceil$        ▷ The number of steps
3:  $u_v^0[\tau] = 0, \forall v \in V, v \neq t, \tau \in [0, T)$
4:  $u_t^0[\tau] = 0, \tau \in [0, T)$
5:  for $k \in [1, k_{\max}]$ do
6:      $\tau^k = k\delta$
7:      $u_t^k[\tau] = 0, \tau \in [0, T)$
8:      for all $v \in V, v \neq t$ do
9:          $u_v^k[\tau] = u_v^{k-1}[\tau], \tau \in \left[0, \tau^k - \delta\right]$
10:         $u_v^k[\tau] = u_v^{k-1}[\tau], \tau \in \left(\tau^k - \delta, \tau^k\right]$
11:         $u_v^k[\tau] = \max_{a=(v,w) \in A} \int_0^t p_{v,w}[\omega] u_w^{k-1}[\tau - \omega] d\omega, \tau \in \left(\tau^k - \delta, \tau^k\right]$
12:     end for
13: end for
14: return $u_i^k[\cdot]$
```

total increase of τ_i is therefore given by δ_i. The process, however, is dependent on the order in which the vertices are processed (compare Figure 13.2). Whereas the updates in topological order (see Figure 13.2c) increase the respective τ_i values by $\delta_i = 11$, updates in reverse topological order behave less well (Figure 13.2b).

To determine the best possible order to process the vertices in, Samaranayake et al. [SBB12a] discuss an algorithm related to Dijkstra's algorithm (compare Algorithm 2.2 and Algorithm 13.2). The algorithm processes the elements by a reverse pass through the time-space domain. Beginning at the source, for which the CDF has to be computed up until the target budget, the algorithm iteratively computes the most constraint vertices. The most constrained prerequisite, i.e. the vertex that is required to be known up to the largest budget, is selected next and added to the (reverse) order. For this process, the algorithm maintains a max heap that contains the most *restricted* elements; we present a pseudo-code for this approach in Algorithm 13.2.

An example: If we were to compute the best order for the graph shown in Figure 13.2a to get from b to t with a budget of 30, the order algorithm would yield $(b, 30), (c, 25), (a, 23), (b, 19), (c, 14), (a, 12), (b, 8), (c, 3), (a, 1)$. The reversal of this order yields $a \to c \to b$, the same order as processed in Figure 13.2c. The target t is omitted in this process as its complete CDF is known in advance and provides the base case.



(a) *graph*

Iter.	a	b	c
1	1	5	2
2	9	7	3
3	11	8	11

(b) $a \to b \to c$

Iter.	a	b	c
1	1	8	8
2	12	19	14
3	23	30	25

(c) $a \to c \to b$

Figure 13.2: *Updates of the τ_i values over the course of the SOTA algorithm for a small cyclic graph.*

Algorithm 13.2 Order Algorithm

Input: A graph $G(V, A)$, a metric in the form of minimal realizable travel times (μ), a source s, a target t, a time budget T

Output: An optimal order for the SOTA algorithm to process the vertices in

```
1: $\mathbb{O} = \emptyset$
2: $\mathcal{Q} = \{(s, T)\}$
3: while $\mathcal{Q} \neq \emptyset$ do
4:     $n = (v, \tau) = \max_\tau \mathcal{Q}$                    ▷ get most constraint vertex
5:     $\mathcal{Q} = \mathcal{Q} \setminus \{n\}$
6:     $\mathbb{O} = \mathbb{O} \cup \{n\}$
7:     for all $w : (v, w) \in A, w \neq t, \tau - \delta_{v,w} > 0$ do
8:         $\tau_w = \max(\mathcal{Q}.getKey(w), \tau - \delta_{v,w})$
9:         $\mathcal{Q}.updateKey(w, \tau_w)$               ▷ insert $w$ if not already contained
10:    end for
11: end while
12: return $\mathbb{O}$
```

Zero Delay Convolution. During the process of the baseline algorithm, we require the continuous evaluation of CDFs from the CDFs of neighboring vertices and the probability density functions assigned to the respective arcs. The probability density functions $p_{a \in A}[\cdot]$ are fixed and can be evaluated in advance. The former parameter, however, is continuously updated over the progress of the algorithm. To avoid unnecessary calculations, Samaranayake et al. propose the usage of so-called zero delay convolutions: a FFT-based method for calculating convolutions between a fixed variable and a continuous stream.

The method considers two vertices i and j as well as the arc $a := (i, j)$ between them in the following way: Given the CDF u_j and the probability density function $p_a[\cdot]$, the CDF u_i can be calculated as $u_i[k] = \sum_{l=1}^{k-1} u_j[l]\, p_a[k-l], \forall k$. Dean [Dea10] shows that Zero-Delay Convolutions(ZDCs) can be applied to SOTA to reduce the computational overhead at each link, due to the convolutions, from $\mathcal{O}(T^2 \log T)$ to $\mathcal{O}\left(T \log^2 T\right)$. While the straightforward approach for a convolution requires an effort quadratic in the number

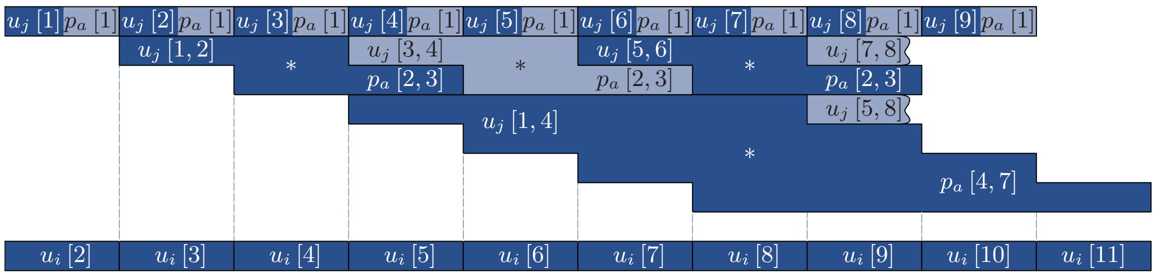

$u_j[1]$ $p_a[1]$ $u_j[2]$ $p_a[1]$ $u_j[3]$ $p_a[1]$ $u_j[4]$ $p_a[1]$ $u_j[5]$ $p_a[1]$ $u_j[6]$ $p_a[1]$ $u_j[7]$ $p_a[1]$ $u_j[8]$ $p_a[1]$ $u_j[9]$ $p_a[1]$
$u_j[1,2]$
$*$
$u_j[3,4]$
$p_a[2,3]$
$*$
$u_j[5,6]$
$p_a[2,3]$
$*$
$u_j[7,8]$
$p_a[2,3]$
$u_j[1,4]$
$u_j[5,8]$
$*$
$p_a[4,7]$
$u_i[2]$ $u_i[3]$ $u_i[4]$ $u_i[5]$ $u_i[6]$ $u_i[7]$ $u_i[8]$ $u_i[9]$ $u_i[10]$ $u_i[11]$

Figure 13.3: *Illustration of a ZDC based arc relaxation under the assumption that the vertices i and j are connected by a single arc. At the computation of $u_i[k]$, we only require access to values of $u_j[k-1]$ and other earlier values. We can extend the size of the convolution blocks when the minimum cost of an arc is larger than 1 .*

of discrete values ($\mathcal{O}(T^2)$), FFT can compute such a Discrete Fourier-Transformation (DFT) in $\mathcal{O}(T \log T)$ ($u_i[k] = (u_j[\cdot] * p_a[\cdot])[k]$). The algorithm requires access to all values, though. Given that the probability density function $p_a[\cdot]$ can be evaluated for any k, we can compute $u_i[k]$ as a sum of multiple DFTs. The basic motivation behind this approach is to utilize FFTs as much as possible.

Some values are easy to compute. For example, given $\mu(a \in A) > 0$, we find $u_i[1] = 0$ and $u_i[2] = u_j[1] \cdot p_a[1]$. For the computation of $u_i[3]$, we need to calculate $u_i[3] = u_j[2] p_a[1] + u_j[1] p_a[2]$ with the latter equal to $(u_j[1,2] * p_a[2,3])[1]$. The remaining entries at positions two and three of this DFT are part of $u_i[3]$ and $u_i[4]$ and can be used as such. As a result, we only require a single additional value ($u_j[3] p_a[1]$) to compute $u_i[4]$. The more values of u_j become available over the course of the algorithm, the larger the possible FFTs tasks become; see Figure 13.3 for an illustration. All values of $u_j[\cdot]$ required during the computation of $u_i[\cdot]$ are of a smaller index.

The metric assigned to the problem, defining a minimal realizable travel time (δ) along an arc, enables us to reduce the computational overhead even further by employing δ-multiple ZDC. The factor δ allows for larger blocks of convolutions to be handled at a time, reducing the computational cost from $\mathcal{O}\left(T \log^2 T\right)$ to $\mathcal{O}\left(T \cdot (\log^2 \frac{T}{\delta})\right)$ [SBB12b].

13.1.3 Correctness Preserving Pruning Techniques

It is rather obvious that the algorithm for the computation of an optimal SOTA strategy, even when using ZDC in combination with FFTs and the optimal order, remains a very costly process. Without sacrificing correctness, we can prune some vertices from being considered in the optimal order.

Lower Bounds. Given an optimal traversal of an arc during free-flowing traffic without any delays, the time taken for the traversal marks a lower bound. If we consider only these lower bounds during an instance of Dijkstra's algorithm, it becomes obvious that any budget below the minimal realizable travel time evaluates to zero in terms of its probability of reaching the target on time. The same can be said for vertices that do not offer a combined travel time of $\mathcal{D}(s,v)+\mathcal{D}(v,t)>T$. These do not have to be considered during the calculation, as they cannot contribute a positive value to the CDF at the source vertex.

Upper Bounds. The discretization in our interpretation allows for upper bounds as well; they are usually not considered, as we expect the maximal allowed budget to be lower than the upper bound anyways. For every distribution we find some $100-\epsilon$[2] percent quantile, giving a guarantee in some sense to traverse the arc in the given time. This is an approximation, but can be easily justified by the inaccuracy of the underlying model. The travel time that gives this $100-\epsilon$ percent chance to traverse the arc is assumed as the maximal cost of the arc. Again, utilizing Dijkstra's algorithm, we can set $u_v[\tau]=1, \forall \tau \geq \mathcal{D}(v,t)$ with respect to this maximum metric. This also enables us to directly adjust the time-space domain in case the initial budget has been chosen too large. This is technically an approximation; the measurement-based process of determining the respective probability distributions is an approximation to begin with, though, too. Given the nature of the utilized distributions, a routing algorithm would end up considering up to a day's worth of budget when it comes to traversing a city the size of Stuttgart or San Francisco. This scenario is both unrealistic and undesirable. In terms of the comparison we consider later in this chapter, this restriction only speeds up the baseline algorithm and does not benefit our considered pruning methods.

13.2 Contribution: Alternative-Route Pruning Techniques

After our description of the baseline algorithm, we present our own contributions for pruning the SOTA algorithm.

The main justification for our approach is given in Figure 13.1. As one can see, the optimal strategy given by the SOTA algorithm can be interpreted as the specification of a SPOTAR path for any remaining budget at every reachable vertex. The goal of the SPOTAR model is to find the path with the highest probability of arriving within the given time frame. In the SOTA model, we, figuratively speaking, prepare for any possible outcome and compute a SPOTAR path for every vertex we could end up at with all possible remaining budgets. We encode this in specifying the next vertex to travel to for any potentially remaining budget. At the source, we can imagine that the first vertex proposed by the strategy corresponds to the solution to the SPOTAR problem. As long as we do not experience any delay, the path should

[2]We use $\epsilon = 0.001$ to calculate the 99.999 % quantile.

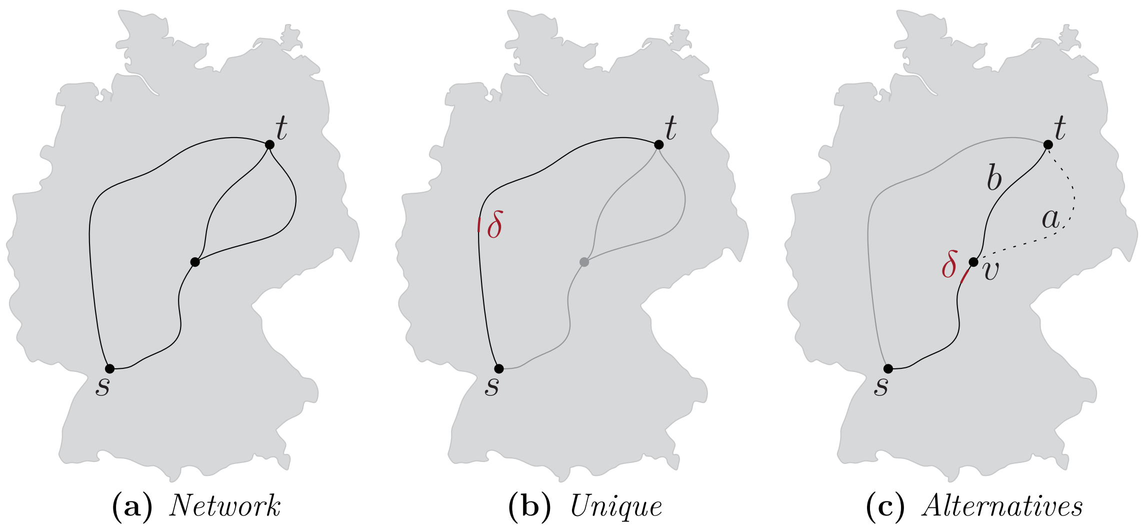


Figure 13.4: *Reasoning behind the alternative-route approach: consider a long-distance route from s to t. Apart from local adjustments, it is unlikely that we have to leave the SPOTAR path at any point. Consider the scenario of a* unique *shortest path. When traveling along it, a small delay can hardly be corrected in anything but a local fashion. In the presence of multiple possibilities, we expect a change between those only if one is slower but more reliable and the other one faster but less reliable. If we find a large enough delay (indicated by δ) at some point that results in a different path specified by the SOTA strategy, the new path (indicated by b chosen over a) cannot be fully unreasonable but rather has to be a good path as well for the rest of the journey.*

continue to offer the maximal chance of arriving on time. The moment the budget changes unexpectedly, due to some random occurrence along the way, we would have to overthrow our initial decision and rethink the directions by computing the most reliable path for the currently remaining budget. This new path, however, cannot be completely unreasonable as well. Assuming that all paths that can be considered by SPOTAR are from a set of reasonable alternatives, we can approximate SOTA with our own techniques.

If we consider a situation as depicted in Figure 13.4, we follow a path from s to t. At some point we might experience (or have accumulated) some delay. In many cases, this delay should not change the general direction of travel, at least during long-distance trips. The reasoning behind this argument is that the natural hierarchy of a road network restricts long-distance travel to a small subset of important roads. If we travel along a highway and consider static and independent probabilities, a change in the shortest path to follow is unlikely [3].

[3]We consider this a potentially oversimplified view, as the probability of a delay in close proximity to a traffic jam increases dramatically.

For the most part, we would expect a strategy to follow slower but more reliable paths, unless the budget gets too small. When the budget gets reduced below a certain size, faster and potentially unreliable paths become the only possible choice.

In our example, a potential change in directions depends on the remaining budget. As long as a reliable, but slow, path (e.g. the one labelled a in Figure 13.4c) remains within the budget, it presents the path of choice. After an experienced delay, however, the budget might not be sufficient to a travel along the initially chosen path a and the strategy presents b as the alternative path to follow. This kind of change is associated with a faster but potentially less reliable path. Completely ignoring the higher road categories at this point is not an option, as the travel time would increase significantly. In addition, roads of similar categories in close vicinity to each other should be roughly equivalent in terms of their reliability.

Similar reliability on identical road types allows to exploit the natural hierarchy of the network in some sense. In fact, our experiments show that for the most part we can restrict the search to a reasonable sub-graph without sacrificing much in terms of correctness. This approach does not offer any guarantees of correctness, though. We only present it as a potential use case for alternative-route techniques. We assume that, even in other settings, the general structure of reasonable routes does not deviate completely from shortest paths with respect to travel time metric. This idea is motivated in the fact that travel time will always be one of the major factors and cannot be ignored completely.

Pruning Model. Given the parameterization of SOTA, pruning the algorithm using the alternative graph DAGs is not viable. Instead, we prune on the basis of vertices. This enables the strategy to hop between neighboring paths and to perform u-turns. However, this model affects the potency of attached component filtering for corridors. Since our test graphs contain only cities, the density within the network and the vertex-based pruning result in next to no attached components. For long-range travels, the component filtering might offer great improvements. Unfortunately, we were not able to obtain any reasonable probability data for larger graphs. Some tests on randomly generated data show promising results, though.

13.3 Experimental Study

The main goal of this study is an experimental evaluation of possible applications for our previously implemented alternative-route techniques. Therefore, we do not present any new algorithmic approaches or techniques specifically tailored to the scenario.

We compare two different settings. The first one is the original SOTA algorithm without any application of additional pruning techniques. The second scenario considers the original SOTA method restricted to a set of preselected vertices. For the selection of these sub-graphs, we test a variation of alternative-route techniques.

Standard of Comparison. While it is rather easy to quantify the decrease in quality that we can experience using our approximation-based methods, the same cannot be said for the benefits. The decrease in quality can be measured in the CDF at the source over the full range of the budget from minimal necessary budget to guaranteed arrival. We present both average values describing the full interval and error values scaled to a range from zero to a hundred percent of the budget range.

Unlike the objective measure of probability in the CDF, the measure of running time is too dependent on the actual implementation choices made during the implementation of the SOTA algorithm. We still present some values for comparison, even though we feel that they might favor our method too strongly.

To potentially give a better impression of our results, we focus on the number of performed convolutions. While this unit of measure is not an optimal one either, it describes the workload of the algorithm and is independent of other implementation choices.

Next to the approximation of the CDF at the source, we present an additional measure. This measure describes how well an optimal a-posteriori pruning set would be covered by the respective strategy. This set, which is introduce shortly in the following section, contains all vertices that are part of an optimal SOTA strategy.

13.3.1 Pruning Techniques

For completeness, we give a short overview of the different pruning methods we evaluate in this chapter. All of these techniques are used to select a set of vertices. Since it is unclear at which positions we might want to turn back, we do not limit this possibility.

The success of the SOTA strategy is reduced significantly if we only consider paths that are directed at the target. Since our alternative graphs define DAGs, a lot of the potential benefits of a SOTA strategy would be lost.

Basic SOTA. The most basic version of the SOTA algorithm is the unpruned version. While of course employing upper and lower bound pruning, it does not use any additional pruning. This version represents our standard SOTA algorithm.

Optimal Pruning. Optimal (a-posteriori) pruning is what we call our reference algorithm for a pruned SOTA instantiation. It is a rather theoretical construct, as it requires a full SOTA query and as such cannot possibly help in speeding up its computation. The method first executes the *unpruned* version of the algorithm. In a second step, we traverse the computed SOTA strategy and extract all vertices that contribute to the CDF of the source vertex. This set of vertices is considered for another run of the SOTA algorithm.

Even though it might not seem so at first, this method could potentially be restricted further without any loss of quality. The method does not prioritize equivalent paths in any way. If the SOTA strategy contains references to multiple choices of the same quality, all of them are considered in the pruned version as well. It can be seen rather easily that this restriction does not lower the quality of the found CDF at the source. By eliminating vertices that did not contribute to the final result in the first place, we only eliminate an unnecessary workload.

The acquired set of vertices provides an idea of the actually workload required to compute an optimal strategy. The vertices found this way are also used to measure how well the other techniques replicate this information.

Corridor Pruning. As a first method of pruning, we employ unmodified versions of our corridor algorithms. All vertices within the corridor are selected as pruning vertices. The amount of vertices is, however, far larger than suitable. Our different methods offer a vast choice for configuration, though. For a tailored method, the actual distributions at the different arcs could be used to restrict necessary deviations. We could create a corridor that limits deviations to parts of the graph in which they are likely. In addition, we could consider more deviations when we are in closer to the target, as the expected delay increases over the time of travel.

The dense corridors contain a far larger amount of circles than the other techniques. Therefore, this method should also reduce the amount of computation less than what we expect from other techniques. We consider multiple-turn versions of our corridors and the recursive detour variant. The graphs tested offer only little potential to reduce the standard corridor in its size; due to the density of the graph, a corridor contains barely any attached trees. We would expect these components to be of greater relevance in long-range queries, though.

Alternative Route and Graph Techniques. For the alternative-route/graph-based pruning techniques, we consider the full set of possible choices. We extract every route that might contribute a possible alternative path. The combination of these different paths forms our pruning set.

For the via-node-based methods, we consider the reference technique X-BDV to avoid any artifacts due to the underlying CH. Instead of only selecting a set of different paths, we consider the full set of plateaux, just as done in HiDAR. This way we do not only consider all potentially viable alternative routes, but also smaller deviations that would not make a reasonable alternative on their own. We discard all paths, though, that do not reach a plateau that would be long enough to pass the test for local optimality.

In the penalty-based approach, we use the upper-bound graphs that we introduced in Section 8.2.2. These graphs consider all paths found during the different penalization iterations, ignoring the viability of segments and decision arcs. We limit the iterations by our usual criteria, though, to keep the processing overhead manageable, performing about ten penalization iterations.

Table 13.1: *Computational overhead of the SOTA algorithm in comparison to a perfect pruning strategy.*

	unpruned		perfect pruning		reduction factor	
graph	steps	vertices	steps	vertices	steps	vertices
Stuttgart	11 302 276	48 462	74 822	318	0.01	0.007
San Francisco	111 348	4 166	1 449	72	0.04	0.045

13.4 Evaluation

First, we take a look at the optimal pruning method. When we compare the workload for the general SOTA algorithm with the a-posteriori optimized version, we can see that most of the work performed does not yield any improvements. Only a fraction of the actually processed vertices contribute to the final CDF at the source. Most of the convolutions that SOTA performs can be pruned. Compare Table 13.1 and Figure 13.5 in this regard.

The data shows that the results follow a similar pattern, no matter whether we are considering measured data (San Francisco) or simulated data (Stuttgart). The optimal strategy is only concerned with a minor fraction of the initially generated prune-space and calculation order.

In our visualization of a perfectly pruned search, the motivation behind our idea to use alternative-route techniques becomes apparent. Along a shortest path, some additional arcs may provide a choice for some budget values. Most of the vertices in the search space of the SOTA algorithm do not contribute to the result, though. Even though they could theoretically offer a benefit, the mostly uniform behavior we expect in a road network manifests itself in only minor detours close to the shortest path. Only in a few cases, the search departs from the shortest path and follows another path. The different path resembles a viable alternative route, though.

Turning to Figure 13.5, we can spot a close resemblance to the previously introduced corridors. In comparison to Figure 11.6, we can instantly make out a lot of similarities[4]. In the following paragraphs, we compare the results we get from executing different alternative techniques to prune the SOTA algorithm with the optimal algorithm.

Table 13.2 indicates the best approximation for our corridor variants. This is also confirmed in Figure 13.6. The corridors offer very small average and maximal errors with a low variance. The direct alternative variants, offer a good approximation on average. In spite of the good average quality, the worst case results are significant, though. If we compare the penalty method to the via-vertex approach, we get a very large difference in the number of vertices which we have to consider. However, for the

[4]These images are, of course, selected to show a large similarity. The images not showing such a characteristic mostly contain a shortest path, though, too.

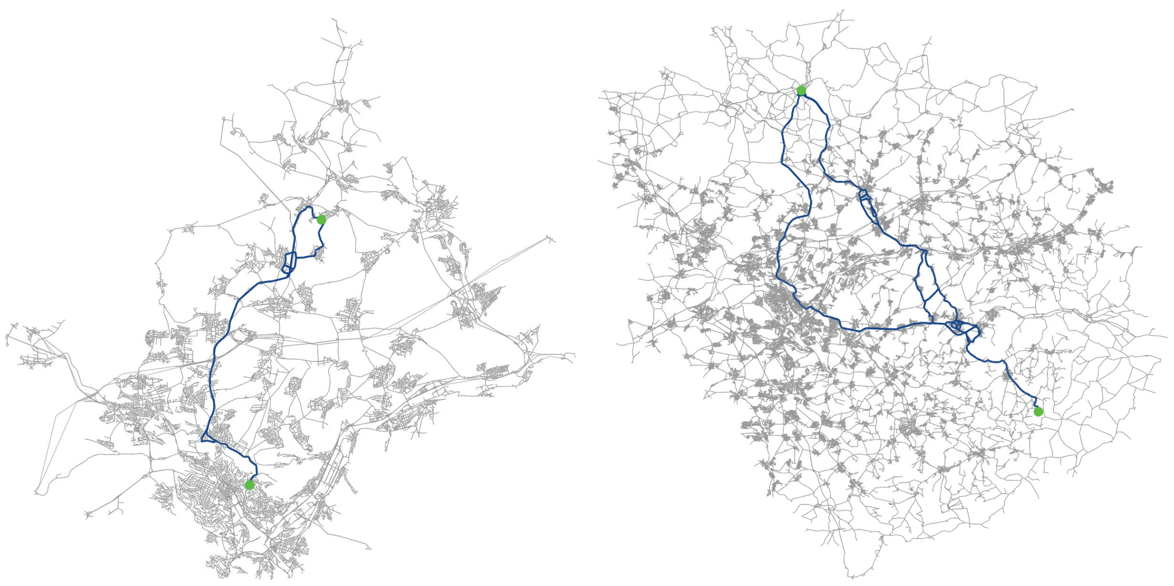


Figure 13.5: *Visualization of a perfect prune space in relation to the prune space of the basic SOTA algorithm. The visualized networks mark the prune space of the basic SOTA algorithm. The highlighted part describes the subset that offers (some) positive results for the chosen budget. This set defines what we refer to as an optimal prune space.*

via-vertex method, the increased number of vertices barely results in lower failure rates. The most expensive pruning method, the recursive detour corridor, offers a very low maximum failure rate and, even though far from being the perfect strategy, requires about a tenth of the convolutions required for the baseline algorithm.

