Karlsruhe Institute of Technology

Fast Computation of Isochrones in Road Networks

Master Thesis of

Valentin Buchhold

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Supervisors:	Moritz Baum, M.Sc.
	Dipl.-Inform. Julian Dibbelt

Period of Time: January 1, 2015 – June 30, 2015

Author's Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work. This thesis has not been submitted either in whole or part, for a degree at this or any other university or institution. This is to certify that the printed version is equivalent to the submitted electronic one.

Karlsruhe, June 30, 2015

Valentin Buchhold

Abstract

We study the problem of computing isochrones in road networks efficiently. Intuitively, an isochrone is the region that is reachable within a certain amount of time from a fixed source. Practical applications include reachability analysis in urban planning, geomarketing, and display a vehicle's remaining cruising range. We systematically collate different formal definitions used in the literature and present several practical algorithms for computing isochrones at continental scale.

Besides revisiting and extending known acceleration techniques, we propose a novel family of algorithms for computing isochrones, which builds upon graph separators and techniques related to contraction hierarchies. In contrast, existing approaches are based on multilevel Dijkstra searches.

Due to the increasing popularity of electric vehicles, we explicitly consider the problem of computing isochrones for electric vehicles, where the isochrone represents the remaining cruising range with the vehicle's current battery charge level. In order to obtain realistic results, we assume that drivers take quickest paths (as opposed to energy-optimal paths). We adapt some of our algorithms to this extended scenario.

We conclude with an experimental evaluation of our algorithms. Whereas the known acceleration techniques offer faster preprocessing times, our novel family of algorithms dominates in terms of query performance. However, all algorithms (except a variant of Dijkstra's algorithm) are fast enough for practical applications. Finally, we show that isochrones for electric vehicles are not "harder" than standard isochrones.

Zusammenfassung

Wir untersuchen das Problem, in einem Straßennetz Isochronen effizient zu berechnen. Intuitiv ist eine Isochrone die Region, die innerhalb einer gewissen Zeit von einem fixen Startpunkt erreicht werden kann. Praktische Anwendungen sind Erreichbarkeitsanalysen in der Städteplanung, Geomarketing und die Anzeige der verbleibenden Reichweite eines Fahrzeugs. Wir stellen verschiedene formale Definitionen aus der Literatur zusammen und präsentieren mehrere Algorithmen zur Berechnung von Isochronen.

Neben der Erweiterung von bekannten Beschleunigungstechniken schlagen wir eine neue Familie von Algorithmen zur Berechnung von Isochronen vor, die auf Graphenseparatoren und Contraction Hierarchies aufbaut. Dagegen basieren bestehende Ansätze auf Multilevel-Dijkstra-Suchen.

Aufgrund der wachsenden Beliebtheit von Elektroautos betrachten wir explizit das Problem, Isochronen für Elektroautos zu berechnen. Hierbei repräsentiert eine Isochrone die verbleibende Reichweite der aktuellen Batterieladung des Fahrzeugs. Um realistische Ergebnisse zu erhalten, nehmen wir an, dass Fahrer schnellste (statt energieeffiziente) Wege nehmen. Wir passen manche der Algorithmen an dieses erweiterte Szenario an.

Wir schließen mit einer experimentellen Evaluation unserer Algorithmen. Während die bekannten Beschleunigungstechniken schnellere Vorberechnungszeiten bieten, dominiert unsere neue Familie von Algorithmen bezüglich der Anfragegeschwindigkeit. Alle Algorithmen (außer einer Variante des Algorithmus von Dijkstra) sind allerdings schnell genug für praktische Anwendungen.

Contents

List of Figures ix

List of Tables xi

List of Algorithms xiii

1 Introduction 1

1.1 Related Work . . . 2
1.2 Contribution . . . 5
1.3 Overview . . . 7

2 Preliminaries 9

2.1 Basic Notation and Terminology . . . 9
2.2 Graph Separators and Partitions . . . 10
2.3 Point-to-Point Shortest Paths . . . 10
2.3.1 Dijkstra's Algorithm . . . 10
2.3.2 Customizable Route Planning . . . 11
2.3.3 Contraction Hierarchies . . . 12
2.4 Batched Shortest Paths . . . 13
2.4.1 The GRASP Algorithm . . . 13
2.4.2 The PHAST Algorithm . . . 14
2.4.3 Restricted PHAST . . . 14

3 Problem Statement 17

3.1 Formal Definition of Isochrones . . . 17
3.2 Isochrones for Electric Vehicles . . . 18
3.3 Dijkstra's Algorithm for Isochrones . . . 19

4 Multilevel Dijkstra Techniques 21

4.1 Basic Multilevel Dijkstra Algorithm for Isochrones . . . 21
4.1.1 General Idea . . . 21
4.1.2 Eliminating False Negatives . . . 22
4.1.3 Eliminating False Positives . . . 23
4.1.4 Further Optimization . . . 24

4.1.5 Determining Isochrone Pairs 25
4.2 Improved Multilevel Dijkstra Algorithm for Isochrones 25
4.2.1 Determining the Output 26
4.2.2 Parallelization 27
4.3 GRASP for Isochrones 28
4.3.1 Determining the Output 30
4.3.2 Parallelization 30

5 Combining Graph Separators and Contraction Hierarchies 33
5.1 Basic Algorithm 33
5.2 Edge Separators 35
5.2.1 Core-Dijkstra 35
5.2.2 Core-PHAST 40
5.2.3 Distance Oracle 43
5.2.4 Drawbacks of the ES+PHAST Algorithms 47
5.3 Vertex Separators 48
5.3.1 Computing Vertex Separators 48
5.3.2 From Vertex Separators to Topologies 49
5.3.3 The VS+PHAST Algorithm 49

6 Isochrones for Electric Vehicles 51
6.1 Modeling Energy Consumption 51
6.2 Basic Operations on Edge Cost Functions 53
6.2.1 Evaluation 53
6.2.2 Linking 54
6.2.3 Dominance Check 55
6.3 Extending Algorithms 57
6.3.1 RangeDijkstra 57
6.3.2 isoCRP 59
6.3.3 isoGRASP 62

7 Experimental Results 63
7.1 Inputs and Experimental Setup 63
7.2 Basic Building Blocks 64
7.3 Parameter Tuning 65
7.3.1 Multilevel Dijkstra Techniques 65
7.3.2 GS+PHAST Techniques 68
7.4 Main Results 74
7.5 Computing Isochrone Pairs 78
7.6 Electric Vehicle Scenario 80

8 Conclusion **83**
8.1 Summary 83
8.2 Future Work 84

Bibliography **87**

A Appendix **93**

List of Figures

3.1 Different types of edges in the context of isochrones 18

4.1 Examples where our approach produces wrong outputs 22
4.2 Example where an isochrone pair is missed 25

5.1 Illustration of the GS+PHAST algorithms 34
5.2 Illustration of the different distances that are involved in the computation of the distance oracle 44
5.3 An example of a topology obtained from a vertex separator 50

6.1 Cost functions on edges with negative and positive consumption costs 52
6.2 Cost function of a path consisting of edges with negative and positive consumption costs 53
6.3 Linking two cost functions 54
6.4 Dominance check seen as a geometric problem 56
6.5 Example where an isochrone edge is missed 61

7.1 Query times for varying number of cores using different scheduling strategies 68
7.2 Sequential query times for varying time limits when computing isochrone edges 76
7.3 Parallel query times using 16 cores for varying time limits when computing isochrone edges 76
7.4 Query times for varying number of cores when computing isochrone edges . 78
7.5 Sequential query times for varying time limits when computing isochrone pairs 79
7.6 Number of isochrone edges and isochrone pairs for varying time limits 80
7.7 Sequential query times for varying initial battery charge levels when computing isochrone edges 82

A.1 Parallel query times using 16 cores for varying time limits on DIMACS Europe when computing isochrone pairs 93
A.2 Parallel query times using 16 cores for varying initial charge levels on PTV Europe when computing isochrone edges 96

List of Tables

7.1 Performance of the basic one-to-all and one-to-many building blocks 65
7.2 Impact of the multilevel partition on the performance of isoCRP for varying number of levels and different cell sizes 66
7.3 Impact of the multilevel partition on the performance of isoGRASP for varying number of levels and different cell sizes 67
7.4 Impact of the number of cells on the (parallel) time and space for the preprocessing of the GS+PHAST techniques 69
7.5 Space required to represent the downward subgraphs in ES+PHAST (do) for different partitions and varying choices of k 70
7.6 Space required to represent the downward subgraphs in VS+PHAST for different partitions and varying choices of k 70
7.7 Performance of ES+PHAST (cd) and ES+PHAST (cp) for varying number of cells 71
7.8 Parallel query times of ES+PHAST (do) for different partitions and varying choices of the parameter k 73
7.9 Parallel query times of VS+PHAST for different partitions and varying choices of the parameter k 74
7.10 Performance of the different algorithms on DIMACS Europe when computing isochrone edges 75
7.11 Performance of the different algorithms on DIMACS Europe when computing isochrone pairs 79
7.12 Performance of the different algorithms in the electric vehicle (EV) scenario . 81

A.1 Sequential query times of ES+PHAST (do) for different partitions and varying choices of the parameter k 94
A.2 Sequential query times of VS+PHAST for different partitions and varying choices of the parameter k 95

List of Algorithms

4.1 ComputeQueryLevel(u) . . . 24
4.2 isoCRP(s, x, G, H) . . . 27
4.3 isoGRASP(s, x, G, H, $G^{\downarrow}_{GS}$) . . . 29

6.1 EvalEdgeCostFunction(f, b) . . . 54
6.2 LinkEdgeCostFunctions(f_1, f_2) . . . 55
6.3 Dominates(f_1, f_2) . . . 56

1 Introduction

Web-based map services, autonomous navigation systems and other location-based applications have gained wide currency in the last two decades. This motivated a great deal of research on practical algorithms for routing in road networks [3, 59]. Most work focused on computing the *shortest-path distances* and, if required, the complete *path descriptions* from single sources to single targets or between sets of vertices. However, several applications only need to know the vertices that are within reach of a certain location, but not the actual distances or path descriptions to them. Such a set of vertices that are reachable within a certain amount of time from a specific source is called an *isochrone*. This thesis focuses on computing isochrones in road networks.

A practical application that relies on isochrone computations is *reachability analysis in urban planning* [37, 38]. Here, one assesses the coverage of the city by various kinds of public services, such as hospitals, schools or tram stops. Such analyses are an important instrument to place strategic public objects in optimal position. Recently, Bauer et al. [8] deployed such a system for reachability analysis at the Municipality of Bolzano-Bozen.

A related application is *geomarketing* [28], which integrates geographical information into business intelligence. For example, isochrones are used as a basis for decisions such as where to build a new franchise store in order to maximize the pool of customers reached.

Isochrones are also useful for online applications like *job markets* or *real estate portals* [49]. When somebody searches for vacancies close to the home town or apartments near to the workplace, the portal should only display the offers that are reachable within the user's maximal acceptable traveling duration.

The above applications use isochrones as a primitive operation to implement sophisticated location services. Isochrones are, however, also a reasonable end user feature of web-based map services and car navigation systems, especially for electric vehicle (EV) drivers: One of the main concerns about electric vehicles is range anxiety [46], which describes the fear of getting stranded. Displaying the region that is reachable with the current battery charge

level may diminish range anxiety among EV drivers. An effective way to obtain this region, which is sometimes called *cruising range* or *isochrone area*, is to compute a special variant of isochrones, which is introduced in this thesis.

1.1 Related Work

Algorithms for route planning in road networks have received so much attention in the last decade that even fairly recent overviews [20, 60] have already become outdated and thus were updated in later publications [3, 59]. Dijkstra's classic algorithm [26, 12] solves several shortest-path problem variants in almost linear time [41], however, it is still too slow for many practical applications. Hence, production systems use various speedup techniques that consist of an offline *preprocessing stage* and an online *query stage*. We now overview several speedup techniques on which this thesis builds on, organized into algorithms for point-to-point shortest paths, batched shortest paths, EV routing and isochrones.

Point-to-Point Shortest Paths. Given a directed graph $G = (V, E)$, a non-negative edge cost function $\ell : E \to \mathbb{R}_{\geq 0}$, a source $s \in V$ and a target $t \in V$, the point-to-point shortest-path problem is to compute the distance, i.e., the length of a shortest path, from s to t in G. Three important families of point-to-point algorithms are goal-directed techniques, separator-based techniques and hierarchical techniques. In the following, we outline from each family the algorithm which is the most relevant in our context.

Goal-Directed Techniques. Dijkstra's algorithm grows a circular search space around the source [53]. Goal-directed techniques, in contrast, grow search spaces that are oriented towards the target. A well-known example of such speedup techniques is the *ALT (A* search, Landmarks, Triangle inequality) algorithm* [42]. Queries build upon the classic *A* search* [44], which works similar to Dijkstra's algorithm. However, whereas Dijkstra's algorithm scans at each step a vertex v with minimum tentative distance $d_s(v)$, the A* search scans a vertex v that minimizes $d_s(v) + f_t(v)$. Here, $f_t(v)$ is a lower bound on the distance $\mathrm{dist}(v, t)$ from v to t. If we used exact lower bounds, i.e., $f_t(v) = \mathrm{dist}(v, t)$ for all $v \in V$, only vertices on shortest $s-t$ paths would be scanned. If $f_t(v) = 0$ for all $v \in V$, the A* search is equivalent to Dijkstra's algorithm. Generally, better lower bounds lead to smaller search spaces and thus faster queries.

To obtain lower bounds, the ALT algorithm selects a small set $L \subseteq V$ of *landmarks*. For each landmark, it precomputes shortest-path distances to and from each vertex. Plugging these distances in the *triangle inequality* we get for each landmark l two lower bounds on the distance from a vertex v to t: $\mathrm{dist}(v, t) \geq \mathrm{dist}(l, t) - \mathrm{dist}(l, v)$ and $\mathrm{dist}(v, t) \geq \mathrm{dist}(v, l) - \mathrm{dist}(t, l)$. Queries take the maximum, over all landmarks, of these lower bounds to compute the tightest lower bound. Since the first publication, there have been significant improvements [43, 21, 29], which make ALT practical even for dynamic scenarios such as real-time traffic updates.

Separator-Based Techniques. Since road networks have small separators [34], recent graph partitioning algorithms such as PUNCH [16] and Buffoon [56] are able to obtain balanced partitions of them with small sets of cut edges. Separator-based techniques build on such partitions. A practical example is the *Customizable Route Planning (CRP) algorithm* [14, 15]. The CRP preprocessing builds an overlay graph that contains the cut edges of the partition and their endpoints, i.e., the boundary vertices of the partition. Furthermore, it creates a shortcut edge between each pair u, v of boundary vertices within a cell, that represents the shortest $u - v$ path within the cell. Hence, the distance between any two vertices in the overlay graph is the same as in the original graph. Queries run a bidirectional version of Dijkstra's algorithm on the union of the overlay graph and the subgraphs that are induced by the source and target cell, respectively. Since the overlay graph is much smaller than the original graph, queries are orders of magnitude faster than Dijkstra's algorithm. Another advantage is the distinction between a *metric-independent preprocessing stage* and a *metric customization stage*, which allows to incorporate new cost functions (e.g. due to traffic updates) quickly. In practice, one uses multiple levels of overlay graphs in order to improve query performance. There is also work on accelerating the metric customization stage, both on the CPU [22] and the GPU [19]. Most recently, Delling et al. [23] extended CRP to support point-of-interest queries.