In all tested versions of distributions, we find the same results. The corridor methods offer a close to perfect set-up, producing errors that are well within reasonable limits in the presence of uncertain data. At the same time, our methods greatly reduce the number of required convolutions. The effect also directly translates into improved running times (see Table 13.3).

The combination of both low error rates and improved running times indicates that alternative routes might offer a good approximation method for difficult problems.

Error Distribution The deviation in the CDF at the source shows some variation, depending on the used strategy. We present Figure 13.6 as a visual aid. As Table 13.2 already indicates, most methods show only little variation in the CDF on the San Francisco network. The classical alternative approaches show a rather large uniform error, though. On the Stuttgart network, we can see a more general deviation with a lesser error. The deviations tend to be worse for the higher budgets. Judging by the steps in the different settings, we expect that we could consider different metrics besides the expected travel time for further improvement. Finding reasonable routes

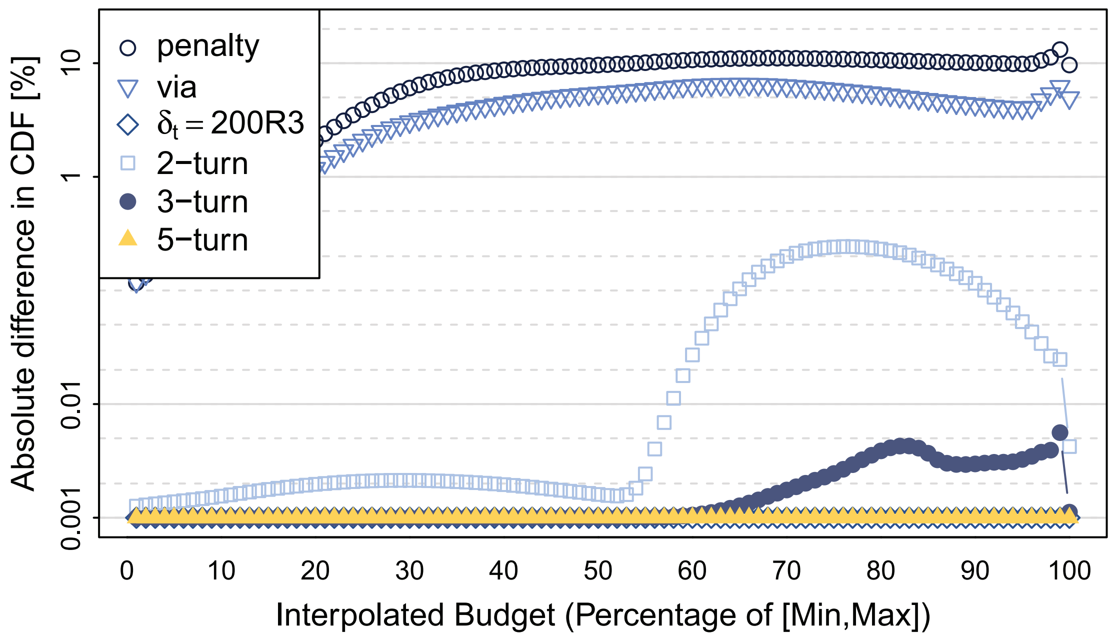
Table 13.2: *Pruning Results for different techniques on the Stuttgart and the San Francisco network. The results are nearly identical to the random inputs generated. The error specifies the difference to the CDF calculated by the full SOTA algorithm, restricted to the interval between the last occurrence of zero probability to the first occurrence of a guaranteed arrival.*

					error		
network	method	steps	prune-space	coverage	mean	std	max
Stuttgart	none	11 302 276.16	48 462.05	100.00	0	0	0
	perfect	74 822.17	318.55	100.00	0	0	0
	penalty	34 286.40	188.92	71.88	0.0039	0.0077	0.0452
	via	390 473.66	5 658.27	72.30	0.0053	0.0099	0.0574
	turn-2	208 070.15	872.40	88.06	0.0019	0.0043	0.0285
	turn-3	368 225.02	1 491.62	92.24	0.0012	0.0033	0.0237
	turn-5	843 541.14	3 354.27	96.28	0.0004	0.0016	0.0150
	dt-200-3	1 520 196.55	5 469.49	98.32	0.0001	0.0003	0.0020
San Francisco	none	111 348.84	1 449.19	100.00	0	0	0
	perfect	4 166.02	72.02	100.00	0	0	0
	penalty	1 786.09	36.15	64.13	0.0683	0.0528	0.2843
	via	31 289.88	1 233.53	83.17	0.0312	0.0305	0.1473
	turn-2	12 339.19	218.38	97.47	0.0003	0.0006	0.0025
	turn-3	18 567.09	335.30	99.39	0	0	0
	turn-5	30 393.01	573.16	99.99	0	0	0
	dt-200-3	50 551.61	1 209.74	100.00	0	0	0

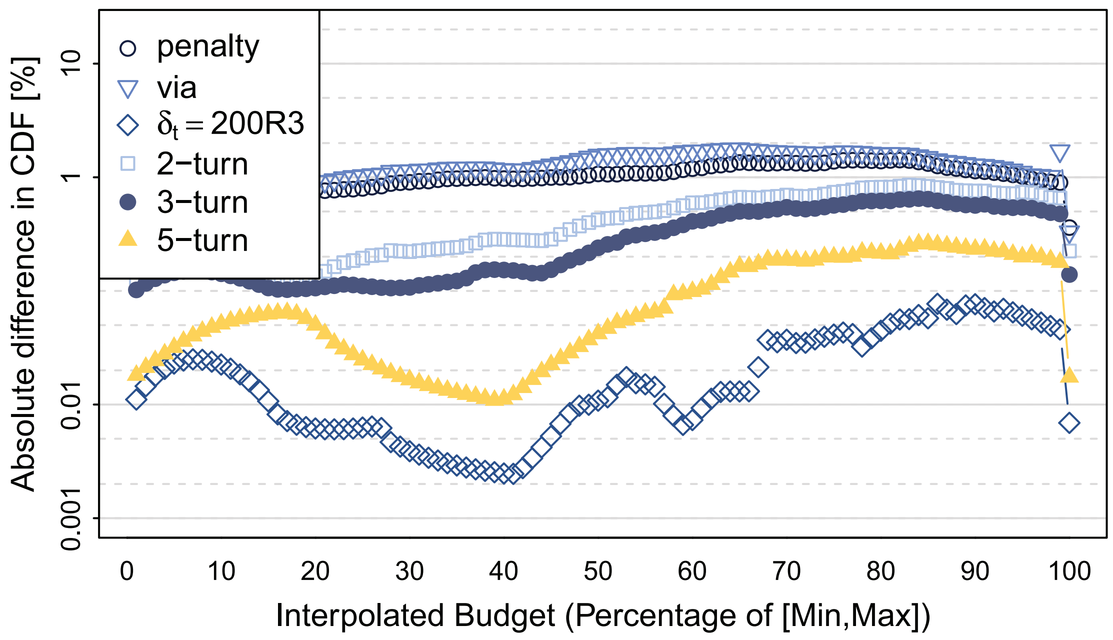
for the maximal required budget should contribute to reducing the loss in quality at the upper end of the budget range. The errors we find are more than reasonable, though, given the inaccuracy of the model to start with.

13.5 Conclusion

Even though we cannot give any approximation guarantee with the presented techniques, we have presented evidence that alternative-route techniques might be used to approximate difficult problems. When we present navigational choices to a human, travel time will most likely be among the most prominent features to consider. The information gained by applying our techniques to other problems can be used to further improve upon alternative-route techniques which, in turn, can be used to get better approximations for difficult algorithmic problems. The reasoning behind this study lies within our assumption that travel time, which is the metric used for most shortest-path algorithms, dominates many other influences. Given a metric that offers a good approximation for the average traversal of arcs, most routes that can be found



(a) *San Francisco*



(b) *Stuttgart*

Figure 13.6: *Error variation in the CDF at the source over the range between the minimal and the maximal required budget.*

Table 13.3: *Running times of our SOTA implementation on the different pruning strategies. Remember that these have to be treated carefully as different implementation choices might offer severely different times.*

graph	strategy	time [s]	graph	strategy	time [ms]
Stuttgart	none	240.5	San Francisco	none	692.2
	perfect	2.3		perfect	25.7
	penalty	1.1		penalty	13.0
	via	15.5		via	334.1
	turn-2	5.3		turn-2	89.0
	turn-3	8.8		turn-3	141.0
	turn-5	18.4		turn-5	248.6
	dt-200-3	27.1		dt-200-3	493.3

using reasonable methods should, in some sense, follow the set of reasonable paths we can find in this basic metric.

We expect, of course, a need for some adapting to fit our methods to different purposes. Nevertheless, we are positive that our techniques can be applied to a wide range of different use-cases. For the most part, these applications will be limited to approximations without guarantee but which should nevertheless work well in practice.

Chapter 14

14 Conclusion

A few final thoughts and closing remarks.

Whereas the calculation of shortest paths in road networks is a thoroughly studied problem and offers solutions that can answer distance queries within the time required for a few random memory accesses, the same does not hold true for the calculation of alternative routes. The problem is, in itself, hard to specify as it remains an ambiguous task. Opposed to a static metric for which we can easily define optimal paths, the question of how to define an optimal alternative route is hard to answer, apart from a few rare cases. As long as we cannot find two perfectly disjoint paths of the same length, the paths most likely become incomparable and offer a full set of Pareto-optimal choices.

Alternative Route Quality. The propositions of Abraham et al. offer an intuitive set of criteria for the quality of an alternative route. Especially the bounded stretch and limited sharing present requirements that are, in our view, without a doubt absolutely justified. The travel time alone might not be the only reasonable criterion, especially as we cannot determine it to a perfect accuracy, but one can easily see that it remains an important part of any travel. The value of the criterion of limited sharing is even more obvious. Offering a tiny difference on a long journey might constitute a good path on its own, the relevance to a driver is questionable, though. This justifies the requirement of a certain amount of variance very well. The requirement for local optimality is, due to its origin in the via-vertex paradigm, a bit too specialized in our view. In some regions we can find parallel roads of different types which might each offer a preferable choice for different drivers, but one of them might not offer an optimal path between any two points on it. The considerations purely based on travel-time can also be considered problematic when – as it is the case in Berlin – there are some much faster roads that might nevertheless be considered undesirable by some. An inner city highway might offer far better travel times on average but is associated with a higher stress level. In shorter travels, the relative amount of additional time someone might consider acceptable to invest can be larger than for long journeys. When driving for ten hours, a detour of twenty-five percent in length is a considerable amount of time. Given a journey of ten minutes, one might choose a fifteen-minute one in order to avoid some uncomfortable driving.

Results. Besides considerations regarding quality, we have presented a wide range of techniques in this work that offer ways to compute a great variety of different alternative route types as well as the possibility of solving other problems and supporting other tasks. As is the case with the basic problem that we studied, it is, in our eyes, impossible to offer a single solution that presents the best method to compute alternative routes. As the routes themselves, each of the presented approaches offers its own merits and can be considered viable on its own. Whereas the corridors seem inferior to the techniques specifically designed to compute alternative routes, they shine in our experimental study of the stochastic on-time arrival problem. The parallel implementation of the bi-criteria shortest-path problem, while too slow for an interactive algorithm on continental networks, offers a tool to use in one of the many algorithms that concern themselves with bi-criteria searches. Our presented data structures that enable this parallel algorithm are even useful on their own for sets with bulk operations. One of the major benefits of our algorithms to compute alternative routes is their independence of the quality criteria in Definition 5.3. Since in all of our techniques but the bi-criteria search we compute graphs that are independent of these criteria, the selection process should be able to adjust in reasonable ways to other possible criteria one might think of. Other techniques are much more sensitive to these criteria and could suffer severely under a change of requirements. The considerations we presented offer a good selection, using multiple accepted speed-up techniques that are also used in commercial systems.

Open Questions. Next to further potential optimizations that could bring the penalty method down to below a hundred milliseconds for all tested queries, some other questions remain. The most interesting in our view is not an algorithmic question, though. In our opinion, to improve the quality of alternative routes it is most relevant to understand what constitutes a good alternative route. This would require a thorough study that is beyond the scope of this thesis, however. Nevertheless, we are positive that the results presented in this thesis offer a good way of calculating alternative routes for many possible definitions of quality, as long as the travel time remains a central component of the quality criteria[1].

[1]Of course we would be able to consider any other metric for our penalty method and all other metrics that offer a similar hierarchy in a network as travel time does.

Ω Glossary

Symbols

#arc m
Number of arcs in a graph. [10, 14, 18, 23, 54–56]

#vertex n
Number of vertices in a graph. [10, 14, 18–21, 23, 29, 37, 40, 63, 71, 85, 87, 88, 101, 196, 197, 245]

k-ary heap
A specific implementation of a priority queue, using balanced trees with a branching factor of k. [13]

A

algorithm engineering
A branch of algorithmic theory that focuses both on asymptotic guarantees and strong performance on real-world instances. It is usually described at as cycle of design, analysis, implementation, and experiments. [3, 15, 17, 22, 23, 40]

arc A
An ordered pair (u, v) of vertices. The first vertex (u) is referred to as the origin, the second (v) as the destination of the arc. In our setting, an arc represents a road segment between two intersections. [9–22, 24–30, 33–39, 41, 43–46, 48, 50–52, 54–60, 64, 65, 68–70, 73, 74, 76, 84–86, 89, 101, 103, 110, 112, 113, 117–125, 127, 129, 133–136, 141, 142, 148–150, 152–159, 161–163, 165, 166, 173, 178–182, 185, 192, 194–196, 199–201, 203, 205, 206, 208, 217, 218, 222, 226, 235, 245–251, 255, 256, 258, 263–268, 270–273, 318, 320–322]

B

B-tree
A search tree with multiple router keys per node. Its origins stem from databases that try to minimize block accesses to a HDD. [83, 86, 90, 92–112, 114, 306, 313, 316, 317]

binary heap
A specific implementation of a priority queue, footing a a balanced binary tree. Also generalized as k-ary heap. [13, 144]

border arc
An arc whose origin and destination belong two different cells of a partition. [27, 43]

border vertex
In a partition of the vertices of a graph, a border vertex is a vertex that has an adjacent arc whose destination is located in a different cell than its origin. [22, 27, 42, 44, 45, 65, 132, 134]

bounded stretch BS
Condition for the viability of alternative routes due to Abraham et al.: an alternative round may not contain segments that are much longer than the best possible distance between the endpoints of said segment. [60, 61, 78, 121–123, 142, 164, 172, 178, 180, 185, 199, 239, 261, 273, 335]

bucket heap
A specific implementation of a priority queue, using buckets in a simplified way compared to radix heap. [13, 14]

budget T
The longest time available for a journey from a source to a target. [244–254, 256–259]

C

cell $\mathcal{C}$
Element of a partition. [22, 26, 27, 41–45, 47, 48, 64, 65, 130–138, 141, 142, 174, 176, 177, 181, 195, 264, 311, 321, 322]

clique
A clique is a (sub-)graph ($G\left(V, A\right)$)that contains the arc (u, v) for all $u, v \in V$. [41, 43, 44]

contract
Contraction describes an operation that reduces the size of a graph by reducing the number of vertices in a controlled (and reverseable) way. Usually part of a heuristic to

enable more expensive algorithms to operate on smaller graphs. [28, 29, 33–35, 37, 39, 153, 159, 165, 220, 271, 272]

corridor Γ
A set of vertices with associated shortest path information directed at a common target. [4, 123, 191–206, 208, 212, 213, 216–227, 229, 232–235, 240, 253, 255–257, 262, 327, 328, 333–336]

cost μ
Synonym for the value that a metric assigns to an arc. [13, 18, 19, 26, 35, 36, 39, 46, 55, 74, 84, 110, 112, 113, 119, 132, 135, 136, 165, 172, 198, 200, 245, 247, 250, 251, 266, 269, 318, 320]

cost bound Δ
An upper bound for the value of $\mu(a \in A)$ [13, 14, 23]

cost function μ
Synonym for a metric. [9]

D

degree $\delta(\cdot)$
Measure. Describes the number of outgoing (or ingoing) arcs at a vertex. [10, 28, 40, 69, 74, 89, 97, 100, 101, 109–111, 133, 196, 197, 199, 217, 218]

destination
The second vertex of an arc. [9, 22, 24, 45, 152, 161, 218, 263, 264]

diameter D
The length of the longest shortest path within a weighted graph. [11, 41, 71]

distance $\mathcal{D}$
The length of the shortest path between two vertices in a weighted graph. [10–12, 17–19, 21, 23–31, 33–35, 37, 38, 41–44, 49, 54–62, 73, 74, 76, 78, 84, 120–125, 127–130, 134, 142–144, 150, 152, 154, 157, 158, 162–165, 167, 173–176, 178, 180, 183–185, 192–194, 196–198, 218–223, 226, 235, 251, 252, 261, 264, 266, 267, 269–271, 321, 322]

dominate
A label dominates another label if it is better or equal in every aspect considered for the label. [83–87, 89, 90, 107]

F

FIFO property
In terms of time-dependent, the FIFO property requires the metrics to assure that waiting at a vertex is never beneficial. [16]

G

goal-direction
A paradigm in which a search is augmented with a sense of direction that restricts searches to reasonable directions. [24–26, 193]

graph G
A collection of vertices and arcs. The theoretical concept of a graph is used to describe a road network by mapping intersections to vertices and road segments to arcs. [x, 5, 9–16, 18–22, 24, 25, 27–30, 33–46, 51, 54–57, 60, 62–64, 68–74, 76, 78, 85, 86, 97, 109–111, 113, 115, 117–125, 127–129, 133, 135, 138, 140, 142, 143, 147–167, 171, 173, 177, 178, 180–182, 185, 192, 194–197, 199, 203, 205, 206, 208, 212, 218–223, 226, 231, 233, 235, 239, 244–249, 253–255, 262–265, 268, 270–272, 306, 307, 309, 310, 322, 326, 327]

H

highway dimension h
A measure classifying the minimal size of a hitting set covering all shortest paths. [40, 41, 196, 197]

hitting set
A hitting set with respect to a collection of sets S_i is a set of elements s_j such that $S_i \cap s_j \neq \emptyset$. [40, 41, 266]

L

label ℓ
A label is a value or set of values assigned to a vertex. It is usually used as a distance-label and describes the length of a path from a given source to the respective vertex. [13, 14, 17, 18, 20, 25, 73, 74, 83–87, 89, 90, 95–104, 107, 109–114, 265, 266, 269–271]

label-setting
A label-setting algorithm is an implementation of the general labeling algorithm. During the execution, a label or vertex is relaxed only once. [18, 86, 107, 110, 113, 246]

leaf
A leaf is a reference to a special node in a search tree. Leaves are located at the lowest level of the tree and has no descendants. [86, 87, 92–96, 98–100, 102–104, 106, 107, 270, 316, 317]

length $\mathcal{L}$
Summed cost of the arcs of a path [11, 16–20, 34, 40, 41, 49, 50, 52, 55–57, 60, 61, 68, 72, 78, 79, 85, 102, 103, 117, 121–123, 125, 157, 162, 175, 177, 178, 180, 181, 184, 185, 187, 196, 205, 219, 261, 265, 266, 270]

level ℓ
Hierarchical data structures and algorithms are based on levels. A level in a hierarchy describes a set of elements that are equal in their height in the hierarchy. [22, 25, 27–29, 31, 38, 41–45, 47, 51, 63, 64, 93, 98, 102, 103, 130–141, 147, 150, 152–154, 176, 177, 179, 194, 195, 205, 215, 220, 222, 231, 232, 311, 312, 321–323]

LIFO
Last in first out principle. The element last added is the first one to be removed. [271]

limited sharing BS
Condition for the viability of alternative routes due to Abraham et al.: the alternative route is allowed only to share a limited amount of distance traveled with the original shortest path. [59–61, 63, 76, 78, 122–124, 167, 174, 175, 178, 180, 185, 233, 261, 275, 305, 335]

local optimality LO
Condition for the viability of alternative routes due to Abraham et al.: every sufficiently short segment of an alternative route has to be a shortest path. [60, 61, 78, 122, 133, 142, 148, 167, 185, 199, 239, 247, 255, 261, 275, 335]

M

metric μ
A metric is a function $\mu : A \mapsto \mathbb{R}$. Typical examples are travel time, distance, or economic cost required to traverse an arc. [9–11, 13, 16–21, 26–29, 34, 35, 41–43, 46, 49, 50, 52, 54, 56, 58, 59, 73, 74, 83–86, 103, 110, 114, 118–122, 124, 127, 128, 142, 152, 163, 172, 173, 192, 196, 197, 231, 243, 245, 246, 248–251, 253, 257, 258, 260–262, 265, 267–269, 271, 272, 322]

microcode
Operations describing the process of constructing a CH under the assumptions of all shortcuts to be necessary. [43, 44, 47, 48, 134]

N

node
A node is the basic element that makes up a search tree. [86–88, 92–97, 99, 100, 102, 264, 266, 269, 270]

O

Open-MP
A multi-threading library. [176]

origin
The first vertex of an arc. [9, 11, 38, 45, 152, 161, 218, 263, 264, 271]

overlay graph
An overlay to a graph ($G(V, A)$) is a graph itself, defined on a subset of the vertices V. For arcs, the overlay-graph may contain a combination of arcs from A as well as new arcs. The concept of overlay-graphs is usually used to characterize a larger graph by representing essential parts of it using a smaller graph. [10, 27–29, 33–35]

P

Pareto dominance
A solution to a problem dominates another solution if it is better or equal to the dominated solution in every aspect of the criteria. [16]

Pareto-optimal
In a multi-criteria setting, a Pareto-optimal solution describes a solution to which no other solution exists that is better in every aspect of the desired criteria. [ix, 16, 52, 72, 73, 83–91, 95–98, 100–104, 107–109, 111–115, 261, 306, 313, 318–320]

partition $\mathcal{P}$
Collection of subsets S_i to a set S with $\bigcup_i S_i = S \land S_i \cap S_j = \emptyset$. [21, 22, 26–28, 41–45, 64, 65, 133, 134, 136, 142, 144, 197, 264, 309, 311]

path P
A series of vertices of a graph, connected by arcs. [11, 15–20, 23–25, 29, 30, 33–39, 44, 45, 49–57, 59–61, 63, 65, 69, 72, 73, 77, 78, 83, 87, 88, 102, 103, 117–125, 127, 129–131, 133–137, 141, 142, 147, 148, 150, 152, 155, 157–159, 162–165, 167, 172–180, 185, 191, 192, 194–198, 202, 203, 208, 212, 217, 219, 223, 225, 231–233, 239, 240, 243–247, 251–256, 260, 261, 266, 270, 272, 322]

planar
Planar is a possible property of a graph. A planar graph can be drawn on a 2D-surface without any crossings in the arcs. [9, 27, 40, 73]

plateau
A plateau describes a shared segment between two shortest paths. [52, 53, 55, 57–59, 62, 133, 148, 149, 152, 154–158, 162, 165, 232, 233, 235, 239, 240, 255]

potential π
A potential describes a built-up of *energy*. For our purposes, we view the potential as a lower bound for the length of a shortest path with respect to a given metric. [26, 58, 84]

priority queue $\mathcal{Q}$
Abstract data structure that enables efficient retrieval of elements ordered by some priority. [ix, 12–14, 18, 19, 23–26, 36, 38, 85–89, 94, 95, 97, 98, 100, 101, 103, 104, 107–109, 112–114, 151, 197, 249, 263, 264, 269, 306, 313]

prune
A process in which unnecessary overhead in a search is detected and the respective computations are eliminated before they are executed. [25, 63, 135, 147, 151, 159, 164, 166, 169, 193–195, 219–221, 225, 250, 254–257, 270]

R

radix heap
A specific implementation of a priority queue, utilizing properties of integer representation, maximum arc weights, and a bucket structure. [14, 144, 264]

reach r
A centrality measure on vertices. The reach of a vertex v is the maximum over the minimal distances to the source/target $(\mathcal{D}(s,v)/\mathcal{D}(v,t))$ of any potential shortest path that contains v. [28, 62–64, 179, 246]

reached
A vertex that has been assigned a tentative distance label. The label is not known to be final. [18, 19, 35, 103]

register
A register describes the typical access form on which the CPU operates. Data has to be transferred from RAM to a register before the CPU can perform operations on it. [92]

relaxation
In the context of shortest path algorithms, the relaxation of an arc represents the creation of new tentative distance labels by extending a path ending at a given vertex by and additional arc, adding the cost (defined by the metric) of the respective arc to the label. In hierarchical speed-up techniques, the relaxation weakens the hierarchy according to some criteria. [17, 18, 48, 64, 86, 101, 103, 113, 147–150, 154, 171, 198, 232, 250, 266]

robust $\Re$
The degree to what a corridor can be used to correct for incorrect driving behavior. [192–194, 196, 200, 201, 203, 226]

root
The root of a search tree is the topmost node. It is the only node that is not a descendant to any other node. [86–88, 93, 94, 98, 270]

S

search space $\mathcal{S}$
The search space with respect to a pruning (or speed-up) technique describes the subset of vertices of a graph that is considered during the pruned search. [25, 36, 37, 40, 46, 58, 131, 149–152, 161, 162, 164, 165, 194, 218–221, 223, 256]

search tree
A search tree is an ordered collection of interconnected nodes used to store elements that are identified by a comparable key. Every node splits its descendants into parts based on a set of router keys, distributing the stored elements into subtrees. Usually search trees are kept in balance, allowing to locate an arbitrary key within logarithmic time by following the router keys from the topmost node (the root) to a leaf. The most common search tree uses a single router key per node, splitting the elements in half. It is referred to as binary search tree. [86–90, 92–95, 97, 100, 102, 104, 264, 266, 267, 269]

segment $\mathcal{S}$
A continuous subsequence of vertices along a path. [4, 79, 117, 122–124, 152, 155, 157–159, 166, 171–181, 184, 185, 189, 240, 255]

separator $\mathcal{S}$
A separator to a set S of connected elements describes a set of elements that fragments S into disjoint parts. [21, 22, 27, 40, 41]

settled
A vertex with final distance label (correct shortest path distance) in an implementation of Dijkstra's algorithm or any implementation following the general procedure of Dijkstra's algorithm. [18, 19, 25, 36, 71, 85, 103, 198, 217]

shortcut
Artificial arc in a graph that represents an entire path (not necessarily a shortest path) within another graph. [27–29, 33–37, 39, 40, 43–46, 51, 63, 132, 147–149, 152–161, 165, 166, 196, 221–223, 267, 271, 272]

shortest path Π
path of minimal length between two vertices in a weighted graph. [1, 3, 10–12, 14–21, 23, 25, 26, 28–30, 33–38, 40, 41, 44–46, 49–57, 59–64, 68, 71, 72, 74, 76, 78, 83–85, 109, 112, 114, 115, 117, 120–122, 129–132, 135, 136, 138, 141, 144, 147–149, 151, 152, 154, 155, 162, 165, 167, 172, 173, 175–178, 180, 181, 187, 189, 191–198, 201, 203, 205–207, 209–211, 217–223, 225, 228, 239, 244, 252, 253, 256, 258, 261, 262, 265–272, 276, 305, 306, 318–320, 329–332]

simple
Property of a path within a graph. A simple path does not contain any vertex twice. [20]

source s
The origin of a shortest path query. [11, 15–20, 23–28, 30, 31, 34–38, 44, 45, 51–53, 56, 57, 59–63, 65, 70, 71, 73, 85, 103, 119–124, 131, 132, 135–138, 142, 147, 149–152, 154, 156, 162, 163, 175, 176, 178, 180, 183–185, 191–197, 199, 200, 202, 203, 217–219, 221, 222, 225, 228, 233, 243, 245, 248, 249, 251, 252, 254–257, 259, 264, 266, 269, 272, 321, 322]

speed-up
Relation between experienced running time of an algorithm and a comparative baseline algorithm. [1, 20, 22, 23, 25, 27–30, 33, 39, 49, 51–53, 59, 62, 64, 65, 105, 110, 112–114, 117, 138, 147, 148, 167, 193, 218, 262, 269–271, 306, 318–320]

stack
A container data structure that operates on the LIFO principle. [217]

T

target t
The destination of a shortest path query. [11, 16, 18–20, 23–28, 30, 31, 34–36, 44, 45, 51–53, 56, 57, 59–63, 65, 70, 71, 119–124, 131, 132, 135–138, 142, 147, 149–152, 156, 162, 163, 175, 176, 178, 180, 183–185, 192–200, 202, 203, 217–219, 221, 222, 225, 228, 233, 243, 245, 247–249, 251, 252, 254, 255, 264, 265, 269, 272, 321, 322]

time-dependent
Time-Dependency in routing considers a series of metrics. The metrics are referenced to by the expected arrival time at the origin of the arc. [16, 17, 39, 53, 55, 74, 76, 243, 244, 265]

U

undirected
Property of a graph. A graph is undirected if $\forall a = (u, v) \in A : (v, u) \in A$. [196]

unpack
Reverse process of shortcut creation or contraction. Used to extract the actual shortest paths from a speed-up-technique based on shortcuts. [37, 51, 153–159, 161]

unreached
A vertex that has not been assigned a tentative distance label less than ∞. [18, 19]

unweighted
Property of a graph. A graph is called unweighted if no metric has been defined on its arcs. [9]

V

vertex V
Basic construction element of a graph. In route planning vertices usually represents an intersection. [4, 9–11, 14–30, 33–38, 40–42, 44, 45, 48, 50–52, 54–56, 58–65, 68–71, 73, 74, 76, 84–89, 96, 97, 100–103, 107, 109, 110, 117, 118, 120–123, 130, 132–134, 147–163, 165–167, 171–181, 185, 189, 192–201, 203, 208, 212, 217–223, 226, 228, 231–233, 235, 240, 245–251, 253–257, 261, 263–272, 322, 335]

via-vertex v
Intermediate destination on a trip that, in combination with source and target fully describes an alternative route via the shortest paths $\Pi_{s,v}$ concatenated with $\Pi_{v,t}$. [4, 52, 58–60, 62–65, 147–149, 165, 167, 171–173, 180, 189, 226, 231, 232, 235, 240, 256, 257, 261, 306, 307, 335]

W

weight μ
Synonym for the value that a metric assigns to an arc. [43, 56, 103, 246]

weighted
Property of a graph. A graph is called weighted if a metric has been defined on its arcs. [9–11, 14–16, 19–21, 34, 35, 41, 56, 85, 150, 265, 270]

witness
In the context of contraction, a witness to a potential shortcut is a path that is shorter then the potential shortcut and thus proves the shortcut to be unnecessary. [29, 33–35, 43, 162]

Ω Acronyms

Symbols

CRP-∞
Customizeable Route Planning with Infinite Penalization [171, 176, 180, 181, 183, 186, 232–235, 239, 240, 338]

CRP-π
Customizeable Route Planning with Penalization [144, 171–173, 176, 180–186, 189, 232–235, 239, 240, 309, 312, 321, 325, 338]

A

ALT
A* Landmarks Triangle Inequality [26, 58, 117–119, 193, 194]

ArcFlags
Arc-Flags [148, 193–195, 246]

B

BFS
breadth-first search [150]

BS
bounded stretch [60, 61, 76, 78, 79, 121–124, 133, 142, 164, 172, 178, 180, 185, 199, 233, 239, 261, 264, 267, 305, 335]

C

CCH
Customizeable Contraction Hierarchies [118]

CDF
Cumulative Distribution Function [247–249, 251, 254–259]

CH
Contraction Hierarchy [29, 30, 33–41, 43–46, 51, 58, 63–65, 118, 135, 148–154, 159, 161, 164, 167, 171, 172, 181, 185, 189, 192, 194, 219, 222, 223, 225, 228, 231, 232, 240, 255, 267, 306]

CPU
Central Processing Unit [29, 37, 46–48, 92, 107, 181, 185, 269, 322, 323]

CRP
Customizeable Route Planning [27, 33, 41–47, 50, 118, 119, 129–137, 139, 140, 144, 171–173, 176, 179–186, 189, 194, 196, 231–235, 239, 240, 273, 307, 309, 312, 321, 322, 325, 338]

D

DAG
Directed Acyclic Graph [50, 150, 154, 192, 198, 253, 254]

DFS
depth-first search [14, 15, 162, 164, 199]

DFT
Discrete Fourier-Transformation [250]

F

FFT
Fast Fourier-Transformation [246, 249, 250]

G

GPGPU
General Purpose Graphics Processing Unit [38, 46–48, 300, 307]

GPS
Global Positioning System [68, 75, 201]

H

HH
Highway Hierarchies [28, 29]

HiDAR
Hierarchy Decomposition for Alternative Routes [149, 151, 152, 154, 158, 162, 164–167, 169, 170, 194, 196, 232–235, 239, 240, 255, 337]

HNR
Highway Node Routing [28, 29, 40, 58, 118]

I

I/O
Input/Output [94]

ID
Identification [96, 156, 158, 161]

L

LO
local optimality [60–62, 76, 78, 122, 129, 133, 142, 148, 167, 185, 199, 239, 247, 255, 261, 267, 305, 335]

LS
limited sharing [60, 61, 124]

P

PEM
Parallel External Memory [94]

PHAST
Parallel Hardware Accelerated Shortest Path Trees [30, 37, 38, 148–151, 218, 220, 222]

R

RAM
Random Access Memory [92, 269]

RPHAST
Restricted PHAST [149–152]

S

SA
Successive Approximation [245, 247]

SCC Strongly Connected Component	[14, 69]
SOTA stochastic on-time arrival problem	[243–258, 260, 262, 307]
SPOTAR shortest path with on-time arrival reliability	[244–246, 251, 252]
SPP Single Source Shortest Path Problem	[16–18, 22, 23, 49, 70, 84, 85, 243]
STD Standard Template Library	[105, 176]

T

TLB Transaction Look-aside Buffer	[94]
TNR Transit Node Routing	[30, 31, 41]
TTF travel time function	[16, 17, 53, 55]

U

UDG Unit Disc Graph	[74]

W

WLAN Wireless Local Area Network	[191]

Z

ZDC Zero-Delay Convolution	[249, 250]

Ω Bibliography

[ADF$^+$11] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. VC-dimension and shortest path algorithms. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 6755 of *Lecture Notes in Computer Science*, pages 690–699. Springer, 2011. [see pages 30 and 40]

[ADF$^+$12] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. HLDB: Location-based services in databases. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, pages 339–348, 2012. [see page 30]

[ADF$^+$13] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension and provably efficient shortest path algorithms. Technical Report MSR-TR-2013-91, Microsoft Research, 2013. [see pages 40 and 41]

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths on road networks. In *International Symposium on Experimental Algorithms (SEA)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011. [see pages 30 and 92]

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hierarchical hub labelings for shortest paths. In *Experimental Algorithms*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer, 2012. [see page 30]

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative routes in road networks. *ACM Journal of Experimental Algorithms*, 18(1):1–17, 2013. [see pages 52, 59, 61, 62, 64, 65, 68, 76, 78, 147, 149, 164, 167, 171, 175, 180, 232, and 233]

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 782–793. Society for Industrial and Applied Mathematics (SIAM), 2010. [see pages 40, 41, and 62]

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *International Symposium on Experimental Algorithms (SEA)*, pages 55–66, 2013. [see page 30]

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, April 1990. [see page 14]

[AV03] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing (SICOMP)*, 32(6):1488–1508, 2003. [see pages 91, 93, 94, and 95]

[BABDR84] M Ben-Akiva, MJ Bergman, Andrew J Daly, and Rohit Ramaswamy. Modeling inter-urban route choice behaviour. In *International Symposium on Transportation and Traffic Theory*, pages 299–330. VNU Press, 1984. [see pages 49 and 50]

[Bat14] Gernod Veit Batz. *Time-Dependent Route Planning with Contraction Hierarchies.* Phd thesis, Karlsruhe Institute of Technology, Department of Informatics, 2014. [see pages 16, 39, 53, and 243]

[BCK$^+$10] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing speed-up techniques is hard. In *Algorithms and Complexity (CIAC)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010. [see page 34]

[BCRW13] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, Lecture Notes in Computer Science, pages 93–104, 2013. [see pages 39 and 43]

[BDFC00] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious B-trees. In *Foundations of Computer Science*, pages 399–409. IEEE Computer Society, 2000. [see page 93]

[BDG$^+$15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *Computing Research ReposFitory*, abs/1504.05140, 2015. [see pages 25, 29, 30, and 68]

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. Alternative route graphs in road networks. In *International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems (TAPAS)*, volume 6595 of *Lecture Notes in Computer Science*, pages 21–32. Springer, 2011. [see pages 50, 51, 52, 54, 55, 57, 58, 59, 76, 77, 115, 117, 119, 120, 121, 125, 128, 144, 148, 167, 181, and 233]

[BDMR96] Armin Bäumker, Wolfgang Dittrich, Friedhelm Meyer auf der Heide, and Ingo Rieping. Realistic parallel algorithms: Priority queue operations and selection for the BSP model. In *Parallel Processing (EuroPar)*, Lecture Notes in Computer Science, pages 369–376. Springer, 1996. [see page 95]

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958. [see pages 15 and 19]

[BFGK05] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237. Association for Computing Machinery (ACM), 2005. [see page 93]

[BFM$^+$07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics (SIAM), Society for Industrial and Applied Mathematics (SIAM), 2007. [see page 30]

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast shortest-path queries via transit nodes. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:175–192, 2009. [see pages 30 and 68]

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824), 2007. [see pages 30 and 41]

[BGL+12] Gernot V. Batz, Robert Geisberger, Dennis Luxen, Peter Sanders, and Roman Zubkov. Efficient route compression for hybrid route planning. In *Mediterranean Conference on Algorithms (MedAlg'12)*, volume 7659 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2012. [see page 53]

[BGSV13] Gernot V. Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithms*, 18(1.4):1–43, 2013. [see page 53]

[BK60] Richard Bellman and Robert Kalaba. On k-th best policies. *Journal of the Society for Industrial & Applied Mathematics*, 8(4):582–588, 1960. [see page 51]

[BKMW10] Reinhard Bauer, Marcus Krug, Sascha Meinert, and Dorothea Wagner. Synthetic road networks. In *Algorithmik Aspects of Information and Management (AAIM)*, Lecture Notes in Computer Science, pages 46–57. Springer, 2010. [see page 41]

[BM72] R Bayer and EM McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. [see page 92]

[BM02] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes.* Springer, 2002. [see page 92]

[BMS+13] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *Computing Research ReposFitory*, abs/1311.3144, 2013. [see page 22]

[Bro99] Gerth Stølting Brodal. Priority queues on parallel machines. *Parallel Computing*, 25(8), 1999. [see page 95]

[Cam05] Cambridge Vehicle Information Tech. Ltd. Choice routing, 2005. `http://camvit.com/camvit-technical-english/Camvit-Choice-Routing-Explanation-english.pdf`. Accessed: 2014-03-28. [see pages 52, 57, 62, and 148]

[CBB07] Yanyan Chen, Michael G. H. Bell, and Klaus Bogenberger. Reliable pretrip multipath planning and dynamic adaptation for a centralized road navigation system. *IEEE Transactions on Intelligent Transportation Systems (ITSC)*, 8(1):14–20, 2007. [see pages 55 and 244]

[CBWB06] Yanyan Chen, Michael GH Bell, Dongzhu Wang, and Klaus Bogenberger. Risk-averse time-dependent route guidance by constrained dynamic A* search in decentralized system architecture. *Transportation Research Record: Journal of the Transportation Research Board*, 1944(1):51–57, 2006. [see page 244]

[CGG$^+$95] Yi-Jen Chiang, Michael T Goodrich, Edward F Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 139–149. Society for Industrial and Applied Mathematics (SIAM), Society for Industrial and Applied Mathematics (SIAM), 1995. [see page 38]

[CGM01] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *ACM SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 235–246. Association for Computing Machinery (ACM), 2001. [see pages 93 and 94]

[CGS97] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 83–92. Society for Industrial and Applied Mathematics (SIAM), 1997. [see page 14]

[CH94] Danny Z. Chen and Xiaobo Hu. Fast and efficient operations on parallel priority queues. In *International Symposium on Algorithms and Computation (ISAAC)*, Lecture Notes in Computer Science, pages 279–287. Springer, 1994. [see page 95]

[CHL99] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, pages 1–12. Association for Computing Machinery (ACM), 1999. [see page 93]

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959. [see page 152]

[Cor14] NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation, 2014. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. [see page 47]

[CYGW07] Kun Chen, Lei Yu, Jifu Guo, and Huimin Wen. Characteristics analysis of road network reliability in beijing based on the data logs from taxis. In *Transportation Research Board 86th Annual Meeting*, number 07-1420 in TRB Annual Meeting Compendium, 2007. [see page 74]

[Dan62] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962. [see page 23]

[DBGO14] Andrew Alan Davidson, Sean Baxter, Michael Garland, and John D Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *International Parallel and Distributed Processing Symposium (IPDPS)*, volume 28. IEEE Computer Society, 2014. [see page 47]

[DBKS10] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. A case for time-dependent shortest path computation in spatial networks. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, pages 474–477. Association for Computing Machinery (ACM), 2010. [see page 244]

[DDF14] Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. Fully dynamic update of arc-flags. *Networks*, 2014. Accepted for publication. [see page 118]

[DDFV12] Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, and Camillo Vitale. Fully dynamic maintenance of arc-flags in road networks. In *International Symposium on Experimental Algorithms (SEA)*, volume 7276 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2012. [see pages 58 and 118]

[DDP$^+$13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing multimodal journeys in practice. In *International Symposium on Experimental Algorithms (SEA)*, Lecture Notes in Computer Science, pages 260–271. Springer, 2013. [see pages 83 and 114]

[Dea10] Brian C Dean. Speeding up stochastic dynamic programming with zero-delay convolution. *Algorithmic Operations Research*, 5(2):Pages–96, 2010. [see page 249]

[DF79] Eric V. Denardo and Bennett L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27(1):pp. 161–186, 1979. [see page 13]

[DFV11] Gianlorenzo D'Angelo, Daniele Frigioni, and Camillo Vitale. Dynamic arc-flags in road networks. In *International Symposium on Experimental*

Algorithms (SEA), volume 6630 of *Lecture Notes in Computer Science*, pages 88–99. Springer, 2011. [see pages 58 and 118]

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book Series*. American Mathematical Society (AMS), 2009. [see page 23]

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013. [see pages 27, 29, 37, 92, 148, 149, and 220]

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In *International Symposium on Experimental Algorithms (SEA)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011. [see pages 27, 41, 45, 129, and 133]

[DGPW13] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks, 2013. Preprint at `http://research.microsoft.com/pubs/198358/crp_web_130724.pdf`. Accessed: 2014-03-28. [see pages 27, 41, 45, 68, 125, and 129]

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph partitioning with natural cuts. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1135–1146. IEEE Computer Society, 2011. [see pages 27, 40, 41, and 133]

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Faster batched shortest paths in road networks. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 20 of *OpenAccess Series in Informatics (OASICS)*, pages 52–63, 2011. [see pages 149, 150, 220, and 222]

[DGW13] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Hub label compression. In *International Symposium on Experimental Algorithms (SEA)*, pages 18–29, 2013. [see page 30]

[DHM$^+$09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-performance multi-level routing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of

DIMACS Book Series, pages 73–92. American Mathematical Society (AMS), 2009. [see page 27]

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959. [see pages 15, 18, and 23]

[DKLW12] Daniel Delling, Moritz Kobitzsch, Dennis Luxen, and Renato F. Werneck. Robust mobile route planning with limited connectivity. In *Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 150–159. Society for Industrial and Applied Mathematics (SIAM), 2012. [see pages 191 and 192]

[DKW14] Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. Customizing driving directions with GPUs. In *Parallel Processing (EuroPar)*, Lecture Notes in Computer Science. Springer, 2014. [see page 46]

[dlBPA93] Tomas de la Barra, B Perez, and J Anez. Multidimensional path search and assignment. In *PTRC Summer Annual Meeting (SAM)*, 1993. [see pages 51, 55, and 56]

[DP84] Narsingh Deo and Chi-Yin Pang. Shortest-path algorithms: Taxonomy and annotation. *Computer Networks*, 14(2):275–323, 1984. [see page 11]

[DP92] Narsingh Deo and Sushil K. Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992. [see page 95]

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *Experimental Algorithms*, volume 8504 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2014. [see pages 46, 51, and 58]

[DW07] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In *Experimental Algorithms*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2007. [see page 117]

[DW09] Daniel Delling and Dorothea Wagner. Pareto paths with SHARC. In *International Symposium on Experimental Algorithms (SEA)*, Lecture Notes in Computer Science, pages 125–136. Springer, 2009. [see page 52]

[DW13] Daniel Delling and Renato F. Werneck. Faster customization of road networks. In *International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013. [see pages 43, 44, 47, 129, and 134]

[Edm70] Jack Edmonds. Exponential growth of the simplex method for shortest path problems. *manuscript [University of Waterloo, Waterloo, Ontario]*, 1970. [see page 18]

[EG08] David Eppstein and Michael T Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, page 16. Association for Computing Machinery (ACM), Association for Computing Machinery (ACM), 2008. [see pages 27 and 40]

[EKS14] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced B-trees with bulk updates. In *International Symposium on Experimental Algorithms (SEA)*, volume 8504 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2014. [see pages 91 and 107]

[Epp94] David Eppstein. Finding the k shortest paths. In *Foundations of Computer Science*, pages 154–165. IEEE Computer Society, 1994. [see page 50]

[Epp98] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing (SICOMP)*, 28(2):652–673, 1998. [see page 50]

[EPV11] Alexandros Efentakis, Dieter Pfoser, and Agnès Voisard. Efficient data management in support of shortest-path computation. In *ACM SIGSPATIAL International Workshop on Computational Transportation Science (CTS)*, pages 28–33. Association for Computing Machinery (ACM), 2011. [see page 28]

[Erb13] Stephan Erb. Engineering parallel bi-criteria shortest path search. Master's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2013. [not cited]

[Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962. [see page 21]

[FN06] Yueyue Fan and Yu Nie. Optimal routing for maximizing the travel time reliability. *NSpEc*, 6(3-4):333–344, 2006. [see pages 245 and 247]

[For56] Lester R. Ford. Network flow theory. Technical Report Report P-923, The Rand Corporation, 1956. [see pages 15, 17, and 19]

[FS07] Leonor Frias and Johannes Singler. Parallelization of bulk operations for STL dictionaries. In *Parallel Processing (EuroPar)*, Lecture Notes in Computer Science, pages 49–58. Springer, 2007. [see pages 90, 97, 102, 103, 104, and 105]

[FSR06] L. Fu, D. Sun, and L. R. Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research*, 33(11), 2006. [see page 28]

[FW56] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. *Naval research logistics quarterly*, 3(1-2):95–110, 1956. [see page 51]

[Gei14] R. Geisberger. Alternate directions in hierarchical road networks, September 2 2014. US Patent 8,824,337. [see pages 64 and 147]

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 156–165. Society for Industrial and Applied Mathematics (SIAM), 2005. [see page 26]

[GJ79] Michael Randolph Garey and David Stifler Johnson. *Computers and Intractability.* Freeman and Company, 1979. [see page 16]

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route planning with flexible objective functions. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 124–137. Society for Industrial and Applied Mathematics (SIAM), 2010. [see page 52]

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest path algorithms with preprocessing. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book Series*, pages 93–139. American Mathematical Society (AMS), 2009. [see page 28]

[GL01] Goetz Graefe and Per-Åke Larson. B-tree indexes and CPU caches. In *International Conference on Data Engineering (ICDE)*, pages 349–358. IEEE Computer Society, 2001. [see page 94]

[GPPR04] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85–112, 2004. [see page 30]

[GS78] Leonidas J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science*, pages 8–21. IEEE Computer Society, 1978. [see page 87]

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012. [see pages 29, 33, 36, 39, 191, and 225]

[GSVGM98] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity search in databases. In *International Conference on Very Large Data Bases (VLDB)*, pages 26–37. Morgan Kaufmann Publishers Inc., 1998. [see page 27]

[Gut04] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111. Society for Industrial and Applied Mathematics (SIAM), 2004. [see page 28]

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing point-to-point shortest paths from external memory. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 26–40. Society for Industrial and Applied Mathematics (SIAM), 2005. [see page 26]

[HAB12] Timothy Hunter, Pieter Abbeel, and Alexandre M. Bayen. The path inference filter: Model-based low-latency map matching of probe vehicle data. In *Algorithmic Foundations of Robotics X*, STAR, pages 591–607. Springer, 2012. [see page 75]

[Han80] Pierre Hansen. Bricriterion path problems. In *Multiple Criteria Decision Making (MCDM)*, volume 177 of *Lecture Notes on Economics and Mathematical Systems*, pages 109–127. Springer, 1980. [see pages 52 and 84]

[HAP12] Xi He, Dinesh Agarwal, and Sushil K. Prasad. Design and implementation of a parallel priority queue on many-core architectures. In *High Performance Computing (HiPC)*, pages 1–10. IEEE Computer Society, 2012. [see page 95]

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book Series*, pages 41–72. American Mathematical Society (AMS), 2009. [see pages 26 and 27]

[HL74] Robert Herman and Tenny Lam. Trip time characteristics of journeys to and from work. *Transportation and traffic theory*, 6:57–86, 1974. [see pages 74 and 75]

[HM82] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. [see page 92]

[HM11] Petr Hlinený and Ondrej Moris. Scope-based route planning. In *Experimental Algorithms*, Lecture Notes in Computer Science, pages 445–456. Springer, 2011. [see page 28]