Hierarchical Techniques. Road categories such as rural roads, minor urban streets or freeways define an explicit hierarchy on the road network. However, many metrics such as travel times or travel distances also impose an implicit hierarchy on the road network. For example, since we can drive faster on freeways, a large faction of sufficiently long quickest paths only uses freeways. The same is true for sufficiently long shortest paths, since freeways tends to have less bends. Hierarchical techniques exploit the implicit hierarchy of the road network. A well-known example of such speedup techniques are *Contraction Hierarchies (CH)* [40]. The CH preprocessing contracts all vertices in increasing order of importance. To contract a vertex v, it removes v temporarily from the graph and creates a shortcut between each pair u, w of neighbors if the shortest $u - w$ path is unique and contains v. The query is a variant of the bidirectional Dijkstra search, which only looks at edges leading to more important vertices. This modification leads to small search spaces and thus fast query times.

In contrast to CRP, Contraction Hierarchies as in the original publication are not customizable. Recently, however, Dibbelt et al. [25] introduced *Customizable Contraction Hierarchies*, which extend Contraction Hierarchies to support customization. To do so, they do not determine the contraction order online and bottom-up, but use nested dissection orders. Such orders were proposed in [6], after showing a theoretical result that suggested to do so.

Batched Shortest Paths. In contrast to point-to-point shortest paths, batched shortest paths involve more than two vertices. The most common variants are *one-to-all* shortest paths, *one-to-many* shortest paths and *many-to-many* shortest paths. Dijkstra's algorithm solves all these problems, however, as mentioned before, it is too slow for practical applications on large road networks.

One-to-All Shortest Paths. The one-to-all shortest path problem asks to compute shortest-path distances from a source vertex s to all other vertices in the graph. Although the fast computation of shortest path trees is a relevant problem and required for the preprocessing of several point-to-point algorithms (such as Arc Flags [47, 51] or CHASE [7]), there has been little work on one-to-all algorithms. Besides Dijkstra's algorithm, we are aware of two other one-to-all techniques. One of them is the *PHAST algorithm* [13], which builds upon Contraction Hierarchies. Queries consist of two phases. In the first phase, PHAST performs a forward upward search from s. In the second, it propagates distance values from more to less important vertices. Since the importance of vertices depends solely on the contraction hierarchy, but not on the source, the order in which the second phase processes the vertices is the same for all sources. Hence, it is possible to reorder the vertices such that the second phase is essentially a linear sweep over the vertices. Note that there are PHAST implementations tailored to multi-core workstations as well as to graphics cards. More recently, Efentakis et al. [30] proposed the *GRASP algorithm*, which is an one-to-all technique that builds upon CRP. Whereas it has slightly slower query times than the PHAST algorithm, it allows for customization in a few seconds. Most recently, Efentakis et al. [31] combined CRP, GRASP and ALT into a unified framework that efficiently solves multiple shortest-path problems.

One-to-Many Shortest Paths. The one-to-many shortest path problem is to compute shortest-path distances from a single source s to all vertices in a target set T. Obviously, we can solve the problem by computing $|T|$ point-to-point shortest paths or, alternatively, by considering one-to-many a special case of one-to-all. We can do better by using dedicated one-to-many algorithms. Delling et al. [17] introduced *RPHAST*, which is a variant of PHAST tailored to one-to-many shortest paths. It uses a three-phase workflow. The first phase is a standard PHAST preprocessing and thus is independent of the target set. The second phase extracts from the contraction hierarchy only the information necessary to compute shortest-path distances from s to each $t \in T$. During queries, RPHAST performs a standard PHAST query, but only on the relevant part of the contraction hierarchy. There is also a variant of the GRASP algorithm that is tailored to one-to-many shortest paths [30].

Many-to-Many Shortest Paths. The many-to-many shortest path problem asks to compute a $|S| \times |T|$ distance table D, where $D(s,t)$ denotes the shortest-path distance between $s \in S$ and $t \in T$. Of course, the problem can be solved by computing $|S| \cdot |T|$ point-to-point shortest paths or, alternatively, by computing $\min\{|S|, |T|\}$ one-to-all shortest paths. However, there are also algorithms tailored to many-to-many shortest paths. The technique proposed by Knopp et al. [50] builds upon a hierarchical technique such as contraction hierarchies and maintains a bucket $B(v)$ for each vertex v. In a first step, it performs a backward upward search from each vertex $t \in T$. When scanning a vertex v, it adds a pair $(t, d_t[v])$ to v's bucket $B(v)$. Here, $d_t[v]$ denotes v's distance label during the search from t. In a second step, the algorithm performs a forward upward search from each $s \in S$. When it scans a vertex v, it checks for each pair $(t, d_t[v]) \in B(v)$ whether $d_s[v] + d_t[v] < D(s,t)$. If so, it updates the distance label accordingly. This many-to-many algorithm was introduced for

Highway Hierarchies [55], but it can also be used with Contraction Hierarchies [17] or even with Hub Labels [1, 17].

EV routing. As observed by Artmeier et al. [2], computing energy-optimal paths differs from obtaining shortest, quickest or cheapest paths, due to the edge costs possibly being negative and battery constraints that need to be obeyed. Negative edge costs occur since electric vehicles are able to recuperate energy when going downhill. Hence, Artmeier et al. apply variants of label-correcting Dijkstra and Bellman-Ford [10, 35] algorithms. Eisner et al. [33] observe that the energy required to get across a path can be modeled as a cost function of bounded descriptive complexity. This observation allows them to adapt contraction hierarchies to compute energy-optimal paths. Note that their approach is somewhat similar to time-dependent contraction hierarchies [4]. Using the same cost functions as in [33], Baum et al. [9] adapted the CRP framework to compute energy-optimal paths.

Isochrones. Most work on the computation of isochrones has been done in the database community. Isochrones were introduced in [8]. In this work, isochrones are computed using an algorithm that resembles a label-correcting breadth-first search. The *MIME algorithm* proposed by Gamper et al. [37] is basically a plain Dijkstra search on top of a spatial network database. The improved variant MIMEX [38] has reduced space requirements. In order to transform an isochrone in the form of a vertex set into an isochrone area, one may use concave hulls or alpha shapes [27]. Both methods compute areas from arbitrary sets of two-dimensional points. To obtain better results, Marciuska et al. [52] propose two algorithms that are tailored to compute areas from isochrones. More recently, Innerebner et al. [49] combined the MIMEX algorithm with methods to compute isochrone areas into a unified web-based application. To summarize, to the best of our knowledge, the database community uses no speedup techniques to compute isochrones on top of spatial network databases.

There has been very little work on computing isochrones outside the database community. Besides one-to-all and one-to-many variants, Efentakis et al. [30] also propose a variant of the GRASP algorithm for computing isochrones. However, their algorithm computes correct distances to all vertices that are reachable within the time limit, which is not required by several applications, for example when computing isochrone areas. There has also been a patent specification [18] on different query scenarios for the CRP algorithm, which outlines a method for computing isochrones using CRP. However, we are not aware of a scientific publication that actually evaluated the method on large road networks.

1.2 Contribution

In this thesis, we study the problem of computing isochrones. Intuitively, an isochrone is the region that is reachable from a certain source within a certain amount of time. The formal definition varies from publication to publication. We formulate the different possibilities to define isochrones and use two of them throughout this thesis. Besides describing a variant

of Dijkstra's algorithm called RangeDijkstra, that is capable of computing isochrones, our four main contributions are as follows.

1. We explore how multilevel overlay graphs can be used to accelerate RangeDijkstra and propose a basic multilevel algorithm for computing isochrones that is one order of magnitude faster than RangeDijkstra. We also revisit two known algorithms for computing isochrones. The patent specification [18] outlines a method within the CRP framework, however, it discusses no parallelization approaches and does not evaluate the method. We provide an efficient implementation and propose different parallelization strategies. The other technique we revisit is the isoGRASP algorithm. In the original publication [30], it computes correct distances to all vertices that are reachable within the specified amount of time. Since actual distances are generally not needed when computing isochrones, we extend isoGRASP to jump over cells that contain only vertices that are reachable within the time limit.

2. Motivated by the promising query times of RPHAST [17], we propose a novel family of so-called GS+PHAST algorithms. Their key ingredients are graph separators and the (R)PHAST algorithm. We present variants that use k-way edge separators (similar to CRP) as well as a variant that uses (k-way) vertex separators. For many time limits, the fastest GS+PHAST variants outperform the multilevel Dijkstra algorithms.

3. We study the problem of computing isochrones for electric vehicles, that is, computing the region that is reachable with the vehicle's current battery charge level. Simply using electric energy consumption instead of travel times as cost function for computing isochrones would lead to unrealistic estimations of the cruising range, since energy-optimal paths differ considerably from quickest paths [9] and even an eco-friendly driver might not take them[1]. Hence, we use two cost functions in the EV scenario: a routing cost function, namely travel times, and a separate consumption cost function, namely the energy consumption of the electric vehicle. We present variants of RangeDijkstra and the multilevel Dijkstra techniques that are capable of computing isochrones in the EV scenario.

4. We provide an experimental evaluation of the different algorithms for computing isochrones. We are the first that evaluate isoCRP on large road networks and compare it with our isoGRASP variant and the different GS+PHAST algorithms. We also show that using electric energy consumption as a separate cost function causes almost no overhead compared to the standard scenario.

[1] Apart from unrealistic estimations, the ability of electric vehicles to recuperate energy when going downhill prevents us from "simply" using electric energy consumption instead of travel times as cost function, since most speedup techniques support only non-negative cost functions without further modifications.

1.3 Overview

The remainder of this thesis is organized into seven chapters. Chapter 2 introduces some basic notation and terminology. It also reviews the basic building blocks this thesis builds upon. The point-to-point techniques relevant to this thesis are Dijkstra's algorithm, the CRP framework and Contraction Hierarchies. Closely connected batched shortest paths techniques are GRASP and the (restricted) PHAST algorithm.

Chapter 3 formalizes the notion of isochrones. It discusses the different possibilities to define isochrones and chooses two of them. We finish the chapter with a description of RangeDijkstra, a variant of Dijkstra's algorithm that is capable of computing isochrones.

Chapter 4 deals with multilevel Dijkstra techniques for computing isochrones. It explores how multilevel overlay graphs can be used to accelerate RangeDijkstra and also revisits and extends two known algorithms, isoCRP and isoGRASP. We detail for each algorithm how it determines the output and how it is parallelized.

Chapter 5 proposes the novel family of GS+PHAST algorithms. With CRP in mind, we start with the variants that leverage k-way edge separators. Afterwards, we move on to (k-way) vertex separators. Again, we detail for each variant how the output is determined and how we parallelize it.

Chapter 6 considers isochrones for electric vehicles. It first reviews how the energy required to get across a path can be modeled as a cost function of bounded descriptive complexity. Afterwards, we introduce some basic operations on such cost functions that we will use in our algorithms. We also elaborate on how to implement each operation efficiently. Having laid the foundations, we adapt RangeDijkstra and the multilevel Dijkstra algorithms to the EV scenario.

Chapter 7 provides an experimental evaluation of the different algorithms for computing isochrones. We focus on the standard scenario, but also include some figures for the EV scenario. Chapter 8 summarizes the results and outlines further routes of study.

2 Preliminaries

This chapter introduces some basic notation and terminology. Afterwards, we review the basic point-to-point building blocks this thesis builds upon. The chapter finishes with some background information on existing algorithms for batched shortest-path computations.

2.1 Basic Notation and Terminology

We represent a road network as a *directed graph* $G = (V, E)$. A *vertex* $v \in V$ represents an intersection or junction, and an *edge* $e \in E$ represents a road segment. Let $|V| = n$ be the number of vertices, and $|E| = m$ the number of edges in G. Consider an edge $e = (u, v)$ going from u to v. We refer to u and v as the *tail* and *head*, respectively. We say that u and v are *adjacent* or *neighbors* and that e is *incident* on u and v. Moreover, e is said to be an *outgoing edge* of u and an *incoming edge* of v. The *degree* of v is the number of edges incident on v.

A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. The subgraph *induced* by a subset $V' \subseteq V$ is defined as $G' = (V', E')$, where $E' = \{(u, v) \in E : u, v \in V'\}$. A *metric* or *cost function* $\ell : E \to \mathbb{R}$ assigns each edge $e = (u, v)$ a *constant edge cost* $\ell(u, v)$. Natural metrics include travel times, travel distances, walking, biking, and many other criteria. We often use the general term *length* to encompass different metrics.

A *path* $P = (v_0, \ldots, v_k)$ is a sequence of vertices in which consecutive vertices are neighbors, that is, $(v_0, v_1) \in E, (v_1, v_2) \in E, \ldots, (v_{k-1}, v_k) \in E$. Metrics carry over to paths straightforwardly, that is, the length of a path P is defined as $\ell(P) = \sum_{i=0}^{k-1} \ell(v_i, v_{i+1})$. A vertex v is *reachable* from a vertex u if there is a path from u to v in G. Note that the relation “being reachable” is not necessarily symmetric in directed graphs. We say that a graph $G = (V, E)$ is *strongly connected* if any two vertices $u, v \in V$ are reachable from each other. Otherwise, the graph is *weakly connected*.

The *distance* $\text{dist}(u, v)$ from a vertex u to a vertex v is the length of the *shortest path* from u to v in G. If v is not reachable from u, we set $\text{dist}(u, v) = \infty$. A *shortest path tree* T_s rooted

at s is a tree that contains all vertices $v \in V$ reachable from s. The path from s to each vertex $v \neq s$ is a shortest $s-v$ path in G. Hence, T_s is a subgraph of G.

2.2 Graph Separators and Partitions

We adopt the notation and terminology from [15]. Given a graph $G=(V,E)$, a k*-way partition* of V is a family $\mathscr{C}=\{C_0,\ldots,C_k\}$ of k *cells* $C_i \subseteq V$ such that $C_0 \cup \cdots \cup C_k = V$ and $C_i \cap C_j = \emptyset$ for all $i \neq j$. A *multilevel partition* of V is a family of partitions $\{\mathscr{C}^0,\ldots,\mathscr{C}^L\}$. Let ℓ be the *level* of a partition $\mathscr{C}^l$. We define $\mathscr{C}^0$ as the "artificial" $|V|$-way partition where each vertex is contained in its own cell. Hence, L denotes the number of levels. To simplify notation, we also define $\mathscr{C}^{L+1}$ as the 1-way partition where all vertices are contained in the same cell.

Throughout this thesis, we use *nested* multilevel partitions. The *supercell* of a cell $C_i^\ell \in \mathscr{C}^\ell$ is the cell $C_j^{\ell+1} \in \mathscr{C}^{\ell+1}$ with $C_i^\ell \subseteq C_j^{\ell+1}$. Conversely, C_i^ℓ is the *subcell* of $C_j^{\ell+1}$ if $C_j^{\ell+1}$ is the supercell of C_i^ℓ. We denote by $c_\ell(v)$ the cell that contains v on level ℓ. When $L=1$ we may use $c(v)$ instead of $c_1(v)$. We refer to an edge $e=(u,v)$ with $c_\ell(u) \neq c_\ell(v)$ as a *boundary edge* on level ℓ, and say that u and v are *boundary vertices* on level ℓ.

We define the *eccentricity* of a boundary vertex u on level ℓ. Let H be the subgraph induced by $c_\ell(u)$. We denote by $f_\ell(u)$ the vertex v farthest from u in H with $\mathrm{dist}(u,v) < \infty$. The eccentricity of u on level ℓ is then defined as $\mathrm{ecc}_\ell(u) = \mathrm{dist}(u, f_\ell(u))$. Note that, throughout this thesis, we often do not use exact eccentricities, but upper bounds on them.

A k*-way edge separator* is a subset $S \subset E$ of the edges such that the removal of S decomposes the graph $G=(V,E)$ into k cells. Note that edge separators are closely related to partitions, since the set of boundary edges of a k-way partition forms a k-way edge separator. A k*-way vertex separator* is a subset $S \subset V$ of the vertices such that the removal of S decomposes G into k cells. Whereas edge separators lead to cell boundaries crossing the separator edges, vertex separators yield cell boundaries passing through the separator vertices.

2.3 Point-to-Point Shortest Paths

Given a directed graph $G=(V,E)$, a non-negative metric $\ell : E \to \mathbb{R}_{\geq 0}$, a source $s \in V$ and a target $t \in V$, the *point-to-point shortest-path problem* is to compute the shortest-path distance from s to t in G. This section briefly reviews three point-to-point algorithms that are important throughout this thesis.

2.3.1 Dijkstra's Algorithm

The standard solution in a textbook on algorithms to the point-to-point shortest path problem is *Dijkstra's algorithm* [26, 12]. It maintains for each vertex v a distance label $d[v]$, which

stores the length of the shortest path from s to v found so far. The algorithm initializes the source's distance label to zero, and all other labels to infinity. Moreover, it maintains a priority queue of *unscanned* vertices, using their distance labels as keys. Vertices are *scanned* in increasing order of distance from s. To scan a vertex u, the algorithm *relaxes* all outgoing edges (u,v). For each such edge, if $d[u]+\ell(u,v) < d[v]$, it sets $d[v] = d[u]+\ell(u,v)$ and adds vertex v with key $d[v]$ to the priority queue. Dijkstra's algorithm has the *label-setting* property, that is, $d[u] = \mathrm{dist}(s,u)$ when vertex u is scanned (see [53] for a proof). Hence, the search may stop after removing the target from the queue.

The running time depends on the priority queue used. Due to the label-setting property, each vertex is scanned at most once, resulting in at most n delete-min operations on the priority queue. Since a scanned vertex is never reinserted, there are also at most n insert operations. Moreover, the algorithm relaxes at most m edges, and each relaxation yields at most one decrease-key operation, there are at most m decrease-key operations in total. Hence, the execution time of Dijkstra's algorithm is $T_{\text{Dijkstra}} = O(m \cdot T_{decreaseKey} + n \cdot (T_{deleteMin} + T_{insert}))$. This solves to $O((m+n)\log n)$ using binary heaps [61], and to $O(m+n\log n)$ using Fibonacci heaps [36]. Note, however, that $m \in O(n)$ for road networks, resulting in a running time of $O(n\log n)$ for both binary heaps and Fibonacci heaps. Since the latter involves larger constant factors, binary heaps are a good choice in practice.

To accelerate queries, a *bidirectional version* of Dijkstra's algorithm may be used, which simultaneously runs a forward search from s in the original graph and a backward search from t in the reversed graph until the search frontiers meet. More precisely, the algorithm stops once the first vertex has been scanned by both searches (see [53] for a proof).

2.3.2 Customizable Route Planning

Customizable Route Planning (CRP) [14, 22, 19, 15] follows a three-phase workflow, subdividing preprocessing into *metric-independent preprocessing* and *metric customization*. The (metric-independent) preprocessing creates a (multilevel) partition of the road network. This partition induces a *partition-based overlay graph* H. In general, a graph $G' = (V', E')$ is an overlay graph of $G = (V,E)$ if $V' \subseteq V$ and G' preserves the shortest-path distances between all vertices $u, v \in V'$. The overlay H contains all boundary vertices and boundary edges of the road network. It also contains a *clique* for each cell C. That is, for each pair (u,v) of boundary vertices in C, there is a *shortcut edge* going from u to v.

After partitioning the road network, the preprocessing builds the topology of the overlay and sets up appropriate data structures. Since the overlay is a collection of cliques, the topology is represented as square matrices in contiguous memory for efficiency. A cell with n boundary vertices corresponds to a $n \times n$ matrix in which position (i, j) will store the length of the shortest path (within the cell) from the cell's i-th boundary vertex to the its j-th boundary vertex. However, preprocessing only sets up the matrices, but does not fill them with valid shortest-path distances. To accelerate queries, multiple levels of overlay graphs may be

used. In order to improve locality and cache efficiency in this setup, boundary vertices of the highest level are assigned the lowest IDs, followed by boundary vertices of the second highest level, and so on.

Customization has access to the actual metric and computes the entries of the matrices. It processes one cell C at a time. Let v be the i-th boundary vertex in C. Customization runs Dijkstra's algorithm from v in the original graph (restricted to C) until the priority queue is empty. This fills the i-th row of the matrix representing C. For a cell C at a higher level H_i, customization runs Dijkstra's algorithm on the subgraph of H_{i-1} that is induced by the subcells of C.

For a query between s and t, a bidirectional version of Dijkstra's algorithm must be run on the graph consisting of the union $c_1(s) \cup \cdots \cup c_{L-1}(s) \cup H_L \cup c_{L-1}(t) \cup \cdots \cup c_1(t)$, that is, the union of the top-level overlay H_L and for each level ℓ the cells $c_\ell(s)$ and $c_\ell(t)$ that contain s and t, respectively (see [48] for a proof). The level at which a vertex v must be scanned during the search is referred to as its *query level* ℓ_q. It is defined as the maximum i such that $c_i(v) \cap \{s, t\} = \emptyset$.

2.3.3 Contraction Hierarchies

Contraction Hierarchies (CH) [40] work in two phases. The preprocessing heuristically orders the vertices by importance and *contracts* them from least to most important. To contract a vertex v, it deletes v from the graph (temporarily) and adds *shortcuts* between v's as-yet-uncontracted neighbors to preserve shortest-path distances. Let S be the set of tails u of the edges (u, v) coming into v, and let T be the set of heads w of the edges (v, w) going out of v. For each vertex $v \in S$, the preprocessing runs Dijkstra's algorithm from v in the as-yet-uncontracted graph until all vertices in $T \setminus \{v\}$ have been scanned. Let $\delta_v(w)$ be the length of the shortest v−w path found by this *witness search* or *local search*. If $\delta_u(w) > \ell(u, v) + \ell(v, w)$, a shortcut (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$ is added. In practice, one may limit the local searches, since otherwise long-distance edges like ferry connections slow down the contraction due to expensive local searches.

The output of the CH preprocessing routine is the set E^+ of shortcuts and the contraction order rank$(\cdot)$ itself. The vertex v with rank$(v) = 1$ is least important, and the vertex v' with rank$(v') = n$ is most important. In addition, *levels* $L(v)$ can be assigned to vertices v during preprocessing. All levels are initialized to zero. Whenever the preprocessing contracts a vertex v, it sets $L(w) = \max\{L(w), L(v) + 1\}$ for each as-yet-uncontracted neighbor w of v. Conceptually, the preprocessing yields two graphs. The edge set $E^\uparrow$ of the *upward graph* $G^\uparrow = (V, E^\uparrow)$ is defined as $E^\uparrow = \{(u, v) \in E \cup E^+ : \text{rank}(u) < \text{rank}(v)\}$. Conversely, the edge set $E^\downarrow$ of the *downward graph* $G^\downarrow = (V, E^\downarrow)$ is defined as $E^\downarrow = \{(u, v) \in E \cup E^+ : \text{rank}(u) > \text{rank}(v)\}$.

The query stage runs a bidirectional version of Dijkstra's algorithm, where the forward search from s is run in $G^\uparrow$ and the backward search from t is run in $G^\downarrow$. Let $d_s(\cdot)$ and $d_t(\cdot)$ be

the distance labels maintained by the upward and downward search, respectively. It is guaranteed that the maximum-rank vertex u on the shortest $s-t$ path is scanned by both searches, and that $d_s(u)+d_t(u)=\text{dist}(s,t)$ (see [40] for a proof). Since both searches only look at upward edges, random $s-t$ queries visit fewer than 400 vertices (out of 18 million [13]), making Contraction Hierarchies four orders of magnitude faster than Dijkstra's algorithm, and up to an order of magnitude faster than CRP queries.

Selecting an optimal contraction order is **NP**-hard [5]. Usually, the order is heuristically determined online and bottom-up. The selection is done using a priority queue, with a linear combination of different *priority terms* as key. Geisberger [39] provides an exhaustive description of different priority terms.

2.4 Batched Shortest Paths

In contrast to point-to-point shortest paths, batched shortest paths involve more than two vertices. Given a directed graph $G=(V,E)$, a non-negative metric $\ell : E \to \mathbb{R}_{\geq 0}$, and a source vertex s, the *one-to-all shortest-path problem* is to compute the shortest-path distances from s to all other vertices in the graph. Moreover, given a nonempty set of targets $T \subseteq V$, the *one-to-many shortest-path problem* requires computing the shortest-path distances between s and all vertices in T. Dijkstra's algorithm may be used to compute batched shortest paths. This section briefly reviews first two existing one-to-all algorithms and then a known one-to-many technique.

2.4.1 The GRASP Algorithm

GRASP (Graph separators, RAnge, Shortest Path) [30, 31] is an extension of the CRP routing engine to handle batched shortest paths. The metric-independent preprocessing runs the standard CRP preprocessing. However, customization does not only compute shortest-path distances from a boundary vertex u in $c_\ell(u)$ to the other boundary vertices in $c_\ell(u)$, but to all vertices $v \in c_\ell(u) \cap H_{\ell-1}$. The overhead is limited since the exact same Dijkstra search are used as before. Besides computing *shortcut edges*, the customization of GRASP also produces *downward edges* (u,v) connecting a boundary vertex u in $c_\ell(u)$ to each vertex $v \in H_{\ell-1}$ in the interior of $c_\ell(u)$. The length of a level-ℓ downward edge (u,v) is the same as the shortest path restricted to $c_\ell(u)$ between u and v. All downward edges are stored in a separate *downward graph* $G^{\downarrow}_{\text{GS}}$ for efficiency.

Queries execute the one-to-all search in two phases. First, they perform a simple forward CRP search. More precisely, they run Dijkstra's algorithm on the graph consisting of the union $c_1(s) \cup \cdots \cup c_{L-1}(s) \cup H_L$, that is, the union of the top-level overlay H_L and for each level ℓ the cell $c_\ell(s)$ that contains s. The search stops when the queue becomes empty. By definition of overlay graphs, all vertices that have been scanned during the *upward phase* have correct distance labels.

The *scanning phase* processes one cell at a time in descending level order. It propagates distance values from boundary vertices u of $C \in \mathscr{C}^{\ell}$ to all vertices $v \in C \cap H_{\ell-1}$ by examining the level-ℓ downward edges within C. To examine a downward edge (u, v), it checks whether $d[u] + \ell(u, v) < d[v]$ and updates v's distance label accordingly. In order to reduce the number of downward edges, the edge reduction optimization from [32] may be used. It stores a level-ℓ downward edge (u, v) in the downward graph only if the shortest $u - v$ path within $c_{\ell}(u)$ contains no boundary vertices of $c_{\ell}(u)$ (except u, of course).

2.4.2 The PHAST Algorithm

PHAST (for PHAST hardware-accelerated shortest path trees) [13] is an one-to-all algorithm building upon Contraction Hierarchies. Preprocessing is the same as in CH. That is, it defines a contraction order rank(·), assigns levels $L(v)$ to vertices v, and builds the upward graph $G^{\uparrow}$ and the downward graph $G^{\downarrow}$. Queries work in two phases. The *upward phase* performs a simple forward CH search. In other words, it runs Dijkstra's algorithm from s in $G^{\uparrow}$ until the priority queue becomes empty. The *scanning phase* propagates distance values from more to less important vertices, by processing all vertices in descending rank order. To process a vertex v, it checks for each incoming edge $(u, v) \in E^{\downarrow}$ whether $d[u] + \ell(u, v) < d[v]$ and updates v's distance label accordingly.

It is easy to see that queries are correct. Consider any vertex v and let u be the maximum-rank vertex on the shortest $s - v$ path. The shortest $s - u$ subpath is found during the upward phase, due to the correctness of CH. Clearly, the shortest $u - v$ subpath is found during the scanning phase. Hence, queries compute correct distance labels. Since the instruction flow of the scanning phase depends only on the contraction order, but not on the source, vertices are reordered during preprocessing. Vertices at higher levels are assigned lower IDs. Then, the scanning phase becomes a simple linear sweep through the edge array of $G^{\downarrow}$, making PHAST 15 times faster than Dijkstra's algorithm [13].

2.4.3 Restricted PHAST

Restricted PHAST (RPHAST) [17] is an extension of the PHAST algorithm to handle one-to-many shortest paths. It follows a three-phase workflow, introducing an additional *target selection phase* "between" preprocessing and queries. The preprocessing does not depend on the targets and is exactly the same as in PHAST. The target selection has access to the actual targets T and extracts a subgraph $G^{\downarrow}_{T}$ of the downward graph $G^{\downarrow}$ that contains only the information necessary to compute shortest-path distances to all targets T. Queries resemble PHAST queries, but use $G^{\downarrow}_{T}$ instead of $G^{\downarrow}$ during the scanning phase.

To ensure correctness, $G^{\downarrow}_{T}$ needs to contain the reverse CH search spaces of all targets. They are computed at once by performing a single search from all vertices in T. More precisely, the target selection phase maintains a set T' of relevant vertices and a FIFO queue Q of

"unchecked" vertices. Initially $T' = Q = T$. While Q is not empty, the target selection phase removes a vertex v from it and checks for each incoming edge (u, v) whether $u \in T'$. If not, it adds u to T' and Q. Finally, $G^{\downarrow}_{T}$ is the subgraph of $G^{\downarrow}$ induced by T'.

3 Problem Statement

Intuitively, an isochrone is the region that is reachable from a certain source within a specific amount of time. This chapter formalizes the notion of isochrones and specifies the output that our algorithms should produce. To do so, we first explore the different possibilities for defining isochrones and compare them with the definitions of known algorithms. Afterwards, we formalize isochrones in the context of electric vehicles. The chapter finishes with a description of a variant of Dijkstra's algorithm that is capable of computing isochrones.