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Man, and Cybernetics (SMC)*, 4(2):100–107, 1968. [see page 26]

[HP03] Richard A. Hankins and Jignesh M. Patel. Effect of node size on the performance of cache-conscious B$^+$-trees. In *ACM SIGMETRICS International Conference on Measurement and Modelling of Computer Systems*, pages 283–294. Association for Computing Machinery (ACM), 2003. [see pages 94 and 95]

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. *ACM Journal of Experimental Algorithms*, 13(2.5):1–26, 2008. [see page 41]

[IHI$^+$94] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, and Kunihiko Mitoh. A fast algoritm for finding better routes by AI search techniques. In *Vehicle Navigation and Information Systems*, pages 291–296. IEEE Computer Society, 1994. [see page 26]

[Jac55] E Jacobitti. Automatic alternate routing in the 4A crossbar system. *Bell Laboratories Record*, 33:141–145, 1955. [see pages 49 and 50]

[JHR98] Ning Jing, Yun-Wu Huang, and Elke A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Transactions Knowledge and Data Engineering (TKDE)*, 10(3):409–432, 1998. [see page 27]

[JP02] Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions Knowledge and Data Engineering (TKDE)*, 14(5):1029–1046, 2002. [see page 27]

[KK97] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research (JAIR)*, 7:283–317, 1997. [see page 26]

[KMVP14] Martin Kagerbauer, Nicolai Mallig, Peter Vortisch, and Manfred Pfeiffer. Modellierung von Variabilität und Stabilität des Verkehrsverhaltens im Längsschnitt mit Hilfe der Multi-Agenten-Simulation mobiTopp. In *HEUREKA*, pages 155–173. Forschungsgesellschaft für Straßen und Verkehrswesen (FGSV), 2014. [see page 75]

[Kob13] Moritz Kobitzsch. An alternative approach to alternative routes: HiDAR. In *Experimental Algorithms*, volume 8125 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2013. [see page 147]

[Kol15] Orlin Kolev. Blocking alternative routes (tentative title). Diploma's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2015. [not cited]

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. Evolution and evaluation of the penalty method for alternative graphs. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 33 of *OpenAccess Series in Informatics (OASICS)*, pages 94–107. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. [see page 117]

[KSS$^+$07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–45. Society for Industrial and Applied Mathematics (SIAM), 2007. [not cited]

[Lau09] Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book Series*, pages 19–40. American Mathematical Society (AMS), 2009. [see page 26]

[LGJ$^+$57] M Leyzorek, RS Gray, AA Johnson, WC Ladew, SR Meaker Jr, RM Petry, and RN Seitz. A study of model techniques for communication systems. *Investigation of Model Techniques*, 1957. [see page 18]

[LLSL10] Ig-hoon Lee, Jae-won Lee, Junho Shim, and Sang-goo Lee. Cache conscious trees on modern microprocessors. In *International Conference on Ubiquitous Information Management and Communication (ICUIMC)*, page 43. Association for Computing Machinery (ACM), January 2010. [see page 89]

[LRT79] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis (SINUM)*, 16(2):346–358, 1979. [see pages 40 and 45]

[LS11] Dennis Luxen and Peter Sanders. Hierarchy decomposition for faster user equilibria on road networks. In *International Symposium on Experimental Algorithms (SEA)*, Lecture Notes in Computer Science, pages 242–253, 2011. [see pages 51, 58, 148, and 153]

[LS12] Dennis Luxen and Dennis Schieferdecker. Candidate sets for alternative routes in road networks. In *International Symposium on Experimental Algorithms (SEA)*, volume 7276 of *Lecture Notes in Computer Science*, pages 260–270. Springer, 2012. [see pages 52, 64, 76, 78, 147, 148, 164, 231, 232, 233, and 234]

[LSSSS07] Timo Lilja, Riku Saikkonen, Seppo Sippu, and Eljas Soisalon-Soininen. Online bulk deletion. In *International Conference on Data Engineering (ICDE)*, pages 956–965. IEEE Computer Society, 2007. [see page 94]

[LSTM03] Tim Lomax, David Schrank, Shawn Turner, and Richard Margiotta. Selecting travel reliability measures. *Texas Transportation Institute monograph (May 2003)*, 2003. [see page 74]

[LT79] Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. [see pages 27 and 40]

[Luc94] E. Lucas. *Récréations mathématiques*. Number Bd. 4 in Récréations mathématiques. Gauthier-Villars, 1894. [see page 14]

[Mar84] Ernesto Queiros Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, 1984. [see page 84]

[MdlC05] Lawrence Mandow and José-Luis Pérez de-la Cruz. A new approach to multiobjective A* search. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 218–223. Association for the Advancement of Artificial Intelligence, 2005. [see pages 52 and 84]

[MdlC10] Lawrence Mandow and José-Luis Pérez de-la Cruz. Multiobjective A^* search with consistent heuristics. *Journal of the ACM*, 57(5), 2010. [see page 84]

[MHM00] Elise Miller-Hooks and Hani S. Mahmassani. Least expected time paths in stochastic, time-varying transportation networks. *Transportation Science*, 34(2):198–215, 2000. [see page 17]

[MHW01] Matthias Müller-Hannemann and Karsten Weihe. Pareto shortest paths is often feasible in practice. In *Algorithm Engineering*, pages 185–198. Springer, 2001. [see page 16]

[Mil12] Nikola Milosavljević. On optimal preprocessing for contraction hierarchies. In *ACM SIGSPATIAL International Workshop on Computational Transportation Science (CTS)*, pages 33–38. Association for Computing Machinery (ACM), 2012. [see pages 39 and 40]

[Min57] George J Minty. Letter to the editor-A comment on the shortest-route problem. *Operations Research*, 5(5):724–724, 1957. [see page 49]

[MM12] Enrique Machuca and Lawrence Mandow. Multiobjective heuristic search in road maps. *Expert Systems with Applications*, 39(7):6435–6445, 2012. [see page 73]

[MMdlCRS12] Enrique Machuca, Lawrence Mandow, José-Luis Pérez de-la Cruz, and Amparo Ruiz-Sepúlveda. A comparison of heuristic best-first algorithms for bicriterion shortest path problems. *European Journal of Operational Research*, 217(1):44–53, 2012. [see pages 84 and 90]

[MMO91] John Mote, Ishwar Murthy, and David L Olson. A parametric approach to solving bicriterion shortest path problems. *European Journal of Operational Research*, 53(1):81–92, 1991. [see page 73]

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching, and Annals of the Computation Laboratory of Harvard University*, pages 285–292. Harvard University Press, 1959. [see pages 15 and 19]

[MS03] Ulrich Meyer and Peter Sanders. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. [see pages 29 and 103]

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, first edition, 2008. [see pages 10, 11, 14, 87, and 88]

[MSS$^+$06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup Dijkstra's algorithm. *ACM Journal of Experimental Algorithms*, 11, 2006. [see page 26]

[Neu09] Sabine Neubauer. Planung energieeffizienter Routen in Straßennetzwerken. Diploma's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2009. [not cited]

[NF06] Yu Nie and Yueyue Fan. Arriving-on-time problem. *Transport. Res. Rec.*, 1964:193–200, 2006. [see page 246]

[NKBM06] Evdokia Nikolova, Jonathan A. Kelner, Matthew Brand, and Michael Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *Experimental Algorithms*, Lecture Notes in Computer Science, pages 552–563. Springer, 2006. [see page 245]

[NW09] Yu Nie and Xing Wu. Shortest path problem considering on-time arrival probability. *Transport. Res. B-Meth.*, 43(6):597 – 613, 2009. [see page 245]

[OR91] Ariel Orda and Raphael Rom. Minimum weight paths in time-dependent networks. *Networks*, 21(3):295–319, 1991. [see page 16]

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983. [see page 98]

[Par61] Seymour Parter. The use of linear graphs in Gauss elimination. *SIAM review*, 3(2):119–130, 1961. [see page 39]

[Pea05] Karl Pearson. The problem of the random walk. *Nature*, 72(1865):294, 1905. [see page 201]

[Pel00] David Peleg. Proximity-preserving labeling schemes. *Journal of Graph Theory*, 33(3):167–176, 2000. [see page 30]

[PMSS04] Kerttu Pollari-Malmi and Eljas Soisalon-Soininen. Concurrency control and I/O-optimality in bulk insertion. In *International Symposium on String Processing and Information Retrieval (SPIRE)*, Lecture Notes in Computer Science, pages 161–170. Springer, 2004. [see page 94]

[PMSSY96] Kerttu Pollari-Malmi, Eljas Soisalon-Soininen, and Tatu Ylönen. Concurrency control in B-trees with batch updates. *IEEE Transactions Knowledge and Data Engineering (TKDE)*, 8(6), 1996. [see page 94]

[Poh70] Ira Pohl. *Bi-directional search.* IBM TJ Watson Research Center, 1970. [see page 26]

[PP91] Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991. [see page 95]

[PS13] José Manuel Paixão and José Luis Santos. Labeling methods for the general case of the multi-objective shortest path problem – A computational study. In *Computational Intelligence and Decision Making*, pages 489–502. Springer, 2013. [see pages 84 and 194]

[PZ13] Andreas Paraskevopoulos and Christos D. Zaroliagis. Improved alternative route planning. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, volume 33 of *OpenAccess Series in Informatics (OASICS)*, pages 108–122. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. [see pages 58, 59, 70, 117, 118, 119, 120, 121, 125, 127, 128, 129, and 148]

[Rad12] Marcel Radermacher. Schnelle Berechnung von Alternativgraphen. Bachelor's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2012. [not cited]

[RE09] Andrea Raith and Matthias Ehrgott. A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36(4):1299–1331, 2009. [see page 73]

[RESAD06] Hesham Rakha, Ihab El-Shawarby, Mazen Arafeh, and Francois Dion. Estimating path travel-time reliability. In *Intelligent Transportation Systems Conference, 2006. ITSC*, pages 236–241. IEEE, IEEE Computer Society, 2006. [see page 74]

[RK88] V. Nageshwara Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers (TC)*, 37(12):1657–1665, 1988. [see page 95]

[RR00] Jun Rao and Kenneth A. Ross. Making B^+-trees cache conscious in main memory. In *ACM SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 475–486. Association for Computing Machinery (ACM), 2000. [see pages 93 and 94]

[Rup97] Eric Ruppert. Finding the k shortest paths in parallel. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, pages 475–486. Springer, 1997. [see page 50]

[Sai09] Riku Saikkonen. *Bulk updates and cache sensitivity in search trees.* PhD thesis, Aalto University, 2009. [see page 93]

[San98] Peter Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998. [see page 95]

[San09] Peter Sanders. Algorithm engineering - an attempt at a definition. In *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009. [see page 3]

[SB10] David L Sonnier and Matt Bradley. Criteria-based parallelism for multiobjective problem solving. *Journal of Computing Sciences in Colleges*, 25(5):124–130, 2010. [see page 83]

[SBB12a] Samitha Samaranayake, Sebastien Blandin, and Alexandre Bayen. A tractable class of algorithms for reliable routing in stochastic networks. *Transportation Research Part C: Emerging Technologies*, 20(1):199–217, 2012. [see pages 245, 246, 247, and 248]

[SBB12b] Samitha Samaranayake, Sebastien Blandin, and Alexandre M. Bayen. Speedup techniques for the stochastic on-time arrival problem. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS)*, OpenAccess Series in Informatics (OASICS), pages 83–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. [see pages 17, 246, and 250]

[Sch82] Robert Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software (TOMS)*, 8(3):256–276, 1982. [see page 39]

[Sch08a] Dominik Schultes. *Route Planning in Road Networks.* PhD thesis, Universität Karlsruhe TH, 2008. [see page 28]

[Sch08b] Dominik Schultes. Routing in road networks with transit nodes. In *Encyclopedia of Algorithms*, pages 1–99. Springer, 2008. [see page 30]

[Sch10] Alexander Schrijver. On the history of the shortest path problem. *Documenta Mathematica*, pages 155–165, 2010. [see page 17]

[Sch13] Christian Schulz. *High Quality Graph Partitioning.* Phd thesis, Karlsruhe Institute of Technology, Department of Informatics, 2013. [see pages 22, 27, and 41]

[Sch14] Dennis Schieferdecker. *An Algorithmic View on Sensor Networks - Surveillance, Localization, and Communication.* Phd thesis, Karlsruhe Institute of Technology, Department of Informatics, 2014. [see page 74]

[SCK$^+$11] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *International Conference on Very Large Data Bases (VLDB)*, 4(11), 2011. [see page 94]

[Sed96] J.W. Seda. Route planning method for hierarchical map routing and apparatus therefor, May 21 1996. US Patent 5,519,619. [see page 25]

[Shi53] Alfonso Shimbel. Structural parameters of communication networks. *The bulletin of mathematical biophysics*, 15(4):501–507, 1953. [see page 21]

[Shi54] Alfonso Shimbel. Structure in communication nets. In *Symposium on Information Networks*, volume 3, pages 199–203. Polytechnic Institute of Brooklyn, 1954. [see page 19]

[Shi79] Douglas R. Shier. On algorithms for finding the k shortest paths in a network. *Networks*, 9(3):195–214, 1979. [see page 50]

[SI91] Bradley S. Stewart and Chelsea C. White III. Multiobjective A*. *Journal of the ACM*, 38(4):775–814, 1991. [see pages 52 and 84]

[SM13] Peter Sanders and Lawrence Mandow. Parallel label-setting multiobjective shortest path search. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 215–224. IEEE Computer Society, 2013. [see pages 52, 83, 84, 86, 87, 89, 90, 94, 96, 100, 101, 102, and 302]

[Som14] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4), 2014. [see page 40]

[Son06] David L Sonnier. Parallel algorithms for multicriteria shortest path problems. *Journal ofthe Arkansas Academy of Science*, 60, 2006. [see page 83]

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Experimental Algorithms*, pages 568–579, 2005. [see pages 28 and 71]

[SS06] Peter Sanders and Dominik Schultes. Robust, almost constant time shortest-path queries in road networks. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 2006. [see page 30]

[SS07] Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In *International Workshop on Experimental Algorithms (WEA)*, volume 4525 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2007. [see pages 29 and 118]

[SS12a] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. *ACM Journal of Experimental Algorithms*, 17(1):1–40, 2012. [see page 28]

[SS12b] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 16–29. Society for Industrial and Applied Mathematics (SIAM), 2012. [see page 27]

[SS13] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *Lecture Notes in Computer Science*, pages 164–175. Springer, 2013. [see page 27]

[SSB14] Guillaume Sabran, Samitha Samaranayake, and Alexandre M. Bayen. Precomputation techniques for the stochastic on-time arrival problem. In *Meeting on Algorithm Engineering and Experiments (ALENEX)*, pages 138–146. Society for Industrial and Applied Mathematics (SIAM), 2014. [see pages 246 and 247]

[SV86] Robert Sedgewick and Jeffrey Scott Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1(1-4):31–48, 1986. [see page 26]

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In *ACM Journal of Experimental Algorithms*, pages 110–123. Association for Computing Machinery (ACM), 1999. [see page 27]

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Using multilevel graphs for timetable information in railway systems. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002. [see page 27]

[SZ12] Nodari Sitchinava and Norbert Zeh. A parallel buffer tree. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 214–223. Association for Computing Machinery (ACM), 2012. [see page 94]

[Tar95] G. Tarry. *Le problème des labyrinthes.* Gauthier-Villars, 1895. [see page 14]

[Tho04] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences (JCSS)*, 69(3):330–353, 2004. [see pages 23 and 37]

[Tru52] Darrel L Trueblood. Effect of travel time and distance on freeway usage. *Highway Research Board Bulletin*, 61(61), 1952. [see page 49]

[TTLC92] Chi Tung Tung and Kim Lin Chew. A multicriteria pareto-optimal path algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992. [see page 84]

[vEBKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. [see page 92]

[Vet09] Christian Vetter. Parallel time-dependent contraction hierarchies. Student research project, Karlsruhe Institute of Technology, Department of Informatics, 2009. [see page 34]

[VV78] Dirck Van Vliet. Improved shortest path algorithms for transport networks. *Transportation Research*, 12(1):7–20, 1978. [see page 27]

[Wil64] John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347/348, 1964. [see page 13]

[Win02] Stephan Winter. Modeling costs of turns in route planning. *GeoInformatica*, 6(4):363–380, 2002. [see page 10]

[Yen71] Jin Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971. [see page 50]

Ω List of Publications

Peer-reviewed Conference Papers

[DKW14] Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. Customizing driving directions with GPUs. In EuroPar, 2014.

[EKS14] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. Parallel bi-objective shortest paths using weight-balanced B-trees with bulk updates. In Joachim Gudmundsson and Jyrki Katajainen, editors, SEA, volume 8504 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2014.

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. Evolution and Evaluation of the Penalty Method for Alternative Graphs. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 94–107. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.

[Kob13] Moritz Kobitzsch. An alternative approach to alternative routes: Hidar. In *European Symposium on Algorithms (ESA'13)*, volume 8125 of *Lecture Notes in Computer Science*, pages 613–624. Springer, 2013.

[DKLW12] Daniel Delling, Moritz Kobitzsch, Dennis Luxen, and Renato F. Werneck. Robust mobile route planning with limited connectivity. In *Meeting on Algorithm Engineering and Experiments* (ALENEX'12), pages 150–159. SIAM, 2012.

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route planning with flexible objective functions. In *Workshop on Algorithm Engineering and Experiments* (ALENEX'10), pages 124–137. SIAM, 2010.

Technical Reports

[KSS14] Moritz Kobitzsch, Samitha Samaranayake, Dennis Schieferdecker. Pruning Techniques for the Stochastic on-time Arrival Problem- An Experimental Study. 2014

[Kob13] Moritz Kobitzsch. An Alternative Approach to Alternative Routes: HiDAR. Technical Report. Karlsruhe Reports in Informatics 2013,5. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000035372`, 2013.

Theses

[Kob09] Moritz Kobitzsch. Flexible Route-Planning with Contraction Hierarchies. Diploma's thesis, University of Karlsruhe, Department of Informatics, 2009.

[Kob07] Moritz Kobitzsch. Kreuzungsminimierung in Kreislayouts. Student thesis, University of Karlsruhe, Department of Informatics, 2007.

Supervised Theses

[Kol15] Orlin Kolev. Blocking Alternative Routes (tentative title). Diploma's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2015.

[Erb13] Parallel Bi-Criteria Shortest Path Search. Master's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2013.

[Rad12] Marcel Radermacher. Schnelle Berechnung von Alternativgraphen. Bachelor's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2012.

[Neu09] Sabine Neubauer. Planung energieeffizienter Routen in Straßennetzwerken. Diploma's thesis, Karlsruhe Institute of Technology, Department of Informatics, 2009

Ω Deutsche Zusammenfassung

Die Berechnung von Alternativrouten in der Navigation stellt ein sowohl interessantes, aber gleichzeitig auch ein schwieriges Problem dar. Schon die Definition der Qualität einer Alternativroute ist nicht leicht, was in der Wissenschaft zu mehreren Ansätzen geführt hat. Der am weitesten verbreitete Definitionsbegriff beschäftigt sich mit drei Hauptparametern: der lokalen Qualität einer Route, dem Anteil neuer Information im Vergleich zum wohldefinierten kürzesten Weg und der auf der Route zu erwartenden Verzögerung bezüglich der Zeitmetrik. Eine zweite Bewertungsmethode bewertet eine Menge von verschiedenen Möglichkeiten in der Form eines Graphen als Gesamteinheit. Um die Lesbarkeit der enthaltenen Entscheidungsmöglichkeiten zu erhöhen, wird die Menge dieser stark limitiert. Als Qualität wird unter diesem Limit die Varianz zwischen den Pfaden und die durchschnittliche Verlängerung über alle Pfade bewertet. Obwohl sie gewisse Verwandtschaftsbeziehungen erkennen lassen, haben die Modelle ihre eigenen Vor- und Nachteile.

Neben der reinen Qualität ist die darüber hinaus die Berechnung eine große algorithmische Herausforderung. Moderne Techniken zur Berechnung kürzester Wege setzen auf einen zwei- bis dreistufigen Ansatz. Dazu werden unter der Annahme eines statischen Straßennetzes Informationen vorberechnet um im Nachhinein die Berechnung einzelner kürzester Wege stark zu beschleunigen. Dazu eingesetzte Algorithmen schaffen es dabei die Menge unnötigerweise betrachteter Informationen sehr stark einzuschränken. Durch diese Reduktion ist die Möglichkeit, Alternativrouten in den damit berechneten Daten zu finden, natürlich stark eingeschränkt.

In dieser Arbeit haben wir verschiedene Ansätze entwickelt und untersucht, die in der Lage sind, auch mit modernen Beschleunigungstechniken, Alternativrouten zu berechnen. Dabei haben wir ein breites Spektrum an Möglichkeiten untersucht. Unsere Arbeiten erstrecken sich von der parallelen Implementierung einer bikriteriellen Suche über neue Ansätze für bekannte Verfahren bis zu ersten Implementierungen für bisher nur theoretisch untersuchte Probleme und vorher nicht betrachtete Ansätze.

Bikreterielle Suche: Für die bikriterielle Suche haben wir eine effiziente Implementierung eines von Sanders und Mandow [SM13] vorgestellten Algorithmus entwickelt. Die Implementierung basiert auf einer neuen Datenstruktur, der sogenannten Paretoqueue, die parallele Operationen für das Löschen und Einfügen mehrerer Labels in einem Schritt erlaubt. Diese auf einem B-Baum basierende Struktur ermöglichte uns das Erreichen von sehr guten Beschleunigungen und einer hohen Effizienz. Natürlich ist die Problemstruktur relevant, was eine gewisse Komplexität in der Eingabe für eine gute Beschleunigung voraussetzt. Für schwere Instanzen ermöglicht unsere Implementierung deutliche Vorteile. Für die Berechnung auf Straßennetzen kontinentaler Größe ist die Laufzeit allerdings auch mit unseren Optimierungen noch zu langsam. Durch das wiederholte Auftreten des Problems in anderen Aufgabenstellungen als der Berechnung von Alternativrouten stellt unsere Entwicklung dennoch eine Bereicherung der Welt der Algorithmik dar.

Routenkorridore: In unserer zweiten Arbeit haben wir uns mit der Berechnung von Routenkorridoren beschäftigt. Diese sind ursprünglich darauf ausgelegt einen Fahrer in einem hybriden Navigationsszenario – ein Szenario, in dem die eigentliche Berechnungskapazität bei einem Server liegt und das mobile Gerät als Anzeigegerät genutzt wird – eine gewisse Fehlertoleranz zu ermöglichen, indem sie neben dem kürzesten Weg auch eine geringe Menge an zusätzlicher Information übermitteln. Wir haben verschiedene Methoden auf der Basis von CHs entwickelt und konnten damit eine große Robustheit gegen Fahrfehler erreichen. In dem Szenario der Alternativrouten können diese berechneten Korridore als Möglichkeit zur Vorauswahl eines Subgraphen genutzt werden, auf dem dann teure Algorithmen zur Berechnung der eigentlichen Routen ausgeführt werden. Obwohl die Berechnung durchaus Routen hoher Qualität ergeben hat, war durch die Art der Korridore die reine Häufigkeit, in der Alternativrouten gefunden werden konnten, deutlich geringer als bei unseren weiteren Methoden.