3.1 Formal Definition of Isochrones

We consider a directed graph $G = (V, E)$ together with a non-negative cost function $\ell : E \to \mathbb{R}_{\geq 0}$ that assigns each edge the time it takes to travel along the edge. The *isochrone problem* takes as input a source vertex s and a time limit x. We say that an edge $(u, v) \in E$ is *(fully) traversable* if $\text{dist}(s, u) + \ell(u, v) \leq x$. If $\text{dist}(s, u) < x$, but $\text{dist}(s, u) + \ell(u, v) > x$, we say that the edge is *partially traversable*. Sometimes we say that a vertex $v \in V$ is *time-reachable* if $\text{dist}(s, v) \leq x$, in order to not confuse it with the definition of reachability in graph theory.

The database community [8, 37, 38, 52, 49] defines an isochrone as the minimal subgraph that covers all locations that are reachable within the time limit. More precisely, the subgraph contains all time-reachable vertices, all fully traversable edges and the reachable parts of all partially traversable edges. Efentakis et al. [30] adopt this definition for their isoGRASP algorithm. However, they do not consider parts of edges, but only output time-reachable vertices and fully traversable edges. The method in [18], which we call isoCRP, uses a somewhat different definition. It outputs all edges (u, v) that have a time-reachable tail u and a time-unreachable head v, that is, all edges (u, v) with $\text{dist}(s, u) \leq x$ and $\text{dist}(s, v) > x$. These edges "form the boundary" of the *isochrone area*, which we sometimes call *isoline*. Note that the isochrone area is not necessarily a simple polygon, but may have holes (e.g., mountain summits that are time-unreachable).

Let us take a systematic look at the different types of edges in the context of isochrones. We

can distinguish seven types of edges, see Fig. 3.1. Time-reachable vertices are shown in green, and time-unreachable vertices are shown as empty circles with red borders. Green edges are fully traversable, and red dashed edges are not traversable at all. Mixed edges denote partially traversable edges. Edges between two time-reachable vertices can be traversable in full, partially traversable or not traversable at all, since there may be other paths that ensure that the head is time-reachable. Edges with a time-unreachable tail are, of course, always not traversable.

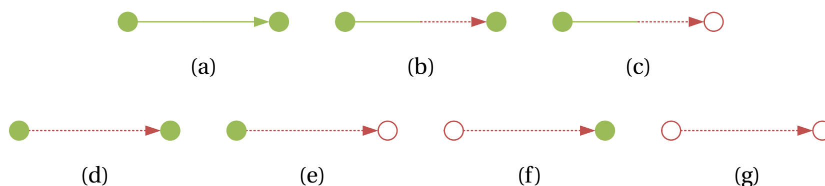


Figure 3.1 – Different types of edges in the context of isochrones.

We now can redefine the output of the algorithms mentioned above in terms of the different edge types. [8, 37, 38] report edges of type A, B and C. The isoGRASP algorithm only outputs edges of type A, and isoCRP reports edges of type C and E. Since our main motivation for computing isochrones is to visualize the isoline, it is reasonable to adopt the output of isoCRP, as these edges resemble the isoline in some sense. We call edges of type C and E *isochrone edges*. Some algorithms for visualizing isolines may need somewhat more information, namely all edges that cross the isoline. These are all isochrone edges complemented by the edges of type F. Hence, we introduce the notion of *isochrone pairs*, which denote pairs (u, v) of neighboring vertices such that u is time-reachable and v is time-unreachable. Throughout this thesis, we describe for each algorithm how to determine isochrone edges and how to determine isochrone pairs.

3.2 Isochrones for Electric Vehicles

Intuitively, we want to compute the region that is reachable with an electric vehicle's current battery charge level. Formally, we consider a directed graph $G = (V, E)$ together with two cost functions (metrics). The *routing cost function* $\ell : E \to \mathbb{R}_{>0}$ assigns each edge the time it takes to travel along the edge, and the *consumption cost function* $c : E \to \mathbb{R}$ assigns each edge the amount of electric energy required to get across the edge. In contrast to travel times, the consumption metric may take on negative function values for some downhill edges, since electric vehicles are able to recuperate energy when going downhill. Due to physical reasons, there are no negative cycles. Recall from the introduction that we use two cost functions, since energy-optimal paths would differ considerably from quickest paths [9].

The *EV variant of the isochrone problem* takes as input a source vertex s and a battery charge level b. We say that a vertex $v \in V$ is *ev-reachable* from s with the initial charge level b if

a quickest $s-v$ path obeys the battery constraints of the electric vehicle. The definitions of isochrone edges and isochrone pairs carry over to the EV scenario straightforwardly. An edge (u,v) with an ev-reachable tail u and an ev-unreachable head v is referred to as an isochrone edge. Isochrone pairs are all pairs (u,v) of neighbors such that u is ev-reachable and v is ev-unreachable. Just like in the standard scenario, the EV variant of the isochrone problem asks for all isochrone edges or, if required, isochrone pairs.

3.3 Dijkstra's Algorithm for Isochrones

Dijkstra's algorithm is a quite versatile technique which can be used in several scenarios, such as point-to-point, one-to-all and one-to-many queries. Adapting it to compute isochrones is straightforward. We run a normal Dijkstra search from the source vertex s, but stop when all elements in the priority queue have a distance label greater than the time limit x. It remains to determine the isochrone edges or, if required, isochrone pairs.

Recall that isochrone edges are all edges (u,v) with $\text{dist}(s,u) \leq x$ and $\text{dist}(s,v) > x$. To find all of them, we extract, after the search has stopped, each vertex v from the queue, loop through its incoming edges (u,v) and output each of them where u is reachable within the time limit.

Theorem 3.1. *The algorithm is correct, that is, it outputs all isochrone edges.*

Proof. We claim that the vertices that are still in the queue when the search stops are exactly the heads of the isochrone edges. Assume that the search has stopped and consider a vertex v in the queue. Due to the stopping criterion, v has to be time-unreachable. Let $(u,v) \in E$ be the edge that caused v to be inserted into the queue. Since the search scans only time-reachable vertices, u has to be time-reachable and thus (u,v) is an isochrone edge. Hence, each vertex in the queue is the head of an isochrone edge.

To prove the other direction, consider an arbitrary isochrone edge $e=(u,v)$. By definition, u has to be time-reachable and v has to be time-unreachable. Since the search scans all time-reachable vertices, u must have been scanned. When this happened, e was examined and v was inserted into the queue, if it was not there before. As the search stops when all elements in the queue have a distance label greater than the time limit, v is still in the queue after the search has terminated. Hence, the vertices in the queue are indeed exactly the heads of the isochrone edges. By looping through the incoming edges of the isochrone heads, the algorithm finds all isochrone edges. $\square$

Determining isochrone pairs requires further modifications. Recall that isochrone pairs are all pairs (u,v) such that $\text{dist}(s,u) \leq x$ and $\text{dist}(s,v) > x$ and there exists an arbitrarily directed edge that connects u and v. The above approach may not find all isochrone pairs that should be reported since Dijkstra's algorithm relaxes only outgoing edges. For example,

consider a time-unreachable vertex v that is only connected by an outgoing edge (v, u) to a time-reachable vertex u (and not by an isochrone edge). Such a vertex may not be inserted into the queue at all.

In order to properly identify these pairs, we need to adapt Dijkstra's algorithm. When scanning u, we first relax all outgoing edges as usual. However, afterwards we loop through its incoming edges (v, u) and check for each v whether it has a finite distance label (and thus was inserted into the queue once already). If not, we insert v into the queue with a key of infinity. After the search has stopped, we again extract each vertex v from the queue, loop through its incident edges and output each pair (u, v) such that u is a neighbor of v that is reachable within the time limit.

We call this variant of Dijkstra's algorithm *RangeDijkstra*. Although it works correctly, its performance is reasonable only for small time limits. In order to enable fast isochrone computations at continental scale, we need to develop speedup techniques.

4 Multilevel Dijkstra Techniques

CRP [14, 22, 19, 15] is a well-known, versatile and robust speedup technique. This makes it a good starting point for accelerating the computation of isochrones. This chapter starts by proposing a basic multilevel algorithm for computing isochrones that is one order of magnitude faster than RangeDijkstra. Afterwards, we revisit and extend two known algorithms for computing isochrones that are based on the CRP and GRASP framework, respectively.

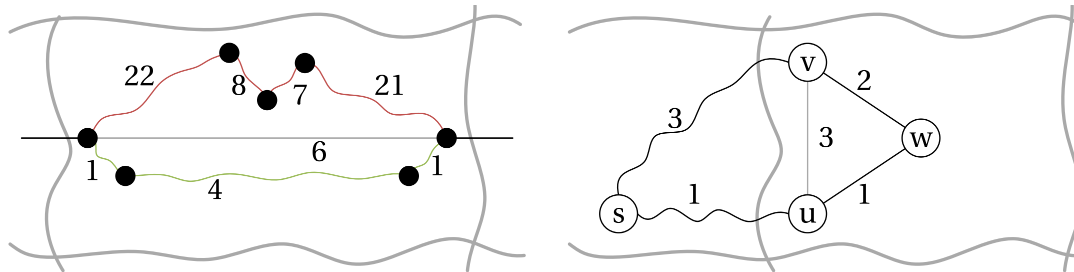
4.1 Basic Multilevel Dijkstra Algorithm for Isochrones

In the following we present a plausible approach for computing isochrone edges that takes advantage of multilevel overlay graphs. However, it will produce false negatives and false positives without further modifications. The necessary changes are described afterwards. We finish this section with an optional optimization and a discussion of how to determine isochrone pairs.

4.1.1 General Idea

We might come up with an approach whose general idea is as follows. We run a normal forward CRP search from the source vertex s, but stop when the search scans a vertex with a distance label greater than the time limit x. However, unlike a normal forward CRP search which transitions only from lower to higher levels, we want the search to descend into cells that may contain isochrone edges. More precisely, if we encounter a shortcut edge we cannot traverse without violating the time limit when scanning a vertex on its current CRP query level, we try to scan this vertex on the level below. If we again encounter a shortcut edge that is too long, we descend one level further, and so on. After the search has stopped, we determine isochrone edges or, if necessary, isochrone pairs as described above.

Unfortunately, without further modifications the above approach produces false negatives, i.e., it misses some isochrone edges, as well as false positives, i.e., it outputs edges that



(a) False negative. Although we can pass through the cell on the fast road through a tunnel (the green edges), we cannot use the mountain road (the red edges).

(b) False positive. Assuming a limit $x = 4$, we erroneously report (v, w) as an isochrone edge.

Figure 4.1 – Examples where our approach produces wrong outputs. Gray edges denote shortcut edges.

actually are no isochrone edges. However, we can get rid of both problems. False negatives may occur when isochrone edges are not part of shortcut edges, see Fig. 4.1a for an example. Imagine a mountain in the middle of the cell. The shortcut edge from the left to the right boundary vertex corresponds to a fast road through a tunnel (the green edges) and can be traversed within the time limit. However, this does not necessarily mean that we can reach the summit of the mountain via a mountain road (the red edges). Thus it may happen that we do not descend into the cell and hence miss an isochrone edge on the mountain road.

False positives may occur when we descend at some boundary vertices into a cell, but jump over the same cell at other boundary vertices. See Fig. 4.1b for an example. Assume a limit $x = 4$. When we scan u, we can traverse the shortcut edge (u, v), since the sum of u's distance label and the shortcut's length equals the limit. Thus we do not descend at u into the cell, but relax the shortcut edge. Next, we are going to scan v. This time the sum of v's distance label and the shortcut's length is greater than the limit, so we descend into the cell and relax the edge (v, w). Since w has a distance label greater than the limit and the queue contains no other vertices, the search stops. Now, we erroneously report (v, w) as an isochrone edge, since v's distance label is less than x and w's distance label is greater than x. The $s - w$ path via u that makes w time-reachable is not found.

4.1.2 Eliminating False Negatives

In order to overcome false negatives, we produce some additional auxiliary data during customization. Recall that each cell in the overlay with n boundary vertices is represented as an $n \times n$ matrix in which position (i, j) contains the length of the shortcut edge from the cell's i-th to the cell's j-th boundary vertex. We append one column to each of these matrices. Position $(i, n+1)$ then stores the eccentricity of the i-th boundary vertex restricted to the cell represented by the matrix. That is, it stores the length of the shortest path (restricted to the cell) from the i-th boundary vertex to its farthest reachable vertex in the cell.

During queries, we no longer check whether we can traverse all shortcut edges without violating the time limit x when scanning a vertex v. Instead, we check whether the sum of v's distance label and its eccentricity on the current query level are greater than x. If so, we descend into the cell and check v's eccentricity on the level below. If not, we simply relax the shortcut edges (and boundary edges, of course) on the current query level. This approach ensures that we miss no isochrone edges (we say more about how to determine isochrone pairs in section 4.1.5).

The eccentricities for all boundary vertices on all levels can be computed with almost no overhead during customization. When constructing the first overlay level, we build from each level-1 boundary vertex v a shortest path tree restricted to v's level-1 cell. The eccentricity of v on level 1 is simply the distance label of the vertex which was scanned last. However, when we build the second overlay level (or any other above), we run Dijkstra's algorithm on the overlay level below to accelerate the construction of higher levels. In this case, we maintain an upper bound on the eccentricity of v which is updated every time we scan a vertex u. If the sum of u's distance label and u's eccentricity on the level below is greater than the current upper bound, the bound is increased accordingly. This gives us safe (but not necessarily tight) eccentricities on all levels at almost no extra cost.

4.1.3 Eliminating False Positives

We move on to describe how to cope with false positives. As already mentioned, the problem occurs when boundary vertices of the same cell are scanned on different query levels. Thus we need to ensure that if the algorithm descends at any boundary vertex, it needs to do so at all other boundary vertices of the same cell. Algorithm 4.1 shows how to compute the query level of a vertex. It is called whenever we scan a vertex. The algorithm maintains two bit-vectors for each level of the multilevel partition. For each cell C on level ℓ, the bit *descend*$[\ell][C]$ indicates whether we need to descend into C and the bit *reach*$[\ell][C]$ indicates if C contains at least one vertex that is reachable within the time limit x. At this point, the bit-vectors *descend*$[\cdot]$ would actually suffice to compute isochrone edges. However, since we will need the bit-vectors *reach*$[\cdot]$ in the GRASP-based algorithm and they will become more important in the EV scenario, we already include them here. Initially, we set *descend*$[\ell][c_\ell(s)]$ and *reach*$[\ell][c_\ell(s)]$ for all levels ℓ, where $c_\ell(s)$ denotes the level-ℓ cell that contains the source s.

To compute the query level of a vertex u, we start by checking whether u's top-level cell is already marked as a cell into which we need to descend (for example, because we have already descended at another boundary vertex into this cell). If it is, we move on to check the same for u's second-highest-level cell. Finally, we will arrive at either level 0 or a level ℓ on which u's cell is not already marked. In the first case we are done. In the second, we check whether the sum of u's distance label and u's eccentricity on level ℓ is greater than x. If it is not, the query level of u is ℓ. If it is, we set the appropriate bit and reinsert all other boundary vertices of $c_\ell(u)$ into the queue, since some of them may have already been scanned without

Algorithm 4.1: ComputeQueryLevel(u)

```
1  ℓ_q ← 0
2  if u is a boundary vertex then
3      ℓ_q ← L
4      while ℓ_q > 0 do
5          if descend[ℓ_q][c_{ℓ_q}(u)] is not set or reach[ℓ_q][c_{ℓ_q}(u)] is not set then
6              if viaShortcut[u] is set then
7                  break
8              if d[u] + ecc_{ℓ_q}(u) > x then
9                  set descend[ℓ_q][c_{ℓ_q}(u)]
10             if d[u] ≤ x then
11                 set reach[ℓ_q][c_{ℓ_q}(u)]
12             if descend[ℓ_q][c_{ℓ_q}(u)] is not set or reach[ℓ_q][c_{ℓ_q}(u)] is not set then
13                 break
14             foreach boundary vertex v ∈ c_{ℓ_q}(u) do
15                 if v ≠ u and d[u] ≠ ∞ then
16                     if v ∉ Q then
17                         Q.insert(v)
18                     unset viaShortcut[v]
19         ℓ_q ← ℓ_q − 1
20 return ℓ_q
```

descending into $c_\ell(u)$. We also set *reach*[ℓ][$c_\ell(u)$] accordingly. We then move on to check whether we need to descend even further.

Having eliminated the false negatives and false positives, we now have a correct algorithm, which is already much faster than RangeDijkstra. We call it *single-phase isoCRP algorithm*, in contrast to the two-phase algorithm which is described in the next section. However, before we move on, we point out an optimization and also come back to isochrone pairs.

4.1.4 Further Optimization

In order to avoid some unnecessary descents, we maintain another bit-vector with as many bits as there are boundary vertices. The bit *viaShortcut*[v] indicates whether v was reached via a shortcut edge (as opposed to via a boundary edge). If it was, we do not have to ensure that the sum of v's distance label and its eccentricity is not greater than x. Whether we need to descend into its cell or not was already determined when we scanned the parent of v at the other end of the shortcut edge.

4.1.5 Determining Isochrone Pairs

We now come back to isochrone pairs. Unfortunately, it does not suffice to insert neighbors that still have an infinite distance label after the outgoing edge relaxations into the queue with a key of infinity. See Fig. 4.2 for an example. Assume a limit $x = 2$. Recall that $\mathrm{ecc}(u)$ denotes the eccentricity of u and note that $\mathrm{ecc}(u) = 1$ since w is the farthest vertex in the cell that is reachable from u. Now, when we scan u, we do not descend into the cell since $d[u] + \mathrm{ecc}(u) \leq x$. Afterwards the search stops as the single vertex v which remains in the queue has a distance label greater than x and we miss the isochrone pair (w, v).

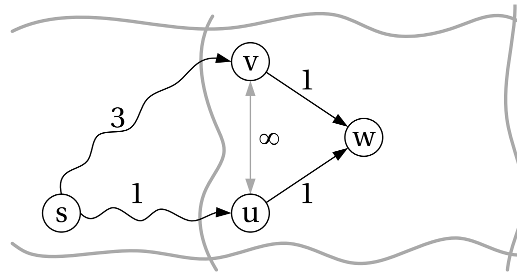


Figure 4.2 – Example where an isochrone pair is missed. The gray edge denotes a shortcut edge. Assuming a limit $x = 2$, we miss the isochrone pair (w, v).

The above example fails because the cell is no strongly connected component. There actually is a vertex in the cell that is not reachable within the limit, namely v, but we are not aware of it when scanning u, since there is no $u - v$ path within the cell. In order to ensure correct results in the presence of weakly connected cells, we set $\mathrm{ecc}(u)$ to infinity whenever there is at least one vertex v in the cell such that no $u - v$ path exists within the cell. Alternatively, one may not restrict searches during customization to the cells, but may search on the whole (overlay) graph until each vertex in the cell has been scanned. This would give us tighter eccentricities, but would probably increase the customization time significantly.

4.2 Improved Multilevel Dijkstra Algorithm for Isochrones

The algorithm in the previous section has two significant drawbacks. First, it suffers from bad locality and cache efficiency. Recall that, within the CRP framework, boundary vertices of the top level have the lowest IDs, followed by boundary vertices of the second highest level, and so on. Since we descend directly, the locality of the accesses to the distance labels and other data structures is far from perfect. Second, the algorithm is hard to parallelize.

Note that when scanning an overlay vertex in the basic multilevel algorithm from the previous section, we do not know its final query level for sure. It can happen that we scan it on a level that is too high. To solve this problem, when we descend into a cell for the first time, we reinsert all other boundary vertices of this cell into the queue (cf. section 4.1.3). Alternatively, we may first scan all vertices of the (top-level) overlay graph until we reach a vertex with a

distance label greater than the time limit. Afterwards, we know for sure at which vertices of the (top-level) overlay graph we need to descend and at which not. This idea leads to an algorithm that has much better locality and is easy to parallelize. The algorithm was outlined before in a patent specification [18] about query scenarios for CRP. However, to the best of our knowledge, there has been no scientific publication that actually evaluated the algorithm on large road networks. Furthermore, the patent specification only considers a single-core implementation and discusses no parallelization techniques.

The customization stage remains unchanged, that is, we still run a normal CRP customization and compute eccentricities on-the-fly, as described in the previous section. However, the query stage now consists of two phases. First, it performs a normal forward CRP search from the source vertex s until all elements in the priority queue have a distance label greater than the time limit x. In other words, it runs RangeDijkstra on the graph consisting of the union $c_1(s) \cup \cdots \cup c_{L-1}(s) \cup H_L$, that is, the union of the top-level overlay graph H_L and for each level ℓ the cell $c_\ell(s)$ that contains s. This will be referred to as the *upward phase*.

We need to apply one extension to RangeDijkstra. Assume it scans a vertex u on an overlay level ℓ. Immediately after relaxing u's outgoing shortcut edges, it checks whether $d[u] + \mathrm{ecc}_\ell(u) > x$. If it is, it marks $c_\ell(u)$ as an *active cell*. These are cells that have at least one boundary vertex from which we cannot jump over the cell. The check is cache efficient, and will therefore cause almost no direct overhead. The distance label $d[u]$ is read anyway and the eccentricity $\mathrm{ecc}_\ell(u)$ is placed directly behind u's shortcut edges in memory.

The *downward phase* consists of L subphases. The first subphase runs, for each active level-L cell C, RangeDijkstra in H_{L-1} (restricted to C) until all elements in the priority queue have a distance label greater than x or the queue is empty. Initially, we insert all boundary vertices of C into the queue using their distance labels as keys. We apply the same extension to RangeDijkstra as before. The second subphase proceeds to run RangeDijkstra on each active level-($L-2$) cell, and so on. Finally, the last subphase runs RangeDijkstra on each active level-1 cell and we are done.

In the end, we scan the same vertices as we did in the previous section, but in a different order than before. This has two advantages. First, we have much better locality since we process one level after the other and inside each level one cell after the other. Second, it is easier to exploit parallelism. Algorithm 4.2 summarizes the new approach which we call *two-phase isoCRP algorithm* or simply *isoCRP*. Note that we again apply the optimization technique to avoid some unnecessary descents from the previous section.

4.2.1 Determining the Output

Recall that RangeDijkstra finds all isochrone edges among the edges it visited. To do so, it extracts, after the search has stopped, each vertex v from the queue, loops through its incoming edges (u, v) and reports each of them where u is reachable within the time limit.

Algorithm 4.2: isoCRP(s, x, G, H)

```
1 set distance label d[v] to ∞ for all v ∈ G
2 unset descend[ℓ][C] and reach[ℓ][C] for all 1 ≤ ℓ ≤ L, C ∈ 𝒞^ℓ
3 RangeDijkstra(s, x, c_1(s) ∪ ··· ∪ c_{L−1}(s) ∪ H_L, d, descend, reach)   // upward phase
4 for ℓ ← L to 1 do                                                          // downward phase
5     foreach cell C ∈ 𝒞^ℓ do
6         if descend[ℓ][C] is set and reach[ℓ][C] is set then
7             RestrictedRangeDijkstra(C, x, d, descend, reach)
```

Thus, the algorithm as described above gives us the isochrone edges. Of course, we need to loop only through incoming boundary edges (and not through incoming shortcuts).

More precisely, isochrone edges that are boundary edges on the top level are determined during the upward phase. The same is true for isochrone edges that are boundary edges on level l and contained in $c_{l+1}(s)$. For all other isochrone edges (u, v) it holds that they are determined when we run RangeDijkstra on $c_{l+1}(u)$, where l is the highest level on which (u, v) is a boundary edge.

As described before, RangeDijkstra is also capable of computing isochrone pairs, and so is isoCRP. Note, however, that in the presence of weakly connected cells, we again need to set ecc(u) to infinity whenever there is at least one vertex v in the cell such that no $u - v$ path exists within the cell or, alternatively, we need to spend more time during customization.

4.2.2 Parallelization

It remains difficult to parallelize the upward phase since it seems to be inherently sequential. However, this is not a big problem, as the time spend on the upward phase is only a small fraction of the running time. Thus we focus on the downward phase. Note that active cells on the same level can be processed in parallel, but we need to process each cell after its supercells (if any). This allows us to parallelize the downward phase in the same fashion as the customization stage. More precisely, we assign active cells on the same level to distinct cores and synchronize the threads at each level of the partition. This scheduling strategy will be referred to as *"distribute cells per level"*. Since the threads operate on distinct vertex sets, there are no concurrent accesses to the distance labels, so we can share the labels among the threads. However, each thread needs its own priority queue.

There are two potential problems with the above strategy. First, we need to synchronize the threads at each level. Although CRP typically uses only a handful of levels, this may result in a performance penalty. Second, if distinct cores often process cells whose vertices are consecutive in memory, false sharing may become a bottleneck. False sharing occurs when distinct cores operate on independent data elements that reside on the same cache line.

Then, each update invalidates the cache line on all other cores, although there truly is no sharing. We reduce the risk of false sharing significantly by assigning chunks of consecutive cells (as opposed to assigning single cells) to distinct cores. However, we still have the synchronization overhead.

Alternatively, we can process each top-level cell in top-down fashion independent from the others. This scheduling strategy will be referred to as *"distribute top-level cells"*. It requires no synchronization constructs at all and false sharing is highly unlikely. However, the amount of parallelism is relatively small. Typically, CRP uses about 20 cells on the top level, which restricts the number of concurrent tasks to 20. Frequently, the number is much less, for example when the time limit is so small that we do not leave the top-level cell that contains the source vertex. In this case, we can exploit no parallelism at all.

The first strategy provides a good amount of parallelism, whereas the second strategy provides no synchronization overhead and practically no false sharing. Combining them may give us the best scheduling strategy. We first assign the active top-level cells to distinct cores. Each core then runs RangeDijkstra on its cells. After that, we synchronize the threads once and assign the larger number of active second-highest-level cells to distinct cores. Now, each core processes its cells in top-down fashion. This will be referred to as *"combined scheduling strategy"*. It combines the advantages of the other strategies.

4.3 GRASP for Isochrones

More recently, Efentakis et al. [30] developed GRASP, an extension of CRP to compute batched shortest paths efficiently. Besides one-to-all and one-to-many algorithms, they also propose a technique called *isoGRASP* for computing isochrones. However, what they do is somewhat different from our objectives introduced in section 3.1. First, they compute all edges (u, v) such that $\text{dist}(s, u) + \text{len}(u, v) \leq x$, that is, all edges that can be traversed within the time limit x. More importantly, they compute the correct distance labels for all vertices that are reachable within x. In other words, they do not jump over cells that contain only reachable vertices. While this may be needed by several applications, we require no shortest-path distances at all. This section describes our variant of isoGRASP, which is capable of jumping over cells and outputs isochrone edges or, if required, isochrone pairs.

Our customization stage runs a normal CRP customization and generates downward edges on-the-fly, just like the customization of the original GRASP algorithm (cf. section 2.4.1). Additionally, we compute eccentricities in the same way as we did in the previous sections. Our query stage consists of two phases. The *upward phase* is identical to the one of isoCRP. The *scanning phase* processes all levels from top to bottom in L subphases, again just like the downward phase of isoCRP. However, instead of running RangeDijkstra on each active level-$(L-i+1)$ cell, the i-th subphase of our isoGRASP algorithm scans all level-$(L-i+1)$ downward edges within each active level-$(L-i+1)$ cell. Algorithm 4.3 gives the details.

Algorithm 4.3: isoGRASP(s, x, G, H, $G^{\downarrow}_{GS}$)

```
1  set distance label d[v] to ∞ for all v ∈ G
2  unset descend[ℓ][C] and reach[ℓ][C] for all 1 ≤ ℓ ≤ L, C ∈ 𝒞^ℓ
3  RangeDijkstra(s, x, c_1(s) ∪ ··· ∪ c_{L−1}(s) ∪ H_L, d, descend, reach)   // upward phase
4  for ℓ ← L to 1 do                                                          // scanning phase
5      foreach cell C ∈ 𝒞^ℓ do
6          if descend[ℓ][C] is set and reach[ℓ][C] is set then
7              foreach internal vertex v ∈ C do
8                  foreach downward edge (u, v) ∈ G↓_GS do    // compute correct distance labels
9                      if d[u] + len(u, v) < d[v] then
10                         d[v] ← d[u] + len(u, v)
11                 if ℓ > 1 then                                             // check for active cells
12                     if d[v] + ecc_{ℓ−1}(v) > x then
13                         set descend[ℓ − 1][c_{ℓ−1}(v)]
14                     if d[v] ≤ x then
15                         set reach[ℓ − 1][c_{ℓ−1}(v)]
16             if ℓ > 1 then                                                 // check for active cells
17                 foreach boundary vertex v ∈ C do
18                     if d[v] + ecc_{ℓ−1}(v) > x then
19                         set descend[ℓ − 1][c_{ℓ−1}(v)]
20                     if d[v] ≤ x then
21                         set reach[ℓ − 1][c_{ℓ−1}(v)]
22             foreach internal vertex u ∈ C do                              // determine isochrone edges
23                 foreach original edge (u, v) ∈ G do
24                     if v < u and d[u] ≤ x and d[v] > x then
25                         output (u, v)
26                 foreach original edge (v, u) ∈ G do
27                     if v < u and d[u] > x and d[v] ≤ x then
28                         output (v, u)
```

To process an active level-ℓ cell C, Alg. 4.3 loops through its internal vertices, which are all the level-$(\ell-1)$ overlay vertices in the interior of C. For each internal vertex v, we scan its incoming downward edges and update the distance label accordingly. Having the correct distance label, we check whether v's cell on the level below is an active cell, which is the case if $d[v] + \text{ecc}_{\ell-1}(v) > x$. Afterwards, we need to do the same check also for all boundary vertices of C.