Via-Node-Alternativen: Auf CHs beruht auch eine weitere Technik, die wir betrachtet haben. Eine Möglichkeit zur Berechnung von Alternativrouten ist das sogenannte Via-Node-Verfahren. Dabei wird eine Alternativroute durch die Angabe eines weiteren Punktes, des sogenannten Via-Node, beschrieben. Die Route selbst besteht dann aus der Konkatenation der kürzesten Wege vom Start zu diesem Knoten und von dort zum Ziel. Neben der Auswahl möglicher Knoten, wofür schon eine schwer zu motivierende Heuristik benötigt wird, ist der Test dieser Kandidaten auf ihre Qualität ein teurer Prozess. Im Endeffekt können nur Heuristiken genutzt werden, für die bereits die Berechnung mehrerer Wege benötigt wird. In unserer Implementierung dieses Verfahrens nutzen wir die Möglichkeit aus, dass sich CHs dahingehend erweitern lassen, dass sie die Wege zu mehreren Zielen auf einmal berechnenen. Diese Erweiterung versetzt uns in die Lage einen kompakten Graphen zu erstellen, der in kurzer Zeit dafür genutzt werden kann Kandidaten schnell auf ihre Qualität hin zu untersuchen.

Damit ist unser Verfahren praktisch unabhängig von dem genutzten Auswahlverfahren und kann daher bessere Alternativen in vergleichbarer Zeit finden.

Penalisierung: In zwei weiteren Verfahren haben wir uns mit einem dualen Problem zu den Via-Node-Alternativen beschäftigt. Dazu können Strafkosten auf den Weg verteilt werden, sei es durch kleinere Skalierungen oder gar durch komplette Blockierung einzelner Teilstellen. Diese Veränderung des Graphen widerspricht der Annahme eines statischen Straßennetzes und benötigt eine andere Herangehensweise als bisherige Algorithmen. In diesem Zusammenhang haben wir sowohl eine auf GPGPUs arbeitende Methode entwickelt, Vorberechnungen auf nahezu Echtzeit zu beschleunigen, zum anderen haben wir ein bestehendes Verfahren, CRP, so weit optimiert und an unsere Zwecke angepasst, dass selbst Anfragen für lange Wege effizient behandelt werden können. Die Beschränkung auf das Blockieren einzelner Knoten kann dabei die einzelnen Strafkosten noch stärker lokal eingrenzen, so dass das Verfahren insgesamt mit wenig Parallelität kompetitiv bleibt.

Anwendung: In einer Anwendung dieser Techniken haben wir die Möglichkeit betrachtet, in wie weit gute Alternativrouten dazu benutzt werden können schwierige Algorithmen in der Routenplanung auf eine approximative Art zu beschleunigen. Die Theorie hinter dieser Anwendung ist darin begründet, dass die Reisezeit ein entscheidender Faktor in allen Arten von kürzesten Wegen ist. Wir haben dafür das sogenannte Stochastic On-Time Arrival Problem untersucht, in dem unter der Annahme von zufallsverteilten Reisezeiten eine optimale Strategie entwickelt wird um die Chance, innerhalb eines gewissen Zeitbudgets sein Ziel zu erreichen, maximiert wird. Auch hier konnten wir experimentell zeigen, dass die Beschränkung auf Alternativrouten in der Berechnung trotz fehlender theoretischer Garantien in verschiedensten Verteilungsmodellen unerwartet gute Qualität bietet.

Schlussfolgerungen: Zusammengefasst gesagt haben wir in dieser Arbeit ein weites Spektrum an Algorithmen untersucht. Diese umspannen verschiedene Aspekte wie parallele Datenstrukturen, GPGPU-Programmierung und intelligente Variationen vorhandener Algorithmen. Die von uns vorgestellten Untersuchungen ermöglichen eine fundierte Entscheidung über Alternativrouten. Ähnlich wie in den Alternativrouten selbst können wir allerdings keine Technik als *das Mittel* der Wahl präsentieren. So wie das Grundproblem bieten auch alle Algorithmen ihre Vor- und Nachteile.

Appendix A

A Additional Information on Inputs

To allow for a better understanding of the presented results, we supply a series of additional plots, graphics, and tables. In Figure A.1, we visualize the graph instances that were used throughout this thesis. Especially for the European road networks, the drawing give a much clearer understanding of the contained regions.

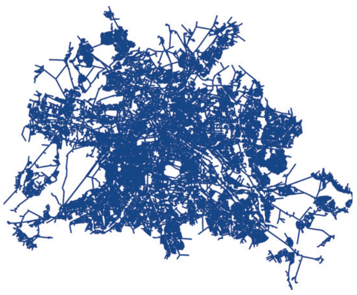
For our CRP implementation, we offer a visual representation of the partition in Figure A.2.

The CRP-π algorithm operates using a subdivision of the partition shown in Figure A.2b. We present a comparison of the original and the subdivided partition in Figure A.3.

(a) *Germany (OSM)*

(b) *Germany (PTV)*

(c) *Baden Wuerttemberg*


(d) *Berlin*

(e) *Netherlands*

(f) *Europe*

(g) *Karlsruhe*

(h) *Luxembourg*

(i) *Western Europe (PTV)*

Figure A.1: *Schematics of the graph instances used in this thesis.*

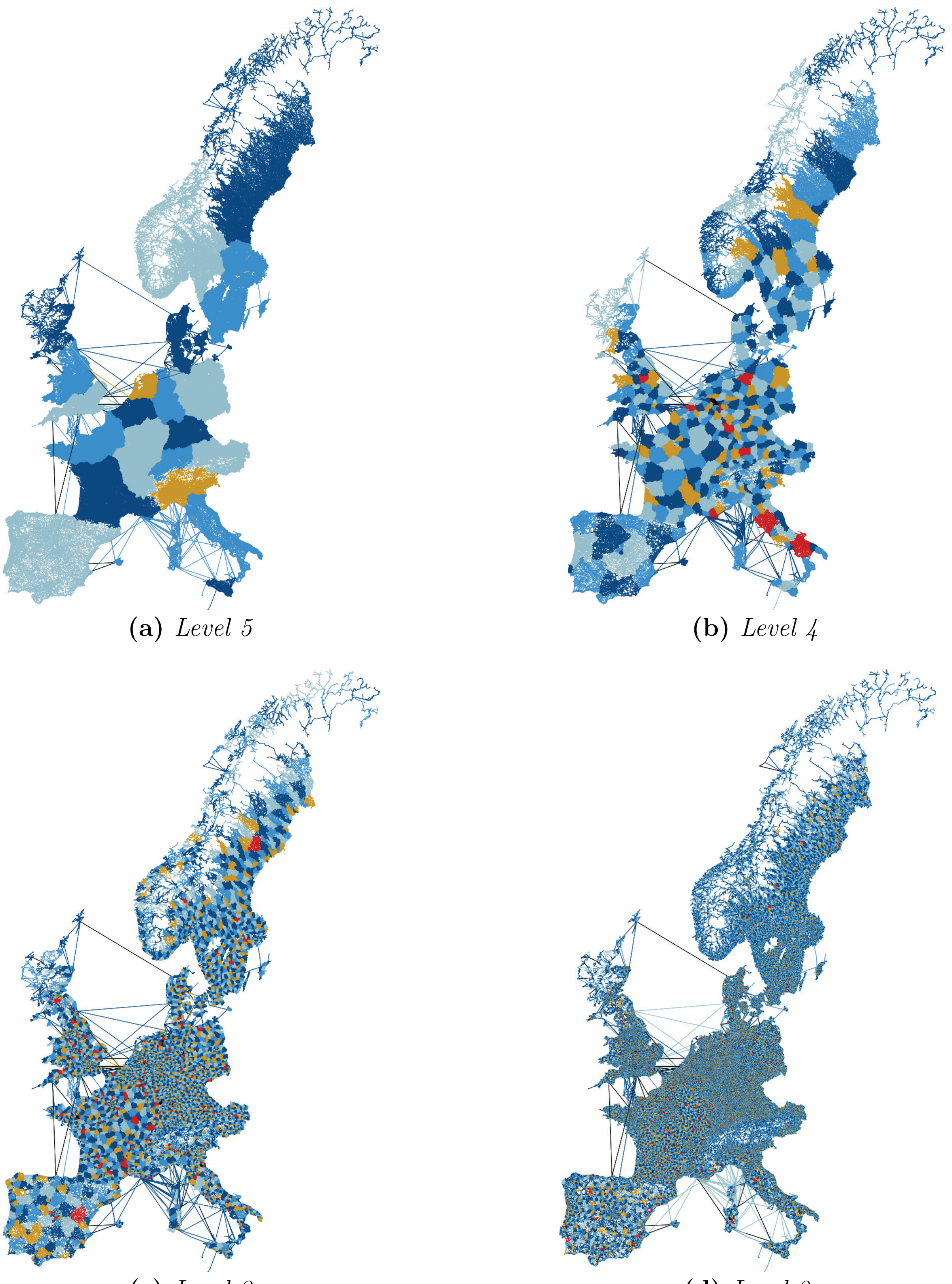
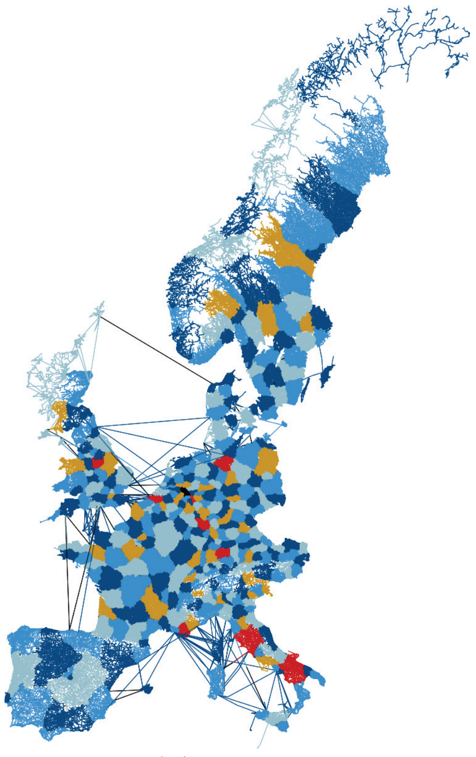

(a) *Level 5* (b) *Level 4* (c) *Level 3* (d) *Level 2*

Figure A.2: *Partitions of the Western European (PTV) road network used in Chapter 8. Lowermost level omitted due to small cell size.*


(a) *Original*

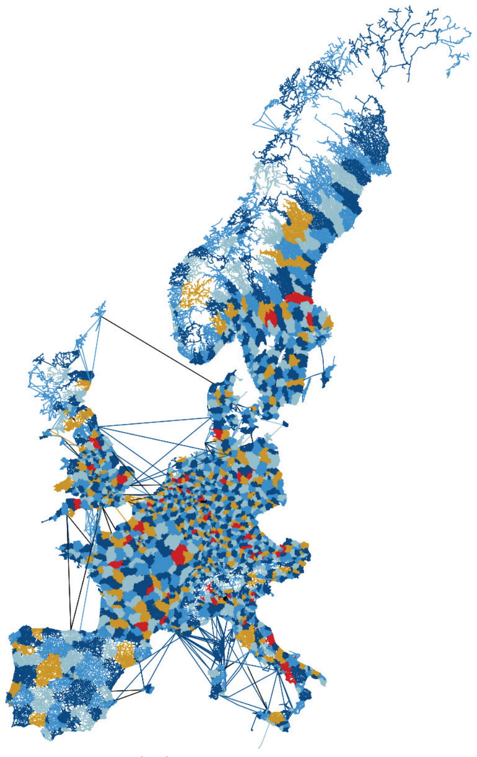

(b) *Subdivision*

Figure A.3: *Juxtaposition of the original level four partition and the one used in CRP-π.*

Appendix B

B B-Tree configuration

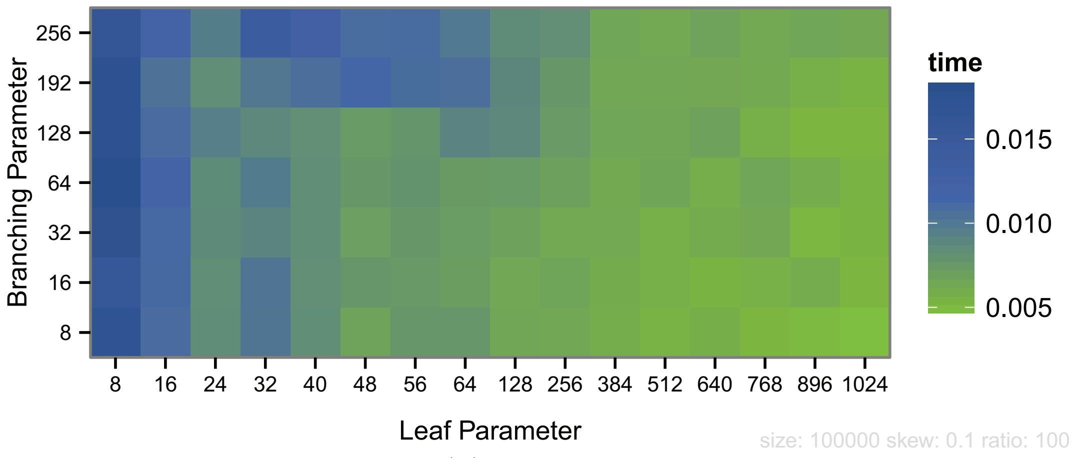
The full amount of plots generated for the configuration of our B-tree Pareto-queue is well beyond the scope of this thesis. For a better understanding of our selections, we present some further plots that focus on different potential use-cases in Figures B.1, B.2, B.3, and B.4.

B.0.1 Skewed Insertion

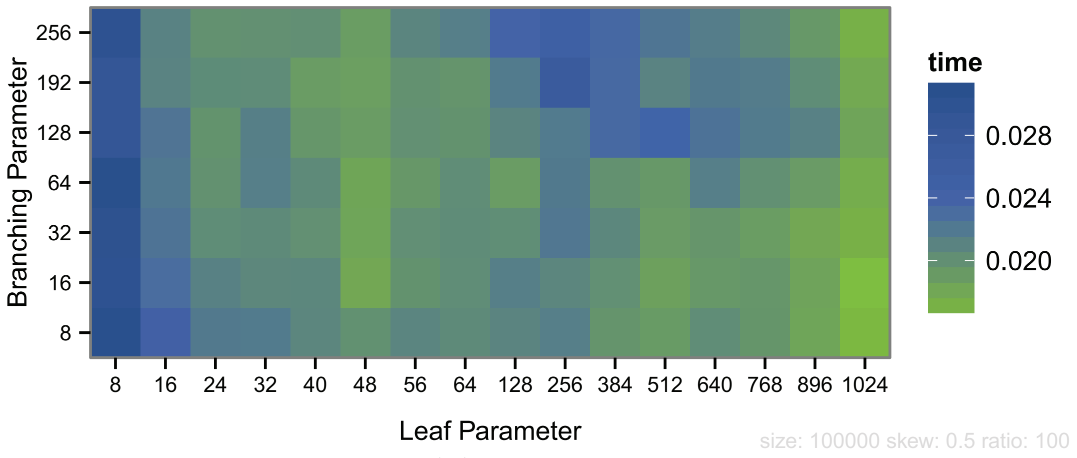
Since our tuning was initially performed on M1, we can see suboptimal behaviour for our B-tree on M2, if we consider skewed insertions. For comparison, we present Figure B.5, showing far more structured running times on M1.

B.0.2 Speed-Up

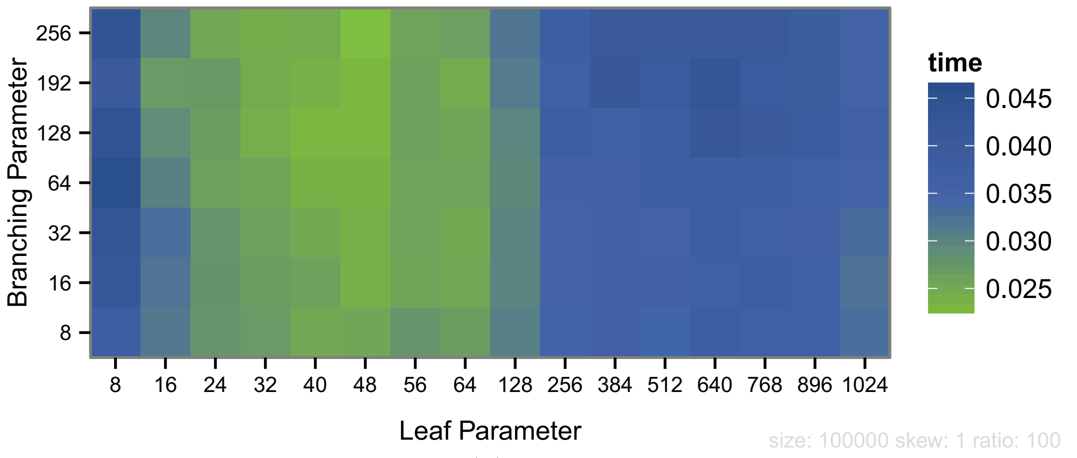
Our machine used for tuning the B-tree implementation differs from the one used in the timing measurements. As a result, we can see a drop in quality. The efficiency of our algorithm is higher on M1. We present Figure B.6 as an addition to Figure 7.11.



(a) *Strong Skew*

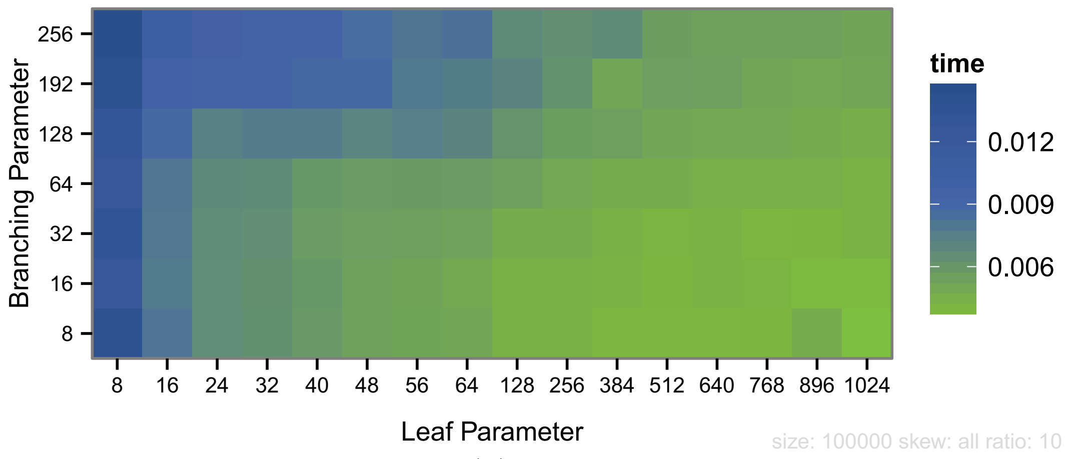


(b) *Medium Skew*

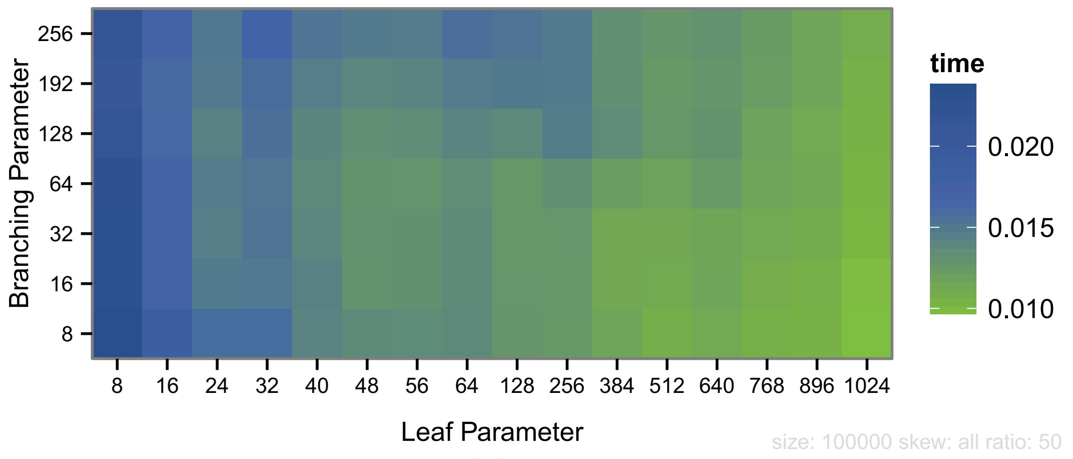


(c) *No Skew*

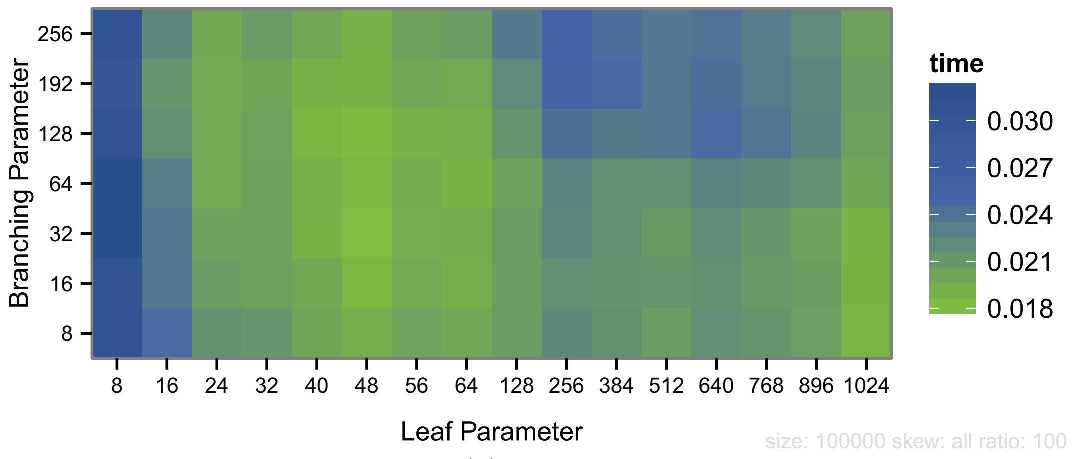
Figure B.1: *Effect of skew on spare insertions*



(a) *Ratio 10*

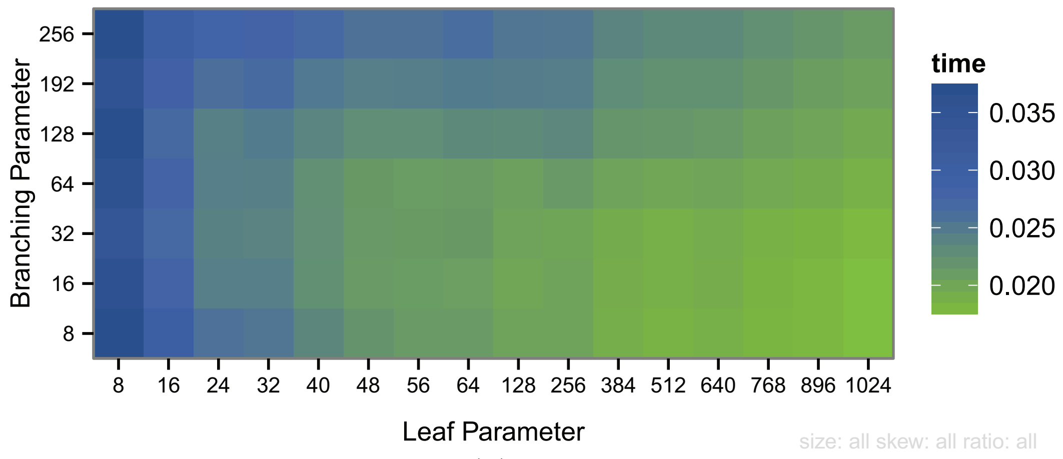


(b) *Ratio 50*

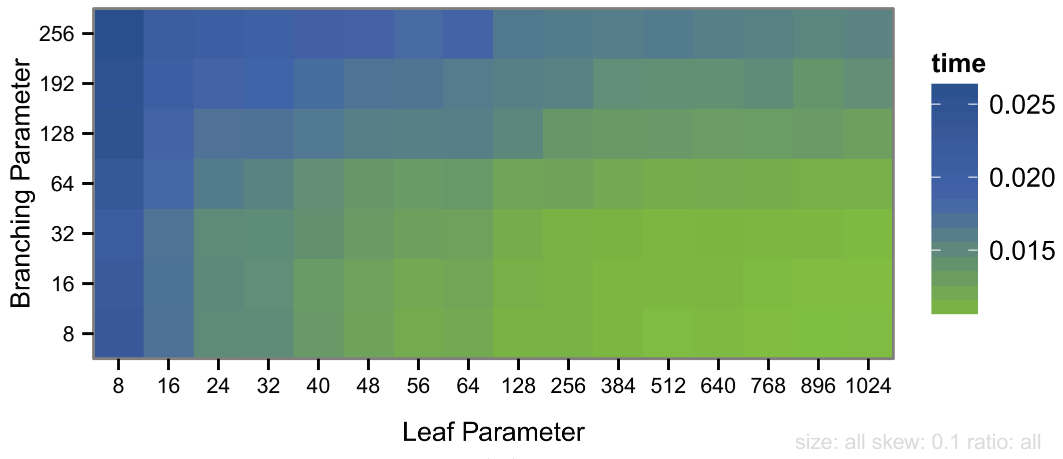


(c) *Ratio 100*

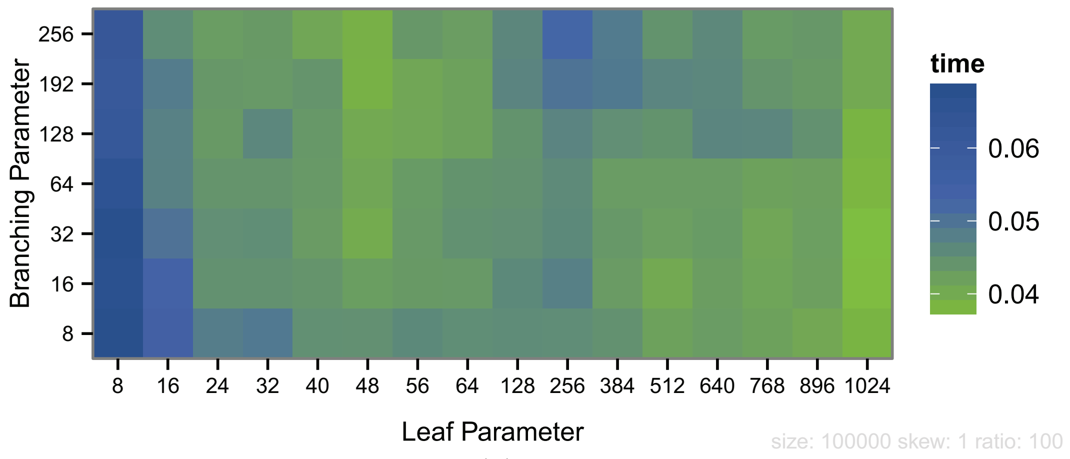
Figure B.2: *Effect of tree-sequence ratio.*



(a) *All Data*

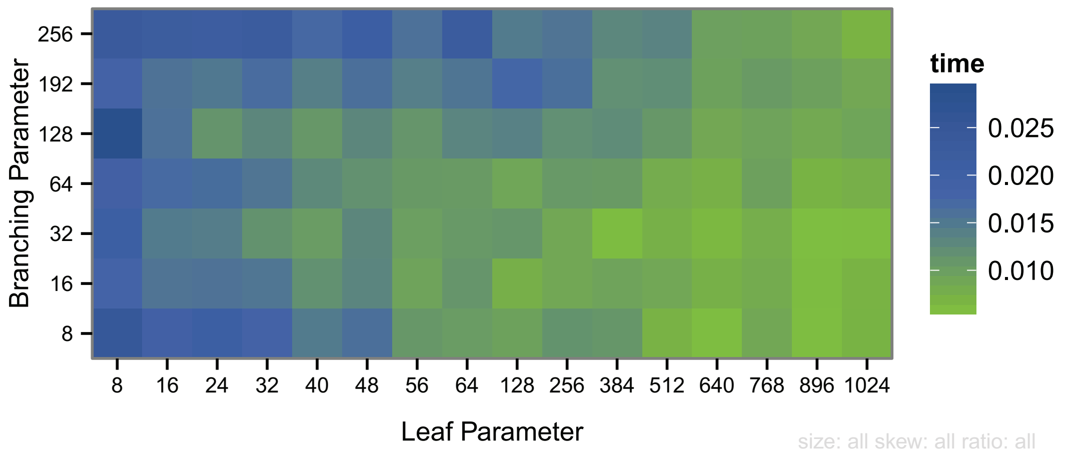


(b) *Skewed*

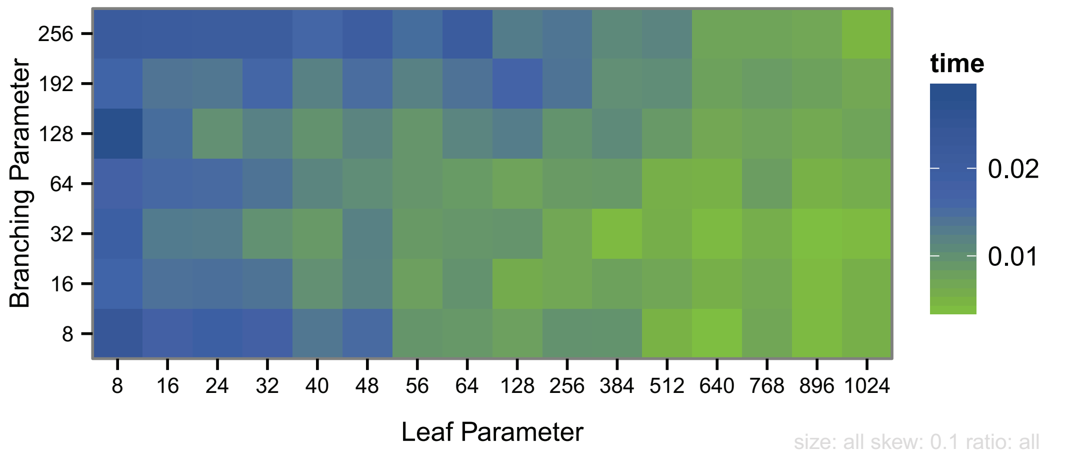


(c) *Sparse*

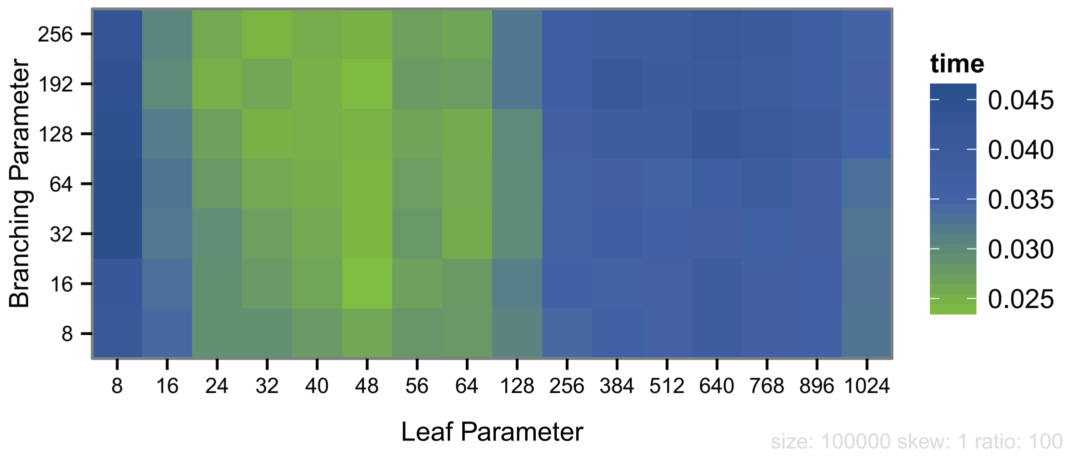
Figure B.3: *Selection of different heatmaps, showing the B-tree performance for a varying set of parameters (leaf/branching). Each plot depicts a different input characterization. Plot show mixed inputs, comparison to Figure 7.6.*



(a) *All Data*



(b) *Skewed*



(c) *Sparse*