As mentioned earlier, the bit-vectors *reach*[·] become important now. We need them to avoid descending into cells where no boundary vertex is reachable within the time limit. This cannot happen in the isoCRP algorithms, since RangeDijkstra processes vertices in increasing order of distance from s, and stops when all elements in the priority queue have a distance label greater than x. Thus we never set $descend[l][C]$ for a level-l cell C with no boundary vertex reachable within x. However, when we process a cell in our isoGRASP algorithm, we process its internal vertices in no specific order and thus have no stopping criterion. Hence, we set the descend bit for unreachable cells. By descending only into cells for which both bits are set, we avoid a considerable amount of work.

4.3.1 Determining the Output

Since the upward phase is identical to the one of isoCRP, it also determines isochrone edges that are boundary edges on level l and contained in $c_{l+1}(s)$. For all the other isochrone edges (u,v) it holds that they are determined when we process $c_{l+1}(u)$, where l is the highest level on which $\max\{u,v\}$ is a boundary vertex (recall that vertices of higher levels have lower IDs). To do this, we loop through the internal vertices once again after having computed their correct distance labels. For each internal vertex u, we output each outgoing edge (u,v) with $d[u] \leq x$ and $d[v] > x$ and each incoming edge (v,u) with $d[u] > x$ and $d[v] \leq x$. To avoid duplicates, we output an edge incident on an internal vertex u only if $v < u$. This check also prevents us from testing edges that connect u to vertices on levels below (which do not have correct distance labels yet). See also Alg. 4.3.

We may determine isochrone pairs in the same way. However, note that in the presence of weakly connected cells, we need to take the same measures as before, that is, set some eccentricities to infinity or spend more time during customization.

4.3.2 Parallelization

The original isoGRASP algorithm assigns active top level cells to distinct cores and processes each of them in top-down fashion. This corresponds to our "distribute top level cells" strategy. However, we may use any of the scheduling strategies described above to parallelize the query stage. Their advantages and drawbacks remain the same.

We now describe how we parallelize the customization stage of GRASP, since the original publication does not elaborate on this algorithmic aspect. Of course, we can build an empty downward graph data structure into which each core inserts downward edges incrementally. However, this needs excessive locking and leads to poor parallel performance. We can do better without locking.

We maintain one edge array for each core. The downward edges into each vertex are stored consecutively in one of these arrays. We also have a vertex array V of size n (the number of

vertices). $V[i]$ stores the index of the edge array that contains v_i's edges, as well as the starting and the ending position in this array of v_i's edges. When processing a cell, we additionally maintain one temporary bucket $B(v)$ for each internal vertex v. After running Dijkstra's algorithm from the boundary vertex u, we add each downward edge from u to an internal vertex v to $B(v)$. When we are done with all boundary vertices of the cell, we concatenate all these little buckets, append them to the core's edge array and set the entries in the vertex array accordingly. Finally, when the whole CRP customization is finished, the master core builds the actual downward graph with a single sweep through the vertex and edge arrays.

5 Combining Graph Separators and Contraction Hierarchies

Besides CRP, Contraction Hierarchies [40] are another widely known and flexible speedup technique, which is even faster for "well-behaved" metrics such as travel times [15]. There are also fast one-to-all (PHAST [13]) and one-to-many (RPHAST [17]) algorithms that build upon Contraction Hierarchies. In this chapter, we use Contraction Hierarchies and (R)PHAST to compute isochrones. In order to do so, we combine Contraction Hierarchies with different graph separators. The first section describes the basic approach of the algorithms in this chapter. Afterwards, we detail this approach when using edge separators, which has some similarity with CRP. However, since the algorithms that build upon edge separators suffer from some drawbacks, we switch to vertex separators in the last section.

5.1 Basic Algorithm

Intuitively, it seems difficult to use Contraction Hierarchies and related techniques to compute isochrones, due to the inherent bidirectional nature of CH. All common types of queries (point-to-point, one-to-all and one-to-many) take the targets as input. However, when computing isochrones, we do not have a priori information about the targets. We need to compute correct distance labels at least for all vertices that are endpoints of an isochrone edge. Otherwise, we would not be able to tell whether an edge is an isochrone edge or not. However, these vertices are not known in advance. After all, computing them is exactly the aim of isochrone queries.

But even if the targets are not known, we may be able to restrict the area that contains isochrone edges to certain regions. Then, we may use (R)PHAST to compute correct distance labels for all vertices in these regions. In [17], Delling et al. conducted the following experiment. Using the standard Dijkstra search, they grew a ball B of varying size $|B|$ centered on a random vertex. Then, they used RPHAST to compute the distances between a source s and all vertices in B. On their standard workstation, RPHAST took only 0.17 ms for a ball of size $|B| = 2^{14}$ on a single core. This promising query time motivates the basic approach of the algorithms in this chapter, which is as follows.

During preprocessing, we partition the graph into cells of roughly equal size. Although the cells are not perfect balls, the vertices in a cell are close together, so that computing distances between a source s and all vertices in a cell should not take much longer than the RPHAST queries from the experiment that was mentioned above. The query stage generally consists of three phases:

1. Perform a forward CH search from the source s.
2. Determine the cells that may contain isochrone edges (so-called *active cells*).
3. Use (R)PHAST to process the previously determined cells.

There may be a fourth phase, which determines isochrone edges that could not be determined on-the-fly during one of the other phases. We call the techniques that follow this basic approach *GS+PHAST algorithms*, since their key ingredients are graph separators and (R)PHAST. See Fig. 5.1 for an illustration. Cells that are colored gray denote active cells. Note that it may happen that some cells are marked as active, although they are not intersected by the isoline. The number of such cells depends on the method used for determining active cells. Also note that besides the active cells along the primary isoline, there may be active cells in the interior of the area bounded by the primary isoline, for example due to mountains whose summits we cannot reach within the time limit.

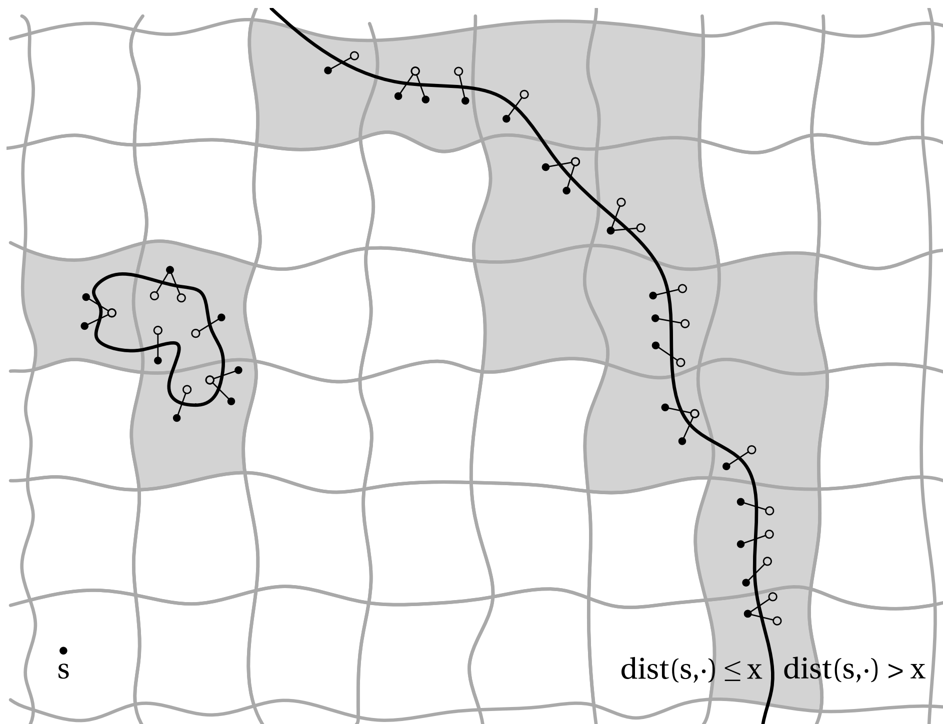


Figure 5.1 – Illustration of the GS+PHAST algorithms. The thick black lines denote the isoline, which is intersected by isochrone edges (thin black lines). Cells that are colored gray denote active cells. Note that there are three cells that are "unnecessarily" active.

We develop different variants of the above approach, which differ (among other respects) in the type of graph separators they use. With CRP in mind, it is natural to start by using edge separators, which we do in the next section. Later, we switch to vertex separators in order to remedy some drawbacks of the variants based on edge separators.

5.2 Edge Separators

Recall that edge separators are closely related to partitions, that is, a k-way edge separator can be defined as the set of boundary edges of a k-way partition. Of course, we can use edge separators to decompose road networks into several preferably balanced cells. The cell boundaries then cross the separator edges, so that separator edges cannot be assigned to a single cell. On the other hand, each vertex is contained in exactly one cell.

We call the techniques in this section *ES+PHAST algorithms*. The three variants described in the next sections use three different methods to determine active cells. We start with a variant that has some similarity with CRP.

5.2.1 Core-Dijkstra

Probably the most straightforward approach to determine active cells is to run RangeDijkstra on the core graph. Here, the core graph is an overlay graph that contains all boundary vertices, all boundary edges and the shortcuts needed to preserve the distances between all boundary vertices. This is similar to the overlay graphs that are used for CRP, except that we always use only a single level. We also store eccentricities for each core vertex. Whenever the Dijkstra search scans a core vertex u with $d[u]+\text{ecc}(u) > x$, where $d[u]$ denotes u's distance label, $\text{ecc}(u)$ denotes u's eccentricity and x denotes the time limit, it marks u's cell $c(u)$ as active. We now discuss the different aspects of this *Core-Dijkstra variant* of the ES+PHAST algorithm in more detail.

Preprocessing Stage

The preprocessing stage consists of seven steps. We first give a brief overview of the different steps and then elaborate on each of them. Note that all algorithms in this chapter do not make a distinction between metric-independent preprocessing and metric-dependent customization. However, we will give hints towards customizable implementations in chapter 8. The preprocessing steps are as follows.

1. Reorder vertices according to the edge separator.
2. Compute topological data.
3. Build cell graphs.
4. Contract each cell graph.

5. Reorder vertices in each cell by level.
6. Build upward and downward graph.
7. Compute eccentricities for all overlay vertices.

Reorder vertices according to the edge separator. The first preprocessing step is to assign new IDs to vertices. Core vertices are pushed to the front, using their cells as a tiebreaker. Non-core vertices are pushed to the back, using the same tiebreaker as before. The new vertex order has two advantages. First, it simplifies mapping between core and original vertices. Second, we have better locality and cache efficiency when processing the active cells.

Compute topological data. Next, we compute some topological data that is needed during preprocessing and queries. Due to the previous step, for each cell the IDs of its boundary vertices are contiguous. The same is true for its internal vertices. Hence, we can assign to each cell a contiguous range of boundary vertices and of internal vertices. We also store for each vertex the cell that contains it.

Build cell graphs. The preprocessing stage needs to contract the internal vertices of each cell. To so do, we can run a standard CH preprocessing on the graph, but block the contraction of core vertices. While this works correctly, its performance is suboptimal. Vertices that are contracted in succession may be spread over the whole graph. That leads to many cache misses and thus increases running time. It is actually faster to build a dedicated graph for each cell and then contract these small graphs. For each cell, the cell graph is the subgraph induced by its vertices.

Contract each cell graph. Now, we actually contract the cell graphs. Basically, we run a standard CH preprocessing on each cell graph, but block the contraction of the boundary vertices. The vertex order is determined online and bottom-up. A vertex v is contracted as usual, that is, we temporarily remove it from the graph and create a shortcut between each pair u, w of neighbors if the shortest $u - w$ path is unique and contains v.

Reorder vertices in each cell by level. The PHAST algorithm requires that the vertices are ordered by level. Vertex levels can be computed during the CH preprocessing. Initially, we set $L(v) = 0$ for all vertices v. When contracting v, we set $L(v) = \max\{L(v), L(u) + 1\}$ for each non-contracted neighbor u. We reorder the internal vertices in each cell according to these levels. The boundary vertices (which have the lowest IDs) keep their relative order.

Build upward and downward graph. During queries, we need the core graph and for each cell its upward and downward graph. As mentioned before, the core graph contains all boundary vertices and all original edges between two boundary vertices. Furthermore, it contains the shortcuts from the CH preprocessing that connect two core vertices. It is convenient for the query stage to store the upward graphs of all cells and the core graph in a single graph data structure. We store all downward graphs in another graph data structure. More precisely, we build two graphs $G^{\uparrow}$ and $G^{\downarrow}$ in this step. The graph $G^{\uparrow}$ stores at each

non-core vertex u the outgoing upward edges (u, v) from the cell graph of $c(u)$. At each core vertex u, $G^{\uparrow}$ stores the outgoing forward edges (u, v) from the core graph. Conversely, $G^{\downarrow}$ stores at each non-core vertex v the incoming downward edges (u, v) from the cell graphs. At core vertices, $G^{\downarrow}$ stores no edges at all.

Compute eccentricities for all overlay vertices. Recall that the eccentricity of a core vertex v is the length of the shortest path (restricted to $c(v)$) from v to its farthest reachable vertex in $c(v)$. Of course, we could compute eccentricities by running Dijkstra's algorithm in the original graph (restricted to the cell) from each core vertex v until the queue is empty. Then, the distance label of the vertex that was scanned last is the eccentricity of v. However, we can do better by using the data that we have already precomputed. To compute ecc(u), we first run Dijkstra's algorithm on the core graph, restricted to $c(u)$. This assigns all reachable boundary vertices of $c(u)$ the same distance value as the first approach. Additionally, we assign a distance value of infinity to all unreachable boundary vertices of $c(u)$. In a second step, we use PHAST's linear sweep through the internal vertices of $c(u)$ to propagate distance values to non-core vertices. After processing an internal vertex v, we check whether its distance label $d[v]$ is finite. If it is, we update the eccentricity by setting $\text{ecc}(u) = \max\{\text{ecc}(u), d[v]\}$. If it is not, there is no $u - v$ path within $c(u)$ and thus the vertex v is not relevant for the eccentricity of u.

Query Stage

As mentioned above, the query stage of the ES+PHAST algorithms generally consists of three phases: a forward CH search from the source s, the determination of active cells and a downward phase that processes the active cells using (R)PHAST. However, the Core-Dijkstra variant of ES+PHAST does not strictly distinguish the first two phases. They rather merge into each other.

A query starts by running RangeDijkstra from the source s in $G^{\uparrow}$. Since $G^{\uparrow}$ stores at each non-core vertex u the outgoing upward edges (u, v), we perform a forward CH search as long as we are in the interior of the cell that contains s. When the search extracts a core vertex from the queue, it "automatically" switches to the second phase (the search on the core graph), since $G^{\uparrow}$ stores at each core vertex the forward edges from the core graph. Here, we exploit that the upward graphs of the cells and the core graph are stored in the same graph data structure. Note that by construction of $G^{\uparrow}$, we never relax edges going from a core vertex to a non-core vertex during the search on the core graph.

In order to determine active cells, we check, when scanning a core vertex v, whether the sum of v's distance label $d[v]$ and its eccentricity ecc(v) is greater than the time limit x. If it is, we mark the cell $c(v)$ as active. If $d[v] + \text{ecc}(v) \leq x$, all vertices in $c(v)$ reachable from v are time-reachable and thus there can be no isochrone edges (or pairs) in the part of the cell that is reachable from v. After the RangeDijkstra search terminated, we know which cells are active and proceed to the third phase.