Figure B.4: *Selection of different heatmaps, showing the B-tree performance for a varying set of parameters (leaf/branching). Each plot depicts a different input characterization. Plot show removal inputs, comparison to Figure 7.6.*

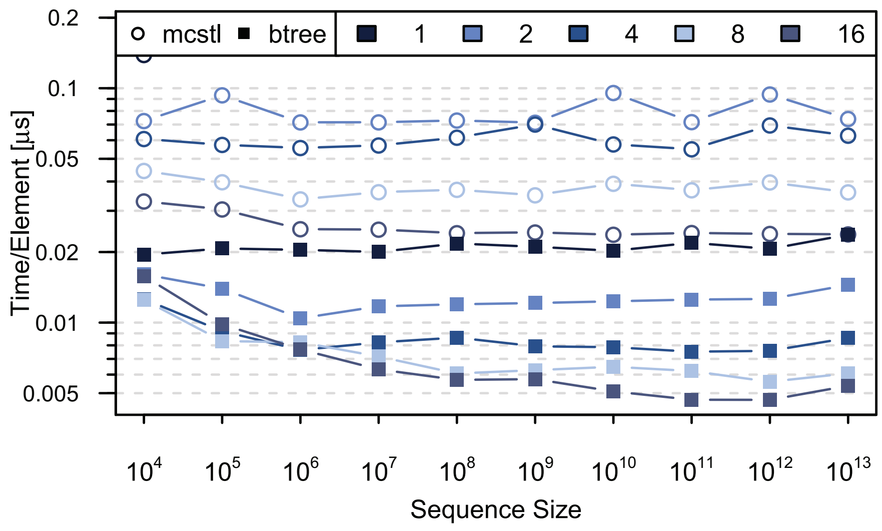
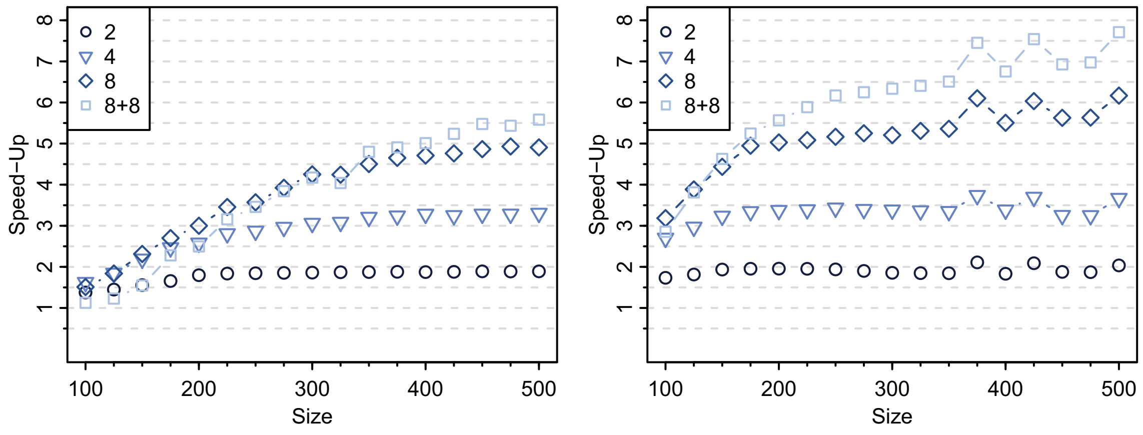


Figure B.5: *Timing measurements for for skewed insertion on M1. Insertions among the first 10 percent.*



(a) *Correlated (q=0.5)* **(b)** *Negatively Correlated (q=-0.5)*

Figure B.6: *Speed-up of parallel Pareto-optimal shortest path search over the sequential version on M1. The maximal cost of an arc is set to ten.*

Appendix C

C Continued Measurements on Parallel Pareto-search

We present speed-up values for the road instances of Machuca et al.. In addition to Figure 7.11, we provide two additional correlations for comparison in Figure C.2.

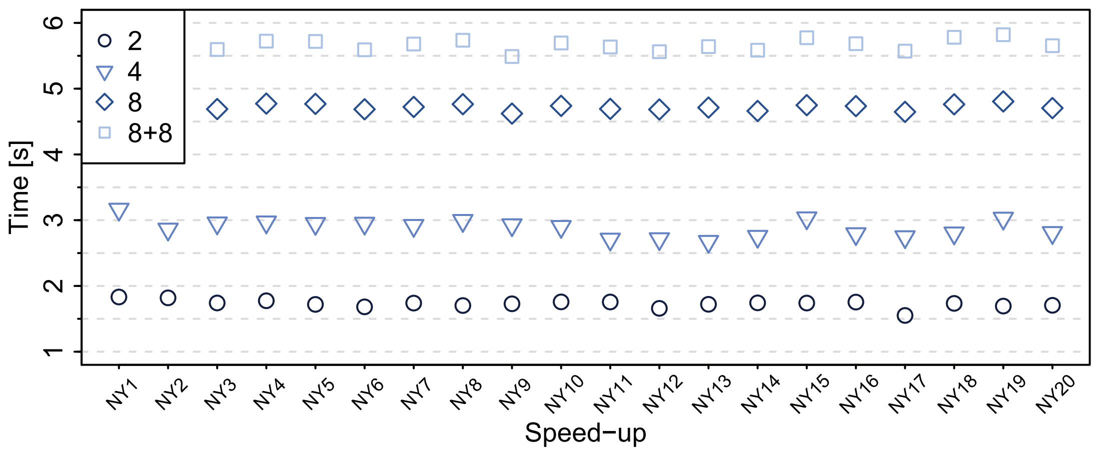
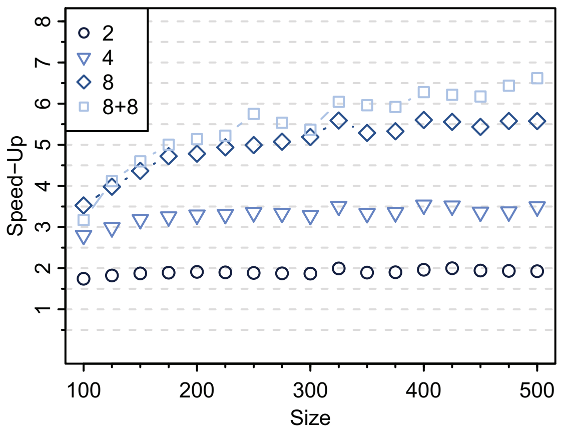
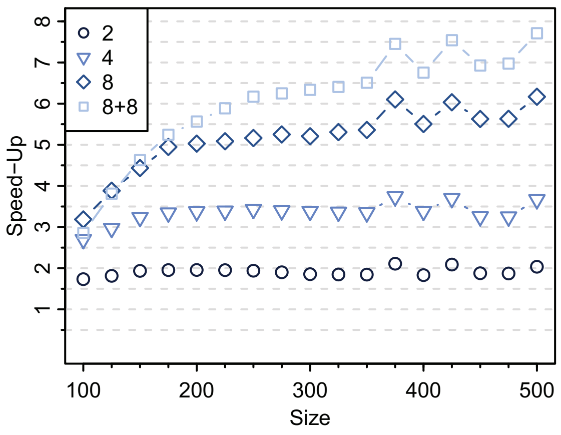


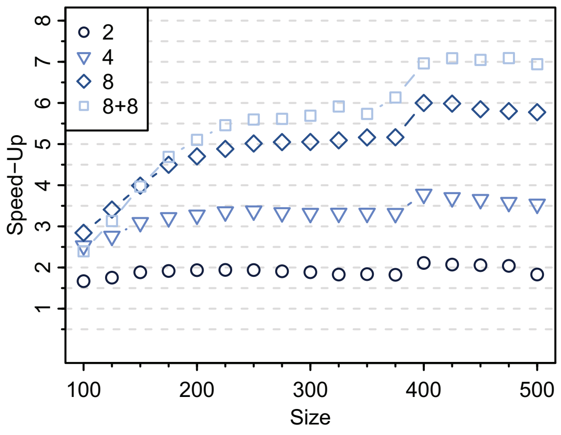
Figure C.1: *Speed-up of parallel Pareto-optimal shortest path search over the sequential version on M1 on the New-York instances.*



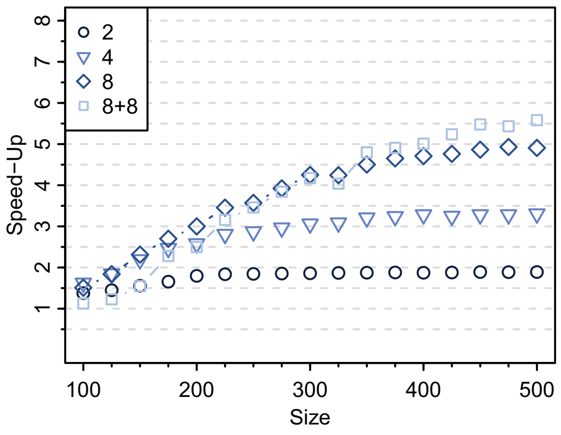
(a) *Negatively Correlated (q=-0.9)*



(b) *Negatively Correlated (q=-0.5)*



(c) *Uncorrelated (q=0)*



(d) *Correlated (q=0.5)*

Figure C.2: *Speed-up of parallel Pareto-optimal shortest path search over the sequential version on M1. The maximal cost of an arc is set to ten. For highly correlated values we barely experience any speed-up. Our standard configuration performs better in the general case than in the scenario with strongly negatively correlated data.*

Appendix D

D Extended Considerations for the Penalty Method

To mark dirty cells, we use Algorithm D.1.

Configuration. We tested a range of different setups for the implementation of CRP-π. Over all of these setups, we visualize the best value for the total distance and every Dijkstra rank in Figure D.1. Labels along the points mark a change in the best technique. A label encodes the type of path penalty (exp = exponential, lin = linear scaling), MA and SA distinguish between singular addition (SA) of rejoin penalty and multiple application of the rejoin penalty (MA). The following numbers specify the chosen parameters for the penalty, starting with the path penalty. The second number specifies the rejoin penalty value. To account for the difference in decision arcs, we tested multiple variations of scaling the respective values for the total distance. The overall differences in penalization are minimal, though. Therefore, we were able to chose a uniform penalization scheme.

Timing. For comparison, we also present values on the parallel execution of the penalization on our machine M1 in Figure D.2.

D.1 Iteration Configuration

Chapter 8 introduced and evolved a way of performing penalized queries on continental-sized networks in a fast and interactive way. We propose a setup that omits the cells that contain either the source or the target during the update process. In addition, an efficient implementation should use less than the originally proposed five levels and especially utilize a smaller than usual cell size for the fourth level. As previously mentioned, we use a combination of *Microcode*, the algorithm of Floyd and Warshall for updating the different cells on the lower levels. For the topmost two levels, we use

Algorithm D.1 Mark Dirty Cells

Input: A graph $G(V, A)$, preprocessed for CRP with associated metric μ, a source s and a target t, and a path P between s and t.
Output: A list of dirty cells

```
1:  $\mathcal{Q} = \emptyset$
2:  $\mathcal{D} = \emptyset$
3:  for all $v \in P$ do
4:      if $cell(v) \neq cell(s) \wedge cell(v) \neq cell(t) \wedge !\mathcal{Q}.contains(cell(v))$ then
5:          $\mathcal{Q}.push(cell(v))$                                   ▷ for next iteration
6:          $\mathcal{D}.push(cell(v))$                                   ▷ for output
7:      end if
8:  end for
9:  level = 1                                                             ▷ lowest level is zero
10: for level < # levels do
11:     $\mathcal{Q}' = \emptyset$
12:     for all $c \in \mathcal{Q}$ do
13:         if $!parent(c).containsAny(\{s, t\}) \wedge !\mathcal{Q}'.contains(parent(c))$ then
14:             $\mathcal{Q}'.push(parent(c))$                            ▷ for next iteration
15:             $\mathcal{D}.push(parent(c))$                             ▷ for output
16:         end if
17:         $\mathcal{Q} = \mathcal{Q}'$
18:     end for
19: end for
20: for all $c \in \mathcal{Q}$ do                                       ▷ topmost level
21:     $\mathcal{D}.push(c)$
22:     $\mathcal{Q}.remove(c)$
23: end for
24: return $\mathcal{D}$
```

a vectorized implementation of the Bellman-Ford algorithm. The number of levels to use should be determined by the range of the query. Queries of a very short range, up to Dijkstra rank 2^{10}, should be restricted to using only a single level. From rank 2^{11} through 2^{18}, we propose the use of two levels for penalization. For higher Dijkstra ranks, the number of levels depends on the allowed usage of parallel computation. In sequential execution, Dijkstra ranks 2^{19} to 2^{21} are fastest using three levels. Above this rank four levels perform best. In parallel execution, the settings depend on the number of CPUs used. We specify the proposed settings in Table D.1. To decide to which class of query a certain pair of source and target belongs, a simple lookup table suffices that considers the average distance between two pairs of vertices.

We find that omitting the source and target cells for updates results in four levels

Table D.1: *Proposed configuration of the penalization process in dependence of the number of used CPUs. The data in this table refers to M2 and specifies the maximum Dijkstra rank for which the number of levels should be used. The depicted ranks are to be understood as powers of two.*

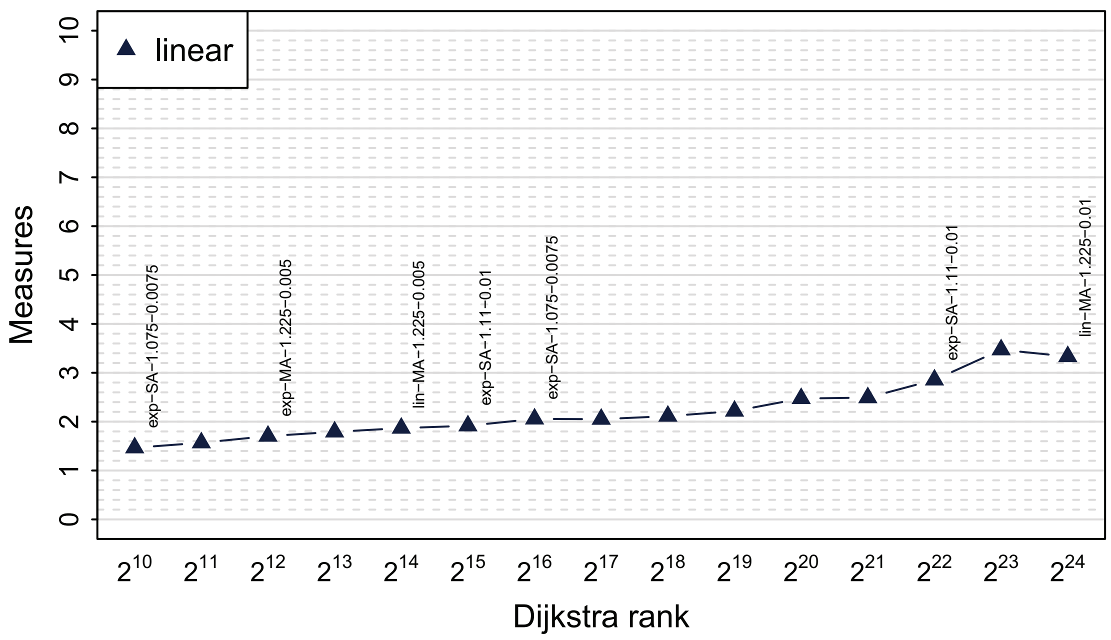
	# CPUs							
# levels	1	2	4	8	12	16	20	24
one	10	10	9	10	10	10	10	10
two	18	16	16	16	16	16	16	16
three	21	19	18	18	18	18	18	18
four	24	24	24	24	24	24	24	24

actually outperforming three levels for short- and long-range queries whereas three levels are best for medium-range queries. The overall cost of the queries is low enough to justify our ignoring this and keeping to the simple selection method depicted in Table D.1. In addition, the variation we experience for the lower Dijkstra ranks is to be expected due to the overall very brief time which is required for a single iteration.

Table D.1 shows that the variation we experience for the different number of CPUs stabilizes at four and more CPUs. Only below this threshold a variation seems to be necessary.

D.2 Upper Bound Graph

For illustration, we present a plot of the upper bound graph used in Section 8.2.