Now, we process each active cell (including the cell that contains s) by running PHAST's linear sweep on it. More precisely, to process an active cell C, we start by checking whether each of its boundary vertices has a valid distance label. Here, a distance label is called valid when it has been set during the RangeDijkstra search on the core graph. If the distance label of a boundary vertex v is not valid, the search terminated before visiting v and thus v is not reachable within the time limit x. Hence, we can safely set v's distance label to infinity. Next, we perform a linear sweep through the internal vertices of C, which propagates distance values from the core vertices to the internal vertices. This approach computes correct distance labels for all vertices that are contained in an active cell and that are reachable within x. Vertices in active cells that are not reachable within x may have incorrect labels. This happens because we may not have correct distance labels for all boundary vertices, as mentioned above. However, these incorrect labels are no problem, since for all vertices v that are not reachable within x it holds that $d[v] > x$. After all, this is enough to safely decide if an edge is an isochrone edge or not.

Determining the Output

We have to distinguish two types of isochrone edges: those that connect two core vertices and those that are incident to at least one internal vertex. Isochrone edges of the first type are visited during the RangeDijkstra search on the core graph. Recall that RangeDijkstra finds all isochrone edges among the edges it visited. To do so, it extracts, after the search has stopped, each vertex v from the queue, loops through its incoming edges (u, v) and reports each of them where u is reachable within the time limit. It remains to determine the isochrone edges that are incident to at least one internal vertex.

By definition, isochrone edges of the second type can occur only in active cells, so we determine them during the linear sweeps. When scanning a vertex v, all of its higher ranked neighbors u already have final distance labels, since we process the vertices in descending order of level. In addition, after examining each incoming downward edge $(u, v) \in G^{\downarrow}$, the distance label of v is also final. Thus we can loop through the incoming downward edges a second time and check which of them is an isochrone edge. To do so, we basically output those that have one endpoint that is reachable within the time limit and one that is not.

There are three difficulties with the above approach: First, each isochrone edge that we output should be an original edge. However, there may also be some shortcuts among the downward edges in $G^{\downarrow}$. To exclude these shortcuts, we store a bit with each downward edge that indicates whether it is an original edge or not. Second, there may be pairs of vertices u, v that are connected by an edge in the original graph, but not in $G^{\downarrow}$. For example, assume that u has a lower level than v and that there is only an original edge going from u to v, but not the other way round. Since this edge goes upward, it is contained in $G^{\uparrow}$. However, there may be no edge between u and v in $G^{\downarrow}$ and thus we may not recognize them as neighbors during the linear sweeps. In order to handle such cases, we need to ensure that if two vertices

are connected in the original graph, there must also be an edge in $G^{\downarrow}$ that connects these vertices. Such "artificial" downward edges are assigned a length of infinity, which ensures that they do not influence the computation of the distance labels.

We now come to the third problem. What we have described so far determines isochrone pairs rather than isochrone edges. To determine the latter, we not only need to ensure that one endpoint of the downward edge is reachable within the time limit and the other is not. Instead, we have to output only those downward edges that have a reachable tail and an unreachable head. Consider a downward edge (u, v) with a length of infinity. We know that it is an artificial downward edge and that v is actually the tail and u is actually the head in the original graph. Thus we output it only if v is reachable and u is not. Now, consider a downward edge (u, v) with a finite length. This time, we know that it is indeed a downward edge and output it if the tail u is reachable and the head v is not. However, if it is the other way round, that is, if v is reachable and u is not, the downward edge can also be a valid isochrone edge. This is the case, when there is a bidirectional edge between u and v in the original graph. In order to recognize such cases, we mark each downward edge (u, v) in $G^{\downarrow}$ with an additional bit as bidirectional, if there is also an edge (v, u) in the original graph.

Having resolved the three problems mentioned above, the algorithm now correctly outputs isochrone edges. Determining isochrone pairs is even simpler, since we do not need to check whether an downward edge in $G^{\downarrow}$ actually represents an upward, downward or bidirectional original edge. We simply output each downward edge that has one endpoint that is reachable within the time limit and one endpoint that is not. However, as already described in chapter 4, we need to be careful in the presence of weakly connected cells. We can ensure correct results as follows. Either we need to set $\text{ecc}(u)$ to infinity whenever there is at least one vertex v in the cell $c(u)$ such that no $u - v$ path exists within $c(u)$ or, alternatively, we may not restrict Dijkstra searches to the cells when computing eccentricities, but may search on the whole core graph until each boundary vertex in the cell has been scanned. See Section 4.1.5 for further details.

Parallelization

Both the preprocessing and the query stage can be parallelized straightforwardly. We start with the preprocessing stage. After executing the first two steps sequentially, we process each cell in parallel. To do so, we assign cells to distinct cores. Each core then first builds the cell graph, contracts the internal vertices of the cell, inserts artificial downward edges necessary to determine the output, and finally reorders the internal vertices by level. Afterwards, we have to synchronize the threads. The master core continues to build the upward and downward graph sequentially. The last step, computing the eccentricities, is done in parallel again, by assigning cells to distinct cores. Since the cores operate on distinct vertex sets, there are no concurrent accesses to the distance labels, and we can share the labels among the cores. However, each core needs its own priority queue.

Let us come to queries. It is difficult to parallelize the RangeDijkstra search since it seems to be inherently sequential. Hence, we focus on the linear sweeps. Since there are no dependencies between different active cells, we can process them in parallel. Again, we can share the distance labels among the cores.

In order to accelerate the linear sweeps even further, we take the computer architecture of modern machines into consideration. Nowadays, most multi-socket systems have more than one NUMA node. Processors in such systems can access memory that is assigned to their NUMA node faster than memory that is assigned to different NUMA nodes. To exploit such computer architectures, we store the downward graph $G^{\downarrow}$ once on each NUMA node. During queries, each core uses the copy of $G^{\downarrow}$ on the NUMA node which its processor is assigned to.

5.2.2 Core-PHAST

The Core-Dijkstra variant of the ES+PHAST algorithm, which was described in the previous section, works reasonably fast if the time limit is small. However, if we need to search a large portion of the core graph, the Dijkstra search is a performance bottleneck, especially when processing the active cells in parallel. To do better, one may want to use another algorithm to determine active cells. Since we use PHAST for the active cells, it seems natural to use it also on the core graph for determining which of the cells are active. We call this the *Core-PHAST variant* of ES+PHAST.

Preprocessing Stage

The preprocessing stage starts by performing the same seven steps as in the preprocessing of the Core-Dijkstra variant. That is, it first reorders the vertices according to the edge separator and then it computes some topological data, such as the ranges of vertices for each cell. Following this, it contracts the internal vertices of each cell and reorders them by level. Finally, it builds the upward graph $G^{\uparrow}$ and downward graph $G^{\downarrow}$ and computes eccentricities for all core vertices. There are no differences in these steps compared to what we described in section 5.2.1. What follows are three additional steps that the preprocessing stage of the Core-PHAST variant has to take.

Since we want to use PHAST on the core graph, we obviously need to contract the core graph. To do so, we run a standard CH preprocessing on it. Afterwards, we also have to reorder the core vertices according to their levels. As usual, vertices on higher levels are pushed to the front. Lower ranked vertices follow behind. In the third additional step, we update the graphs $G^{\uparrow}$ and $G^{\downarrow}$ such that they also contain the upward and downward edges from the contracted core graph. At this point, $G^{\downarrow}$ stores at each non-core vertex v the incoming downward edges (u, v). At core vertices, it stores no edges at all. We update $G^{\downarrow}$ such that it stores the incoming downward edges from the contracted core graph.

Apart from the outgoing upward edges at non-core vertices, in $G^{\uparrow}$ also stores the forward edges from the non-contracted core graph at core vertices so far. We need these edges not only for the query stage of the Core-Dijkstra variant, but also to compute eccentricities. However, since they are not needed for the query stage of the Core-PHAST variant, we now remove them from $G^{\uparrow}$ after preprocessing and instead store at each core vertex the outgoing upward edges from the contracted core graph. This finishes the preprocessing stage.

Query Stage

Queries consist of the three distinct phases that were mentioned in section 5.1. The first phase performs a standard forward CH search. That is, it runs Dijkstra's algorithm from the source s in $G^{\uparrow}$ and stops when the priority queue becomes empty. The next phase determines active cells and computes correct distance labels for all core vertices. As already mentioned, we do so by performing a linear sweep through the core vertices, which propagates distance values from higher ranked to lower ranked vertices.

To process a vertex v, we first examine each incoming downward edge $(u, v) \in G^{\downarrow}$. If $d[u] + \ell(u, v) < d[v]$, we set $d[v] = d[u] + \ell(u, v)$, just like in a standard PHAST query. Afterwards, we check whether $d[v] + \mathrm{ecc}(v) > x$, that is, whether the sum of v's distance label and its eccentricity is greater than the time limit. If it is, we mark v's cell as active. Additionally, we check if $d[v] \leq x$, and if so, we mark $c(v)$ as reachable.

After the linear sweep through the core vertices terminated, we move on to process individual cells. Since isochrone edges can only be in cells that have at least one boundary vertex that is reachable within the time limit, we only consider active cells that are additionally marked as reachable. We also always process the cell that contains s, since we have to do so if the forward CH search does not leave the cell. The cells are processed by a linear sweep over their internal vertices, exactly as described for the Core-Dijkstra variant of ES+PHAST.

Determining the Output

Again, we need to distinguish the two types of isochrone edges: those that connect two core vertices and those that are incident to at least one internal vertex. The latter are determined, as in the Core-Dijkstra variant, during the linear sweeps through the active cells. Isochrone edges of the first type are now determined during the linear sweep over the core vertices, using exactly the same approach as the sweeps through the individual cells. As a consequence, we need to insert some artificial downward edges also into the contracted core graph and mark original as well as bidirectional edges in it with special bits, as was described before.

Determining isochrone pairs works as in the Core-Dijkstra variant. However, note that we neither have to set some eccentricities to infinity, nor do we need to extend the Dijkstra searches (during the computation of the eccentricities) across the cell borders (cf. sec-

tion 4.1.5). The reason is that the Core-PHAST variant always visits all boundary vertices of each cell. Consider an isochrone edge (u, v) inside a cell C and let v be the endpoint that is not reachable within the time limit. Since we assume the network to be strongly connected, there must be some boundary vertex w in C such that there is a $w - v$ path within the cell. When the linear sweep on the core vertices visits w, it must hold that $d[w] + \text{ecc}(w) > x$ and thus C is marked as active.

Parallelization

We parallelize the preprocessing stage of the Core-PHAST variant of ES+PHAST in the same way as that of the Core-Dijkstra variant. The three additional steps (contracting the core graph, reordering the core vertices, updating the upward and downward graph) are executed sequentially. During the query stage, we parallelize only the linear sweeps on the individual cells as described before.

Generally, it is also possible to parallelize a single PHAST query by processing vertices of the same level by distinct threads [13]. When all threads terminate, one can proceed to the next level. This works, since there are no dependencies between vertices on the same level in a standard PHAST query. Unfortunately, in our implementation, there can be such dependencies. Consider the following example. Assume that the CH preprocessing deletes an original edge (u, v) from the hierarchy, because it is superfluous (i.e., it is not needed to preserve shortest-path distances). Then, u and v may not be connected *directly* any more. Hence, the CH preprocessing may assign both vertices the same level.

However, recall that we ensure that if two vertices are connected in the original graph, there must also be an edge in $G^{\downarrow}$ that connects these vertices. Hence, in the example above, we would insert an artificial downward edge between u and v after the contraction process. This leads to the situation that there is an edge between two vertices on the same level. By storing such artificial downward edges at the endpoint with the higher ID, we can ensure that when a sequential linear sweep checks whether such an edge is an isochrone edge, both endpoints have final distance labels. However, considering a parallel linear sweep, it is difficult to resolve these dependencies. That is why we do not parallelize to linear sweep through the core vertices.

One could avoid the problem described above by adapting the CH preprocessing such that it never deletes original edges. However, even then the speedup due to a parallel Core-PHAST would probably be rather low. For a good parallel performance, we need enough vertices (i.e., enough parallel work) on the levels. This is the case in a standard PHAST query. For example, Delling et al. [13] observed that in a contraction hierarchy of Western Europe with 18 million vertices and 138 levels, half of the vertices are on the lowest level and that the lowest 20 levels contain more than 99 % of the vertices. However, the core graph has much fewer vertices, but a similar amount of levels. This makes it difficult to parallelize the linear sweep over the core vertices.

5.2.3 Distance Oracle

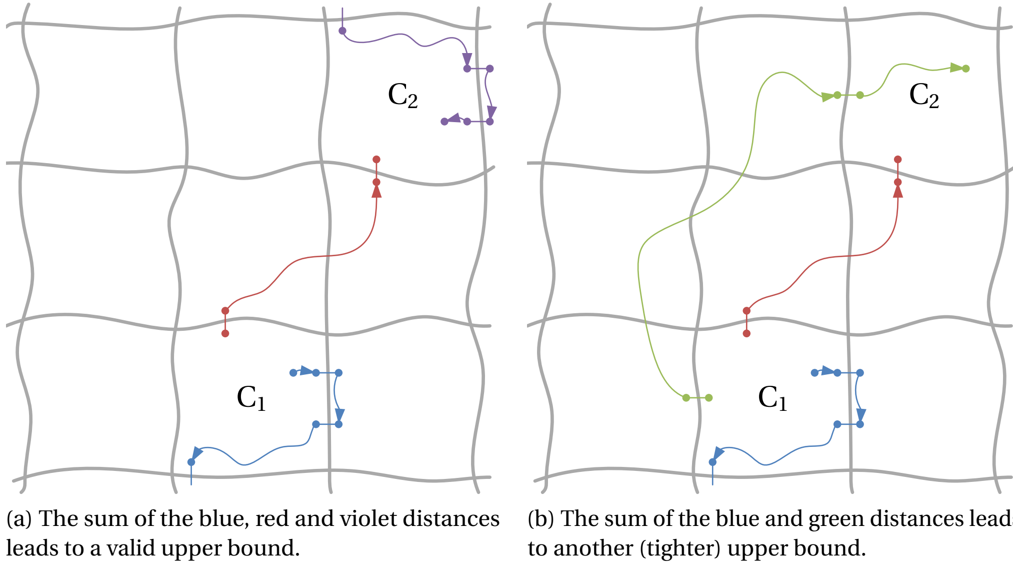
Since Dijkstra's algorithm does not scale well on large graphs and is difficult to parallelize, we replaced the Dijkstra search on the core graph by a linear sweep through the core vertices in the previous section. While the Core-PHAST variant of ES+PHAST tends to be faster than the Core-Dijkstra variant if the time limit is large, it also suffers from two drawbacks. First, as described above, the linear sweep over the core vertices is difficult to parallelize. Especially when processing the active cells in parallel, it is a performance bottleneck. Second, since PHAST processes vertices in decreasing order of level and not in increasing order of distance, it is not clear how to have an early termination criterion. Instead, we always compute correct distance labels for all core vertices, regardless of the time limit, which causes unnecessary work in the case of small time limits. This section describes a third variant of ES+PHAST that reduces the sequential work during queries and avoids to compute unnecessary distance labels. We call it the *distance oracle variant* of ES+PHAST.