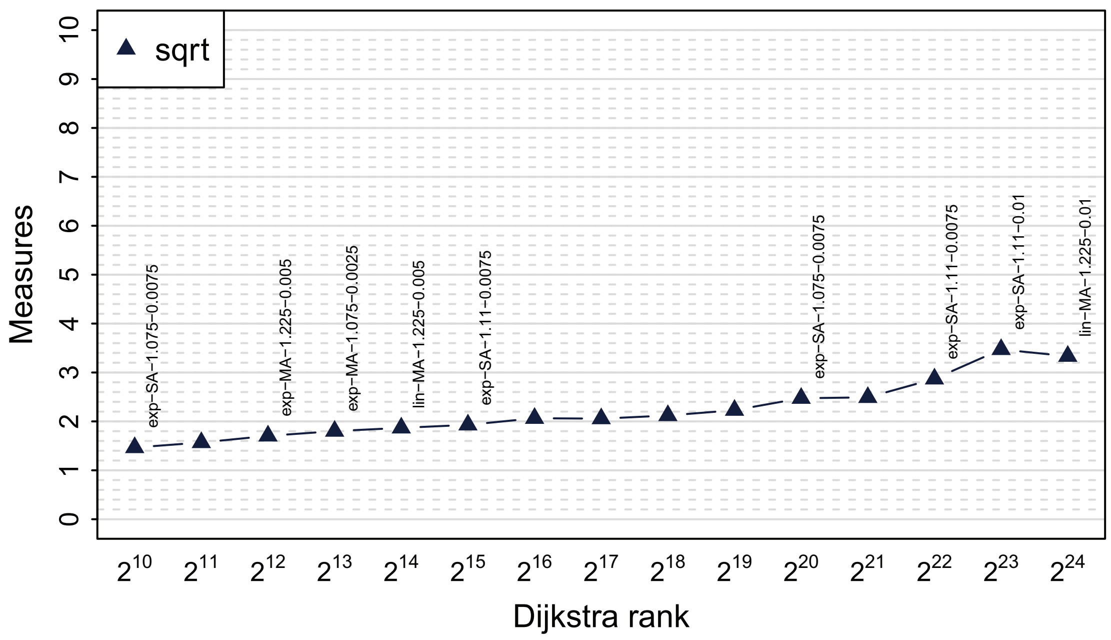
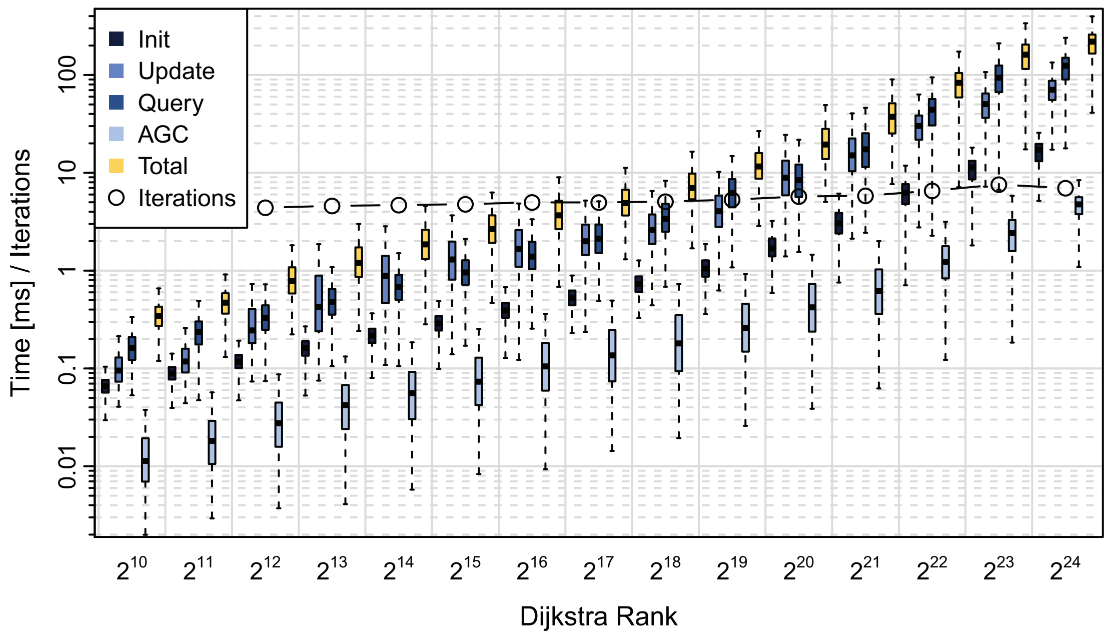
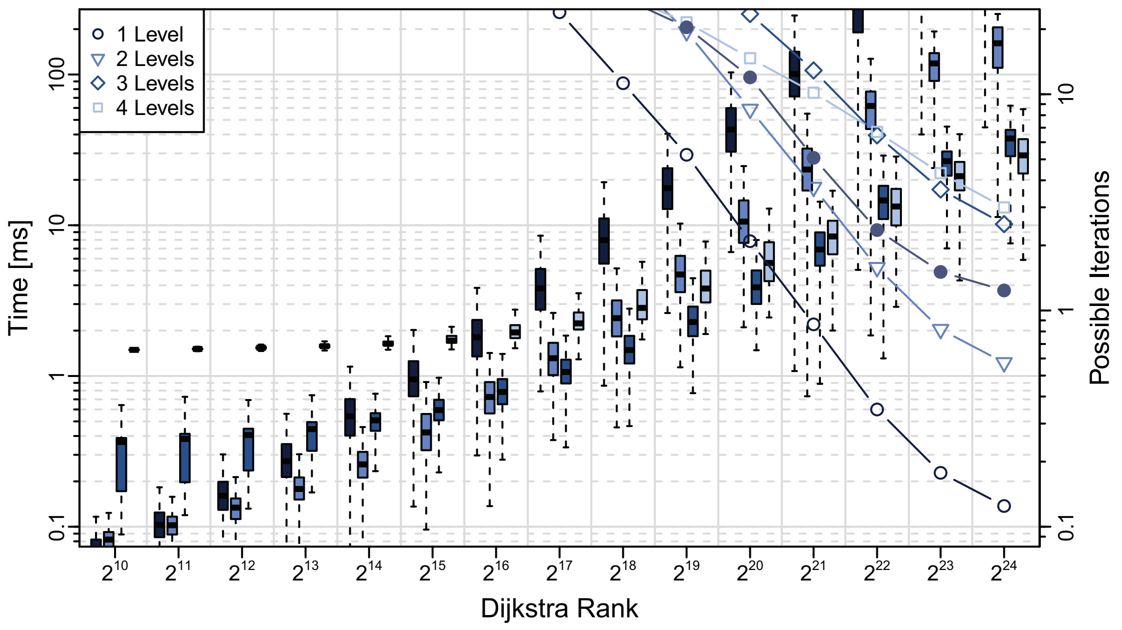


Figure D.1: *Selection method for the final parameters. The points show the different setup methods chosen.*



(a) *Additional timings on M1 as comparison to Figure 8.15a.*



(b) *Eight threads on M1*

Figure D.2: *Final configuration of CRP-π*

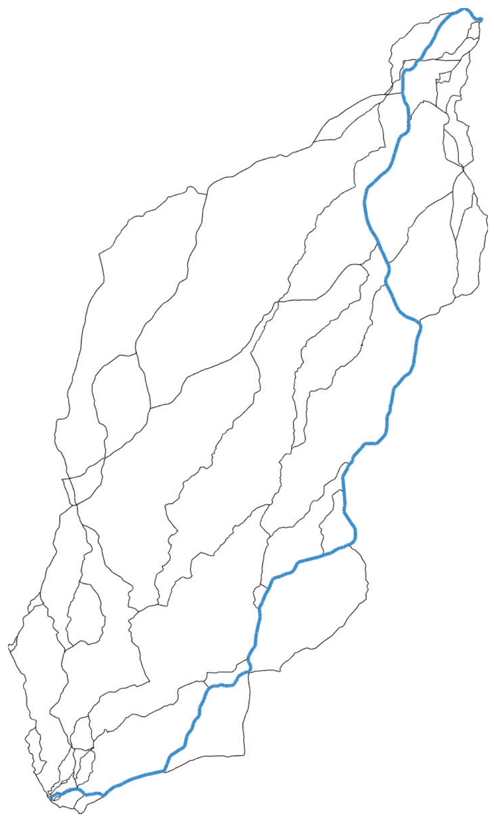


Figure D.3: *Upper bound alternative graph for the German road network. The plot illustrates a Dijkstra rank of two to the power of twenty-one.*

Appendix E

E Additional Information on Corridors

E.1 Random Drives

In addition to the studies already performed in Chapter 11, we present additional random drivers and an extended study of the random walks. The further drivers strengthen the argument that our corridors offer a sub-graph robust to deviations.

The Random Deviation Driver. The most basic driver is the *random deviation driver* (RD). He/She operates according to a single parameter p that indicates the probability for a wrong turn. A random drive by a deviation driver traverses the corridor, following the given driving directions. At every intersection, we roll a dice and, with probability p, take a wrong turn. The wrong turn itself is chosen uniformly at random from all incorrect choices available, not considering a potential u-turn.

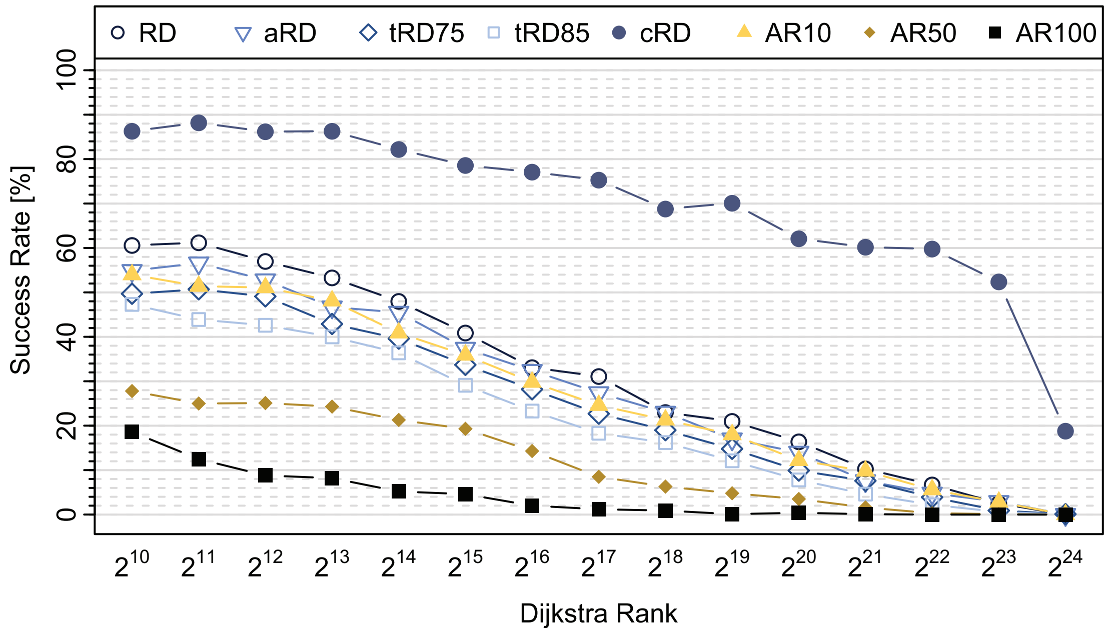
The Time-Adjusted RD Driver. Our final version that is directly derived from the RD driver. In this version, we follow the notion that a aRD driver's behavior might produce a misleading image in urban areas. In a sequence of short roads, a driver might follow the driving directions simply by chance. For our final model, we specify p, p', and a drop-off function $f(t)$. As for the aRD, this model increases the probability for an additional wrong turn from p to p' after the first encountered deviation. Afterwards, depending on the travel time t from the initial mistake, the probability of a wrong turn decreases according to f as long as the driver follows the directions. Every wrong turn resets the probability to p' and sets the error as new initial mistake. We call this model the *time-adjusted random deviation driver* ($taRD$).

E.1.1 Random Walks

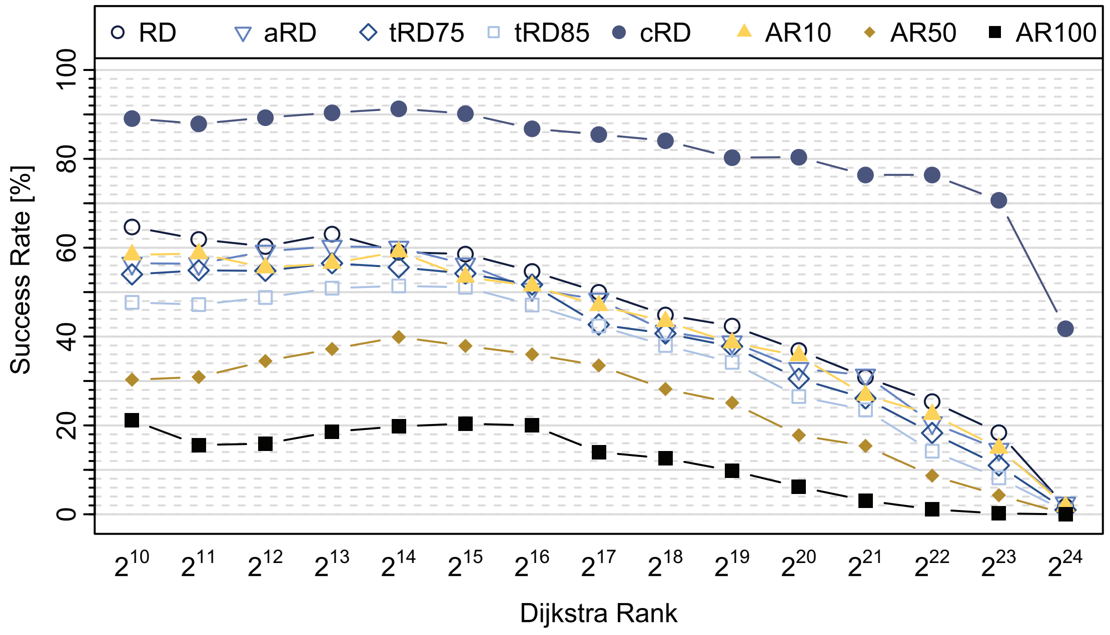
In addition to the previously presented random walks, we present additional plots showing the additional drivers. We see only little difference to the drivers already covered in Chapter 11, though. The general observations stay the same.

E.1.2 Corridor Growth

To give an impression of the growth of an (unfiltered) corridor, we present Figure E.5 and Figure E.6. We can see that the turn corridor offers some robustness along all parts of the corridor, whereas the detour corridor may contain long segments without deviation information.

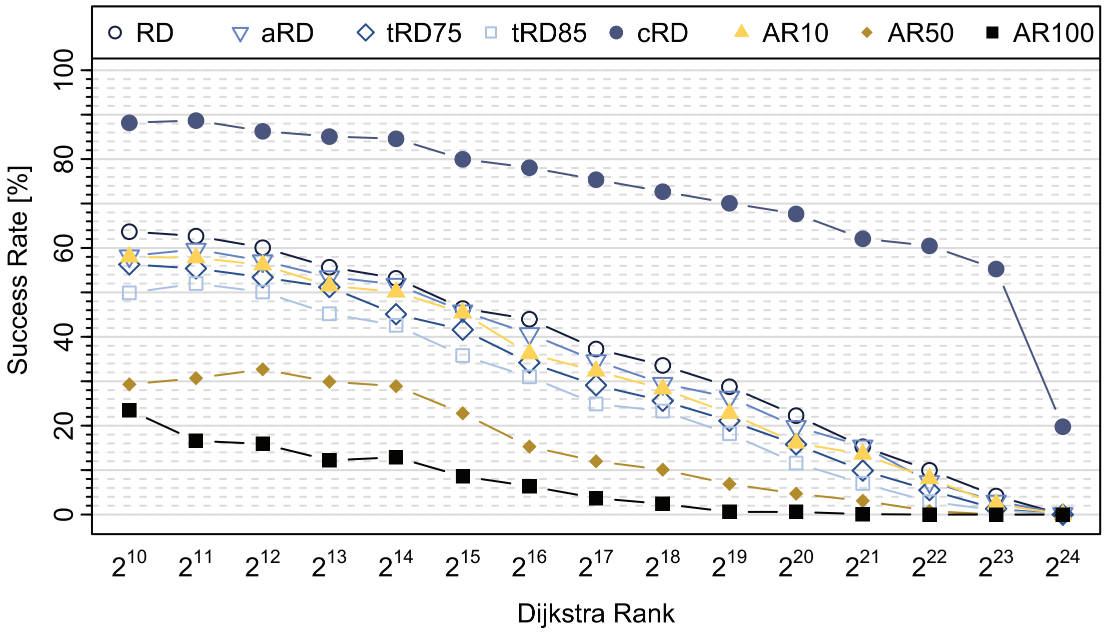


(a) $\delta_t = 100$

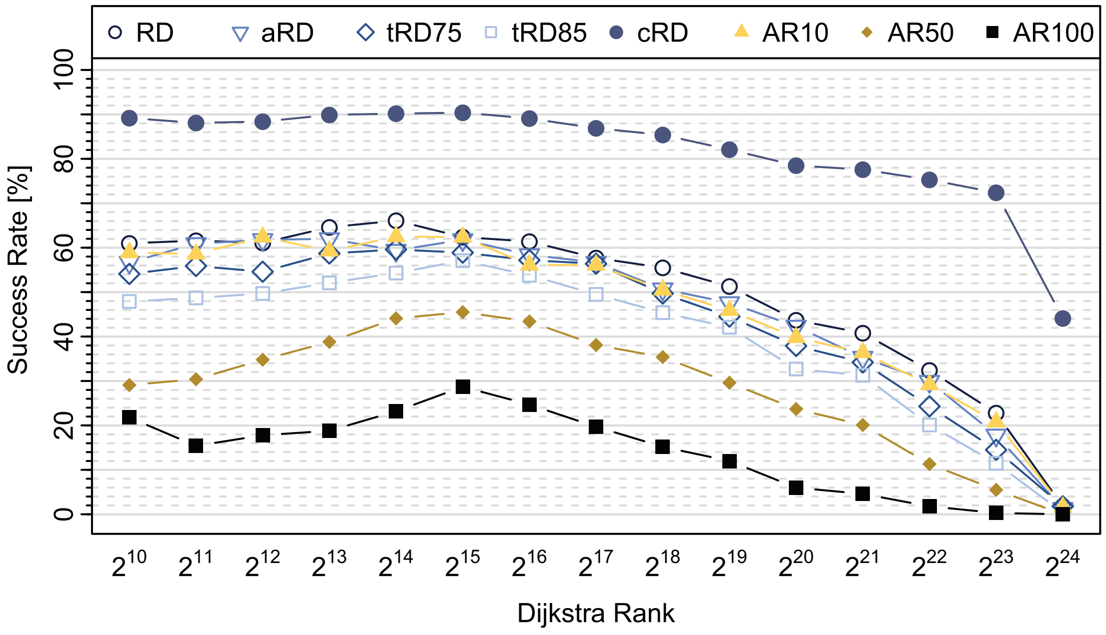


(b) $\delta_t = 200$

Figure E.1: *Success rate in relation to the length of the shortest path for a detour corridor; error probability set to ten.*

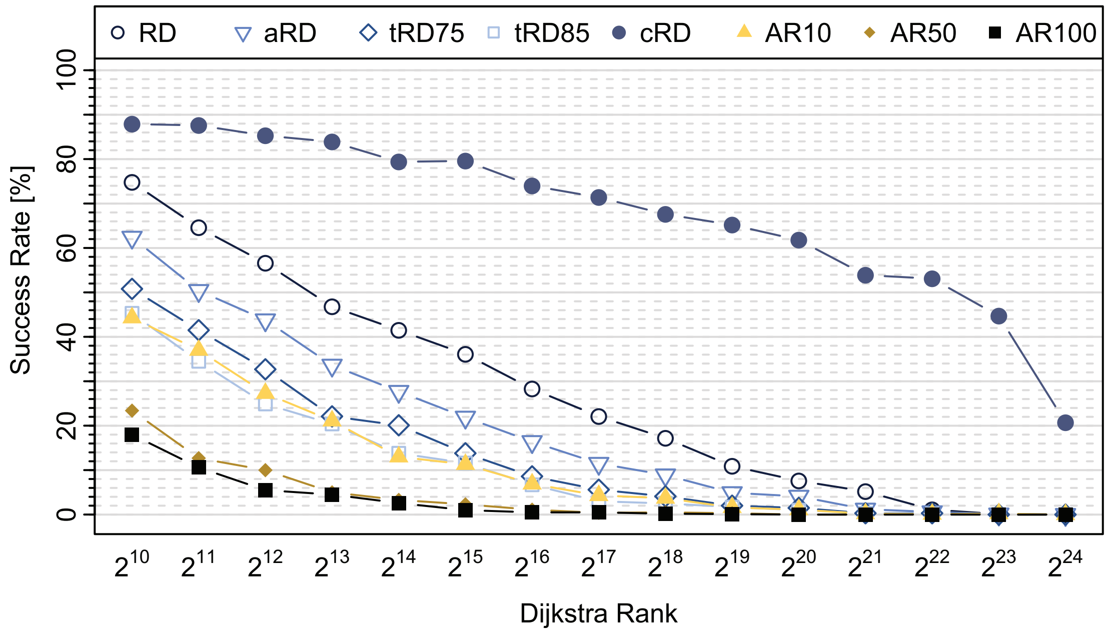


(a) $\delta_t = 100$

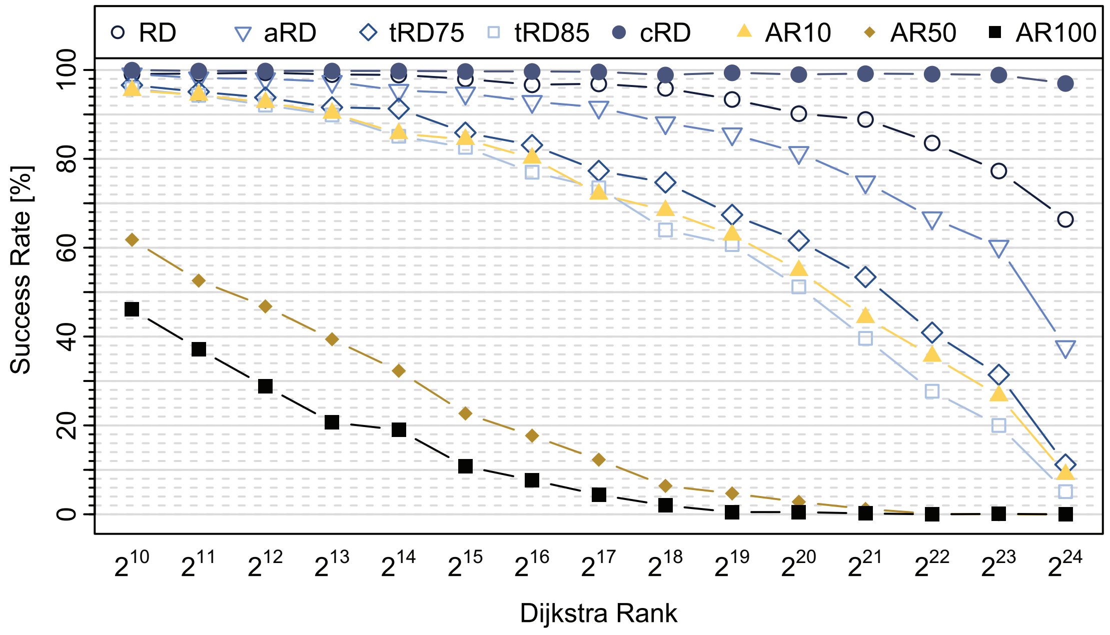


(b) $\delta_t = 200$

Figure E.2: *Success rate in relation to the length of the shortest path for a recursive detour corridor (depth of three); error probability set to ten.*

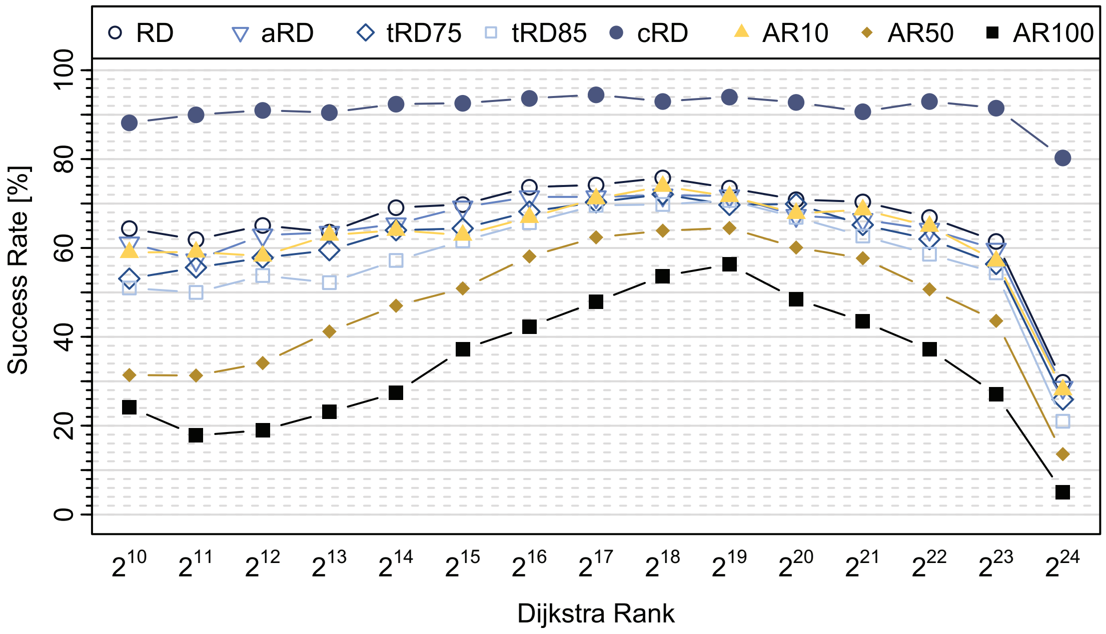


(a) *One turn*

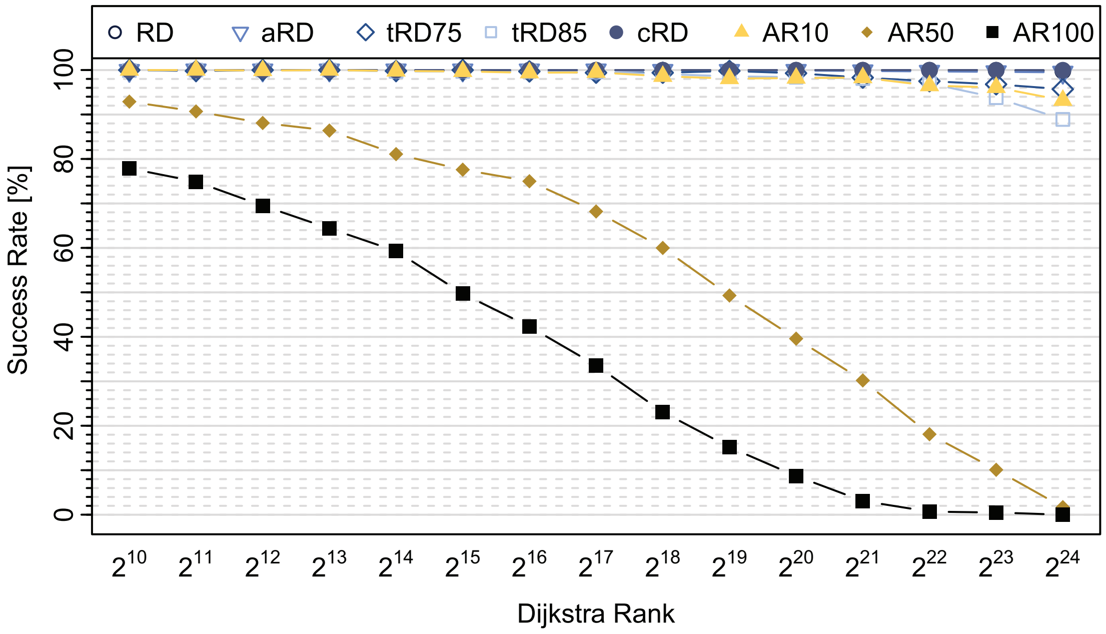


(b) *Three turns*

Figure E.3: *Success rate in relation to the length of the shortest path for a turn corridor; error probability set to ten.*

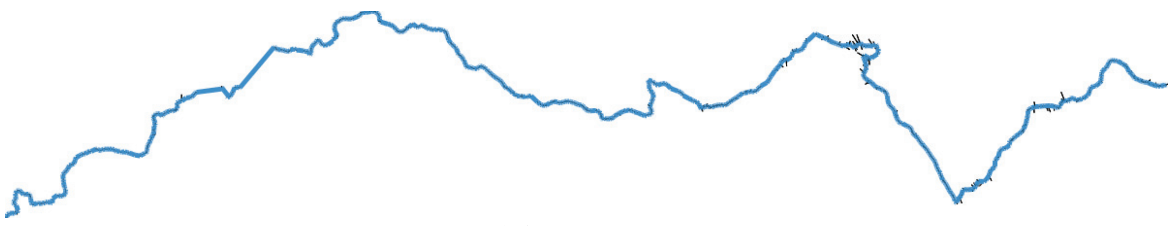


(a) *Detour corridor;* $\delta_t = 500$

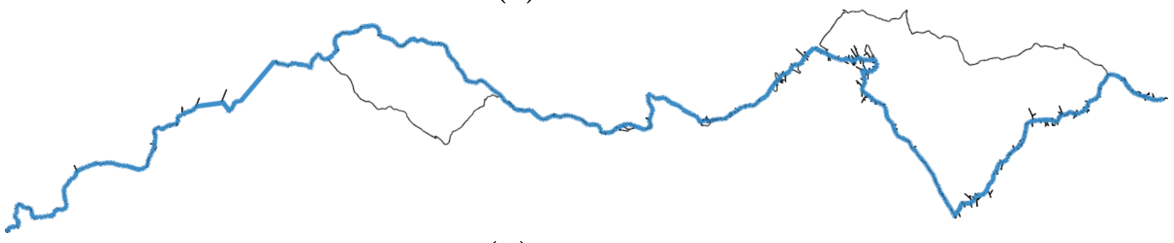


(b) *Turn corridor; six turns*

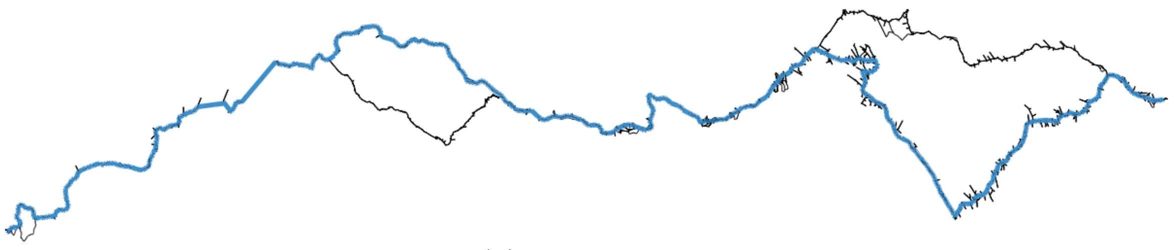
Figure E.4: *Success rate in relation to the length of the shortest path for a detour corridor; error probability set to ten.*



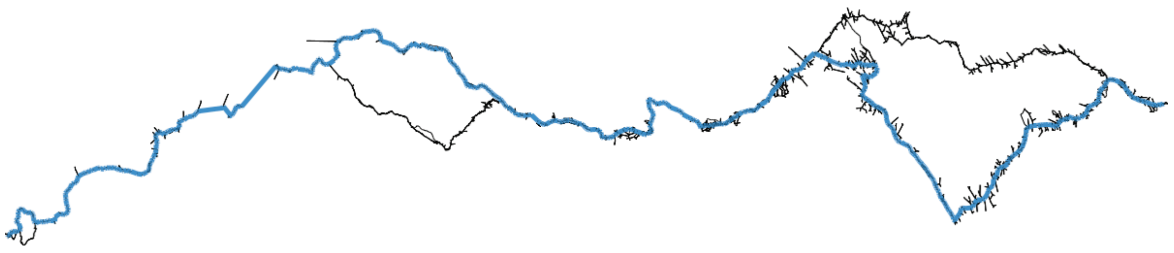
(a) *One turn*



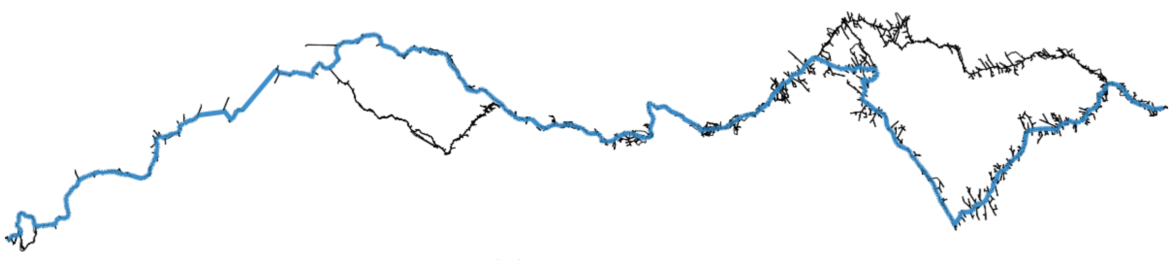
(b) *Two turns*



(c) *Three turns*

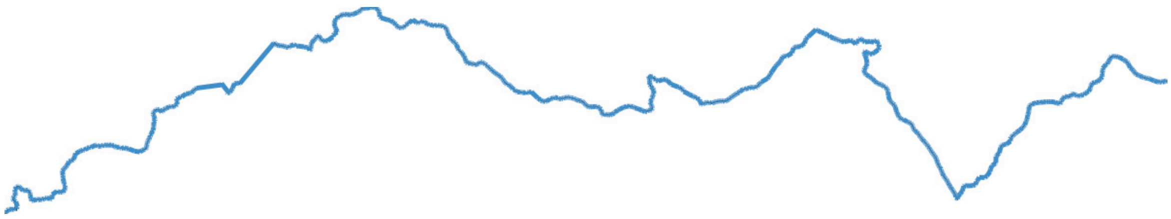


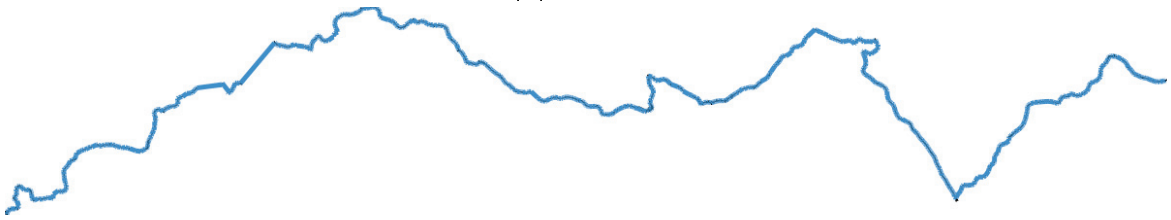
(d) *Four turns*

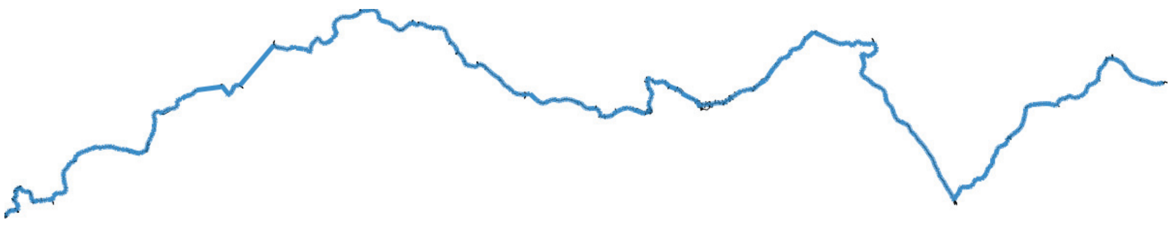


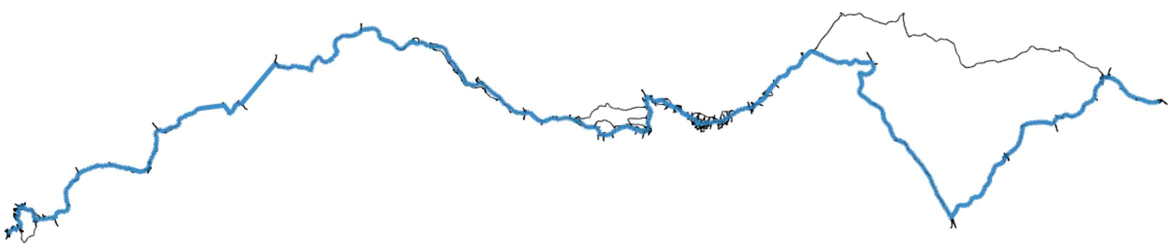
(e) *Five turns*

Figure E.5: *Turn corridor for a varying number of turns (rank 2^{19}).*


(a) $\delta_t = 10$


(b) $\delta_t = 50$


(c) $\delta_t = 100$


(d) $\delta_t = 200$

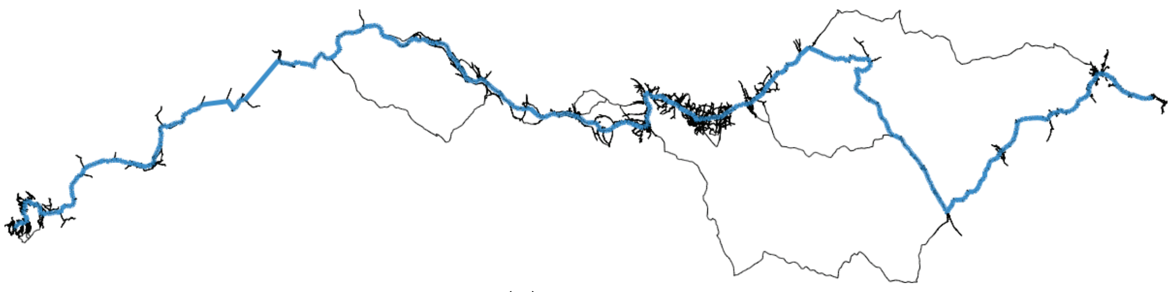
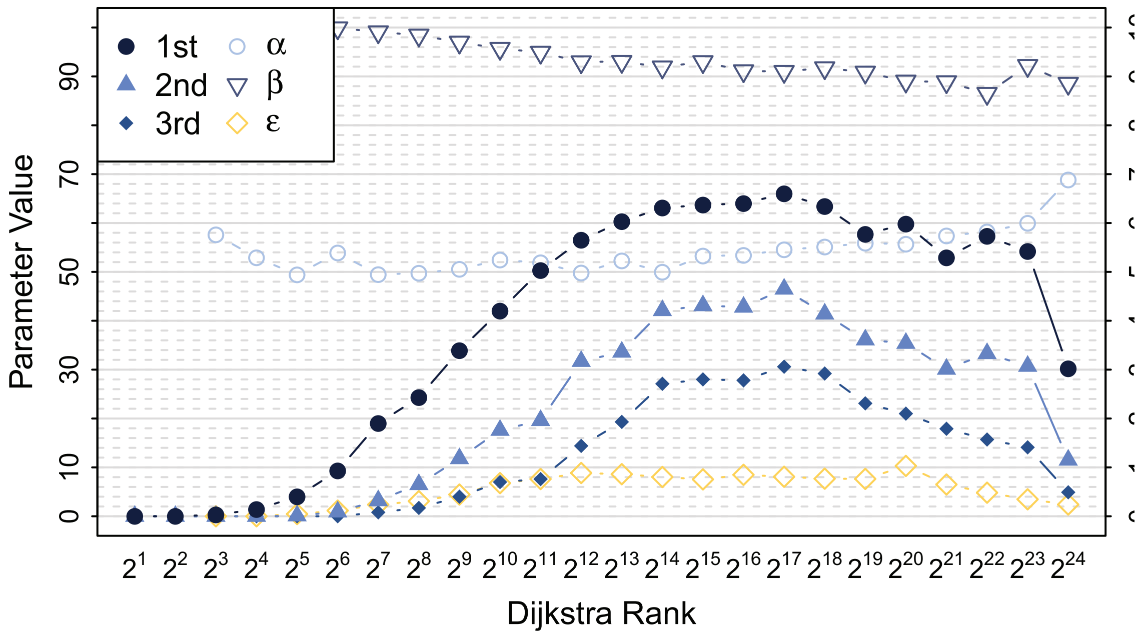

(e) $\delta_t = 500$

Figure E.6: *Detour corridor for a varying size of δ_t (rank 2^{19}).*

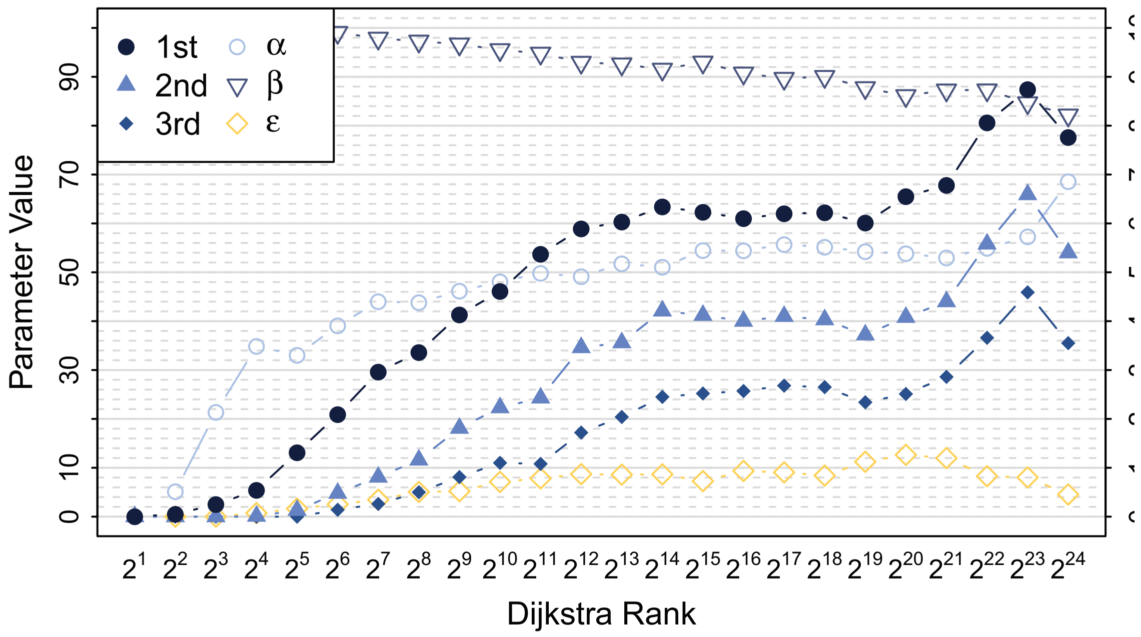
Appendix F

F Further Comparisons

For additional comparison, we present figures depicting the quality of the first alternative route as well as the success rates for the first three alternatives here. The alternative routes have been extracted using the linear combination selection strategy. We can see that penalty methods (Figure F.3) offer a the best success rate in general. The largest difference in the success rate can be found for short-range to mid-range queries. For these types of queries, even the corridors methods can be considered viable (c.f. Figure F.1). The quality, in terms of Definition 5.3 (bounded stretch (BS), local optimality (LO), limited sharing (α)) is better for the via-vertex method (c.f. Figure F.2), though.



(a) *Success rates and quality values for first alternative for a three-recursive detour corridor. Initial detour: twenty seconds.*



(b) *Success rates and quality values for first alternative for a six-turn corridor.*

Figure F.1: *Quality of our corridor approaches*

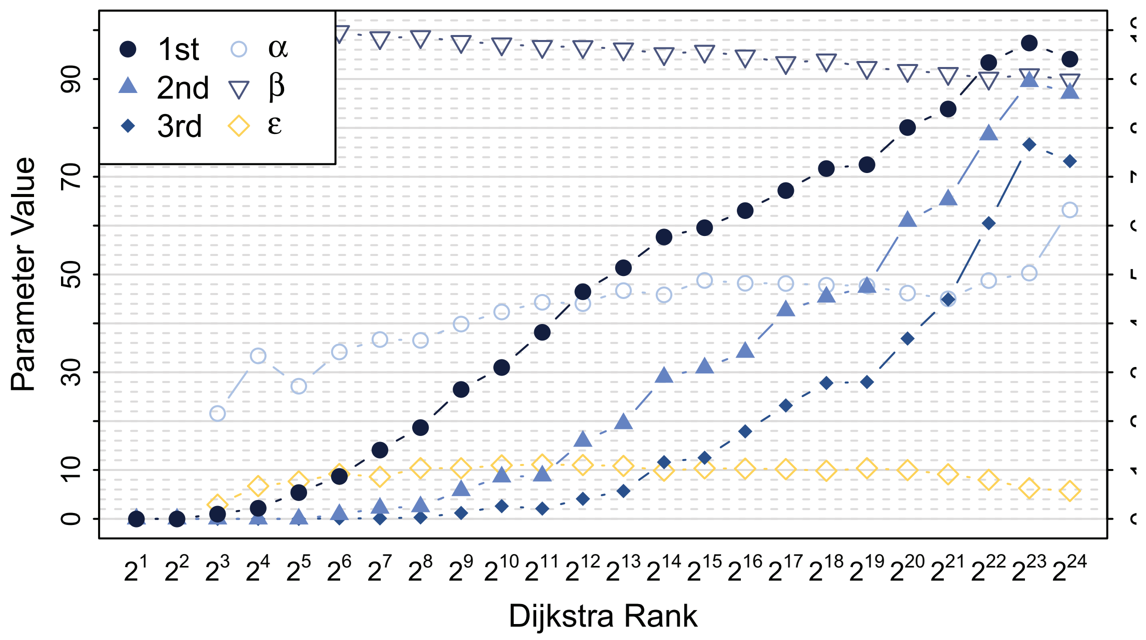
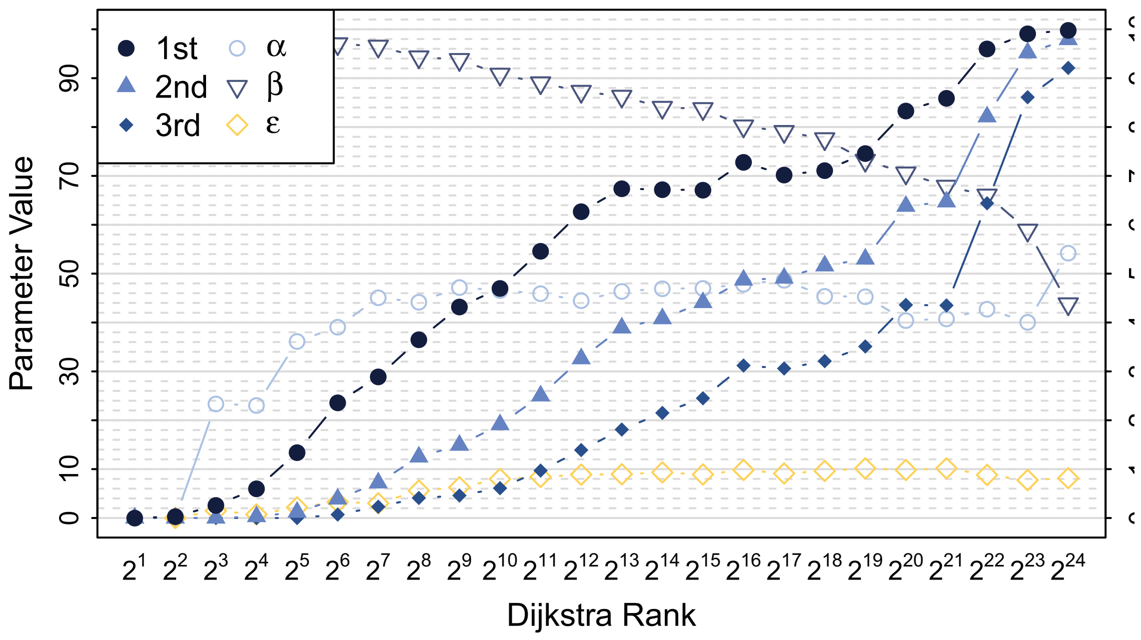
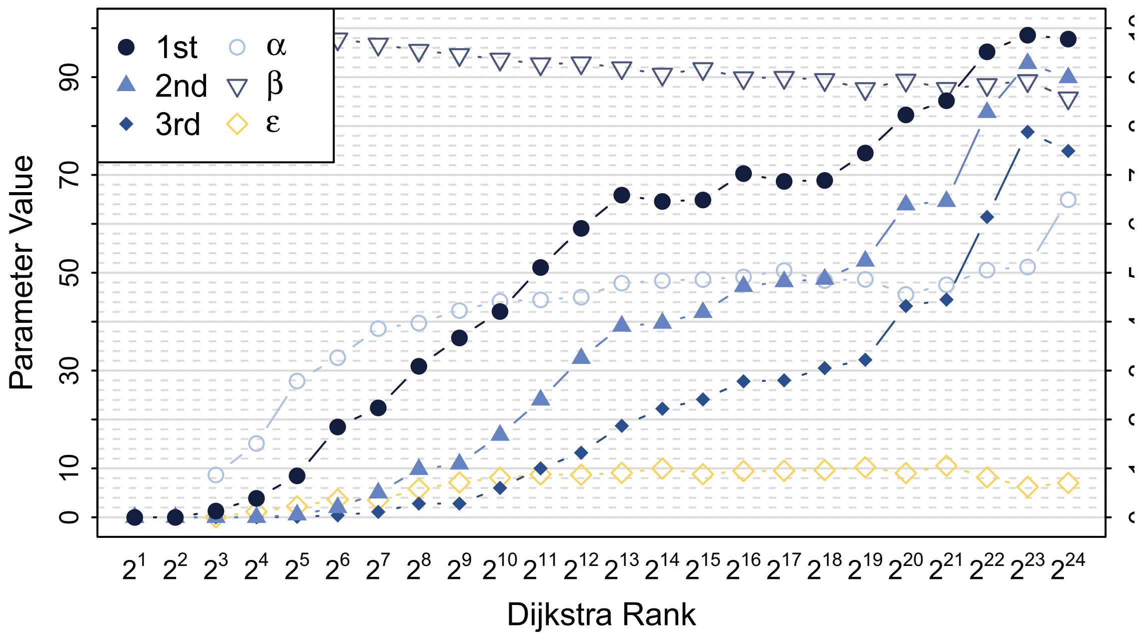


Figure F.2: *Success rates and quality values for first alternative for HiDAR.*



(a) *Success rates and quality values for first alternative for CRP-π.*



(b) *Success rates and quality values for first alternative for CRP-∞.*

Figure F.3: *Quality of our penalization approaches*