Up to know, we have always computed correct distance labels for some or all core vertices as a first sequential step. Afterwards, we performed linear sweeps to propagate distance values from the core vertices to the internal vertices of the active cells. If multiple cores are available, we can process the active cells in parallel. In contrast, the distance oracle variant of ES+PHAST does not need a dedicated phase to compute correct distance labels for core vertices. Instead, it processes each active cell individually using an RPHAST query. During preprocessing, we extract, for each cell, the relevant subgraph (cf. section 2.4.3) of the downward graph $G^{\downarrow}$, just like in a standard RPHAST selection phase. These subgraphs are stored explicitly. During queries, we first perform a forward CH search and then run a linear sweep through the precomputed subgraph of each active cell.

In order to determine active cells, we use a distance oracle. That is, for each pair of cells C_1, C_2, we maintain a lower bound $\ell(C_1, C_2)$ and an upper bound $u(C_1, C_2)$ on the distance between a vertex in C_1 and another vertex in C_2. More precisely, for all pairs of vertices $u \in C_1, v \in C_2$ it holds that $\ell(C_1, C_2) \leq \text{dist}(u, v) \leq u(C_1, C_2)$. We determine active cells as follows. Let C_s be the cell that contains the source vertex and consider an arbitrary cell $C \neq C_s$. If the time limit x is smaller than the lower bound $\ell(C_s, C)$, all vertices in C are not reachable within x and thus there can be no isochrone edges (or pairs) in C. Conversely, if $x \geq u(C_s, C)$, all vertices in C are reachable within x and, again, there can be no isochrone edges in C. However, if $\ell(C_s, C) \leq x < u(C_s, C)$, there may be isochrone edges in C, which is thus marked as an active cell. In the next section, we describe how to compute the lower and upper bounds.

Computing the Distance Oracle

There is an obvious trade-off between preprocessing time and accuracy. On the one hand, we can obtain exact lower and upper bounds by computing all-pairs shortest paths on the whole road network. However, the PHAST algorithm, which is the fastest all-pairs shortest path technique on road networks, needs a few days to do so on a standard multi-core workstation.



(a) The sum of the blue, red and violet distances leads to a valid upper bound.

(b) The sum of the blue and green distances leads to another (tighter) upper bound.

Figure 5.2 – Illustration of the different distances that are involved in the computation of the distance oracle. The violet path denotes the forward diameter of C_2, the blue path denotes the backward diameter of C_1. The lower bound from C_1 to C_2 is colored red. Finally, the longest shortest path from a boundary vertex in C_1 to any vertex in C_2 is colored green.

Even its GPU implementation needs about half a day on a high-end graphics card [13].

We now propose two approaches that are much faster, but do not lead to exact bounds. In the following, we need the concept of *cell diameters*. We call the length of the longest shortest path from a boundary vertex in C to any other vertex in C the *forward diameter* of C. More precisely, $\text{diam}_f(C) = \max_{u,v \in C}\{\text{dist}(u,v)\}$, where u is a boundary vertex. Note that the shortest $u-v$ path is not restricted to C and may pass the cell border. Similarly, we call the length of the longest shortest path from any vertex in C to a boundary vertex in C the *backward diameter* of C. See Fig. 5.2 for an illustration. The violet path denotes the forward diameter of C_2, and the blue path denotes the backward diameter of C_1.

In order to compute (exact) lower bounds, we perform multiple-source PHAST queries on the core graph. To obtain lower bounds from a cell C to all other cells, we initialize the queue for the forward CH search with all boundary vertices of C and set their distance labels to zero. During the linear sweep through the core vertices, we remember for each cell the smallest distance label among its boundary vertices. In other words, after processing a vertex v, we check whether $d[v] < \ell(C, c(v))$. If it is, we set $\ell(C, c(v)) = d[v]$. After the linear sweep terminates, we have lower bounds from C to all other cells. In Fig. 5.2, the red path denotes the lower bound from C_1 to C_2. Now, we can obtain a valid (but not necessarily tight) upper bound by forming the sum $u(C_1, C_2) = \text{diam}_b(C_1) + \ell(C_1, C_2) + \text{diam}_f(C_2)$. Fig. 5.2a illustrates this approach.

If we are willing to spend slightly more time, we can obtain tighter upper bounds. We still perform one multiple-source PHAST query per cell, but run it on the whole graph instead of on the core graph. The initialization remains the same. During the linear sweep, we remember for each cell the smallest distance label among its vertices as before. This gives the lower bounds. Additionally, we remember for each cell the largest distance label among its vertices. This gives the length $\tilde{u}(C_1, C_2)$ of the longest shortest path from a boundary vertex in C_1 to any vertex in C_2. In Fig. 5.2, this length is colored green. Now, we obtain an upper bound from C_1 to C_2 by forming the sum $u(C_1, C_2) = \text{diam}_b(C_1) + \tilde{u}(C_1, C_2)$. Obviously, this upper bound cannot be worse than the one from the previous paragraph and often, it will be significantly tighter. In the following, we use the tighter upper bounds. We move on to describe some details of the distance oracle variant of ES+PHAST.

Preprocessing Stage

The preprocessing stage closely follows that of the Core-PHAST variant. That is, it first reorders the vertices according to the edge separator (boundary vertices are pushed to the front etc.) and then it computes some topological data, such as the range of vertices for each cell. Afterwards, it contracts the cells and reorders their internal vertices by level. However, instead of computing eccentricities, it then computes the backward diameters of the cells. Conceptually, this is similar to what we do when computing eccentricities. We process one cell after the other. Now, consider an arbitrary cell C and a boundary vertex v in it. First, we run Dijkstra's algorithm from v on the core graph until all other boundary vertices of C have been scanned. In contrast to eccentricities, we do not restrict the search to C. Afterwards, we perform a linear sweep through the internal vertices of C to propagate distance values from core vertices to internal vertices. Note that the Dijkstra search relaxes backward edges and that the linear sweep is done on the upward graph $G^{\uparrow}$, since we are looking for distances to v (and not from v). We remember the distance dist($f(v), v$) from the farthest vertex $f(v)$ found during the search and the sweep. The backward diameter of C then is the maximum dist($f(v), v$) over all boundary vertices $v \in C$.

After computing the backward diameters, the preprocessing stage proceeds with the contraction of the core graph and then reorders the core vertices by level. Afterwards, it updates the upward graph $G^{\uparrow}$ and the downward graph $G^{\downarrow}$ such that they also contain the upward and downward edges from the contracted core graph (besides the upward and downward edges from the contracted cell graphs, which we already needed to compute the backward diameters). At this point, the preprocessing of the Core-PHAST variant of ES+PHAST was finished. For the distance oracle variant, it remains to extract the relevant subgraph of the downward graph $G^{\downarrow}$ for each cell. To do so, we simply perform a standard RPHAST selection phase. It expects a set of targets that should be included in the extracted subgraph. In our case, the (boundary and internal) vertices of a cell are the relevant targets. The extracted subgraphs are stored explicitly for the query stage, since it would take too much time to perform the selection during queries.

Query Stage

The query stage of the distance oracle variant consists of the three main phases (forward CH search, determine active cells, process active cells) that were outlined at the beginning of this chapter. Additionally, it needs a fourth phase, which determines isochrone edges (or pairs) that could not be determined on-the-fly during one of the other phases. We come back to the fourth phase in the next section. The forward CH search works as usual and was described several times before. In order to determine active cells, we check for each cell C whether the time limit x lies between the lower bound $\ell(C_s, C)$ and the upper bound $u(C_s, C)$. If $\ell(C_s, C) \leq x < u(C_s, C)$, we mark C as active.

In the third phase, we perform a linear sweep on each subgraph that belongs to an active cell (or the cell which contains the source vertex). As usual, to process a vertex v, we examine each incoming downward edge (u, v). If $d[u] + \ell(u, v) < d[v]$, we set $d[v] = d[u] + \ell(u, v)$. This computes correct distance labels for all vertices that are contained in an active cell. Note that we do not need to reinitialize the distance labels before processing the next subgraph, since the source vertex remains the same. Of course, before computing a whole new query with a new source, distances labels need to be reinitialized, for example by using time-stamps.

Determining the Output

Once again, we need to distinguish isochrone edges between two core vertices and isochrone edges incident on at least one internal vertex. The latter are determined, as in the previous two variants, during the linear sweeps through the active cells. Whenever we there process a non-core vertex, we loop through the incoming downward edges a second time and check which of them is an isochrone edge. Note that we have to take the same precautions that we already described for the previous variants, for example, we need to insert some artificial downward edges into $G^{\downarrow}$ in order to get correct results.

Determining isochrone edges (or pairs) between two core vertices is more difficult. Of course, the linear sweeps could check for all downward edges whether they are isochrone edges (and not only at non-core vertices). This might give some isochrone edges of the first type, but also many duplicates, since the subgraphs are not distinct. While we could eliminate duplicates by using some sorting algorithm, the approach fails because isochrone edges could be missed.

Consider two cells that border on each other and assume that there are some boundary edges crossing this border. Suppose that in one cell, all vertices are reachable within the time limit, and in the other cell, all vertices are not reachable. Depending on the bounds, we may mark neither of the two cells as active. This may lead to the situation where some boundary edges, that truly are isochrone edges, are not contained in any of the processed subgraphs. In order to find such isochrone edges, we look at each edge between two core vertices, after the last linear sweep has terminated, and check which of them is an isochrone

edge (or pair). These edges reside at the beginning of the edge array of the downward graph, since core vertices were assigned the lowest IDs during preprocessing. Hence, we look only at a small fraction of the array.

Parallelization

The preprocessing stage of the distance oracle variant of ES+PHAST can be parallelized similar to that of the Core-PHAST variant. Additionally, we compute the backward diameters in parallel by assigning cells to distinct cores. This is similar to how we parallelize the computation of eccentricities. The extraction of the subgraphs can be done in parallel in the same way (assign cells to distinct cores).

From the query stage, we parallelize the third and fourth phase. The linear sweeps on the subgraphs are independent of each other, so they can be performed in parallel. The same is true for the checks which boundary edge is an isochrone edge (or pair). Determining active cells takes less than a few microseconds, so we leave it sequential. Note that since the subgraphs are not distinct, each core has to use its own copy of the distance labels in order to avoid concurrent accesses. As a consequence, each core also needs to perform the forward CH search to initialize its copy with the correct values.

Compressing the Subgraphs

As mentioned above, the vertex sets of the extracted subgraphs are not distinct. For example, the highest-ranked vertex on top of the hierarchy is guaranteed to be contained in each subgraph. Generally, the more cells we use, the more vertices are contained in multiple subgraphs. This does not only waste a considerable amount of space, but also slows down queries, since high-rank vertices are processed again and again during the same query. In order to save space and accelerate queries, we "compress" the subgraphs.

The idea is as follows. We store the topmost k vertices of the hierarchy and their incoming downward edges in a separate subgraph, and remove them from all other subgraphs. This works very well, for example, when using 1024 cells and $k = 8192$ on Western Europe, the space required by the subgraphs decreases from 2164 MiB to 499 MiB. During queries, each core first needs to perform a linear sweep through this special subgraph, before it can start to process the subgraphs that belong to active cells.

5.2.4 Drawbacks of the ES+PHAST Algorithms

In the previous three sections, we described three variants of the ES+PHAST algorithm. Unfortunately, each of them has significant drawbacks. The Core-Dijkstra variant does not scale well as the number of cells increases and also not with the time limits used. The Core-PHAST variant has no early termination criterion and thus always computes correct distance

labels for all core vertices, regardless of the time limit. In both variants, the searches on the core graph are difficult to parallelize and, as a consequence, are a performance bottleneck on multi-core machines. Another drawback of the Core-PHAST variant is that the hierarchy we built is not as good as a “standard” contraction hierarchy. Since we block the contraction of boundary vertices until all internal vertices are contracted, we force the CH preprocessing to select a suboptimal contraction order.

The distance oracle variant suffers from the latter as well. However, if we contracted the whole graph without blocking boundary vertices, then boundary edges would be spread over the whole edge array. As a consequence, the time required to determine which boundary edges are isochrone edges would increase significantly. This step introduces a slowdown even now, although we look only at a small fraction at the beginning of the edge array of the downward graph (where the boundary edges reside). When using a standard contraction hierarchy, we would need to loop through the whole edge array, which would be way too slow. In order to remedy the drawbacks of the distance oracle variant, we have to eliminate boundary edges. We achieve this by switching from edge to vertex separators, as will be described in the next section.

5.3 Vertex Separators

A *vertex separator* is a subset $S \subset V$ of vertices such that the removal of S decomposes the graph $G = (V, E)$ into several disconnected cells. Usually, the subset should be small and the cells should be balanced. In contrast to edge separators, the cell boundaries here pass through the separator vertices, each of which may be contained in any number of cells. On the other hand, each edge can be assigned to a single cell and thus there are no boundary edges. Before describing the GS+PHAST algorithm when using vertex separators, the next two sections discuss how to compute vertex separators and how to obtain a cell topology from them.

5.3.1 Computing Vertex Separators

A common approach to compute a vertex separator is to partition the graph G in a first step and then obtain the separator from the partition in a postprocessing step. For simplicity, assume for now that the partition contains only two cells C_1, C_2. Clearly, the boundary vertices of C_1 form a valid vertex separator, and so do the boundary vertices of C_2. However, in order to obtain better (smaller) separators, one may also take the boundary edges into consideration. The approach described in [58, 54] computes the smallest subset of boundary vertices that forms a valid vertex separator. To do so, it builds a bipartite graph H that contains all boundary vertices and all boundary edges. Afterwards, it computes a *minimum vertex cover* $S \subset V$ in H. By definition of a vertex cover, each edge in H is incident on at least one vertex from S. Hence, the removal of S eliminates all boundary edges and thus

decomposes the graph G into several cells. In other words, the minimum vertex cover is a valid vertex separator. Note that, although the minimum vertex cover problem is **NP**-complete in general [11], it is efficiently solvable in bipartite graphs, since it can be reduced to a max-flow min-cut computation in this case [58, 54].

We come back to partitions that contain more than two cells. Adapting the approaches from the previous paragraph to compute k-way vertex separators is straightforward. One can simply pick an approach and apply it between all pairs of cells that shared a non-empty boundary. The union of the pairwise separators then forms the k-way separator. In our implementation, we use a helper program from the KaHIP framework [57] to compute vertex separators from partitions. It obtains separators by computing minimum vertex covers in bipartite graphs, as described in this section.

5.3.2 From Vertex Separators to Topologies

A vertex separator is just a set of vertices. However, for the GS+PHAST algorithm we need to know the topology, that is, we have to fix boundaries and determine for each cell its boundary vertices and for each separator vertex its incident cells. To do so, we combine information from the vertex separator S and the underlying partition that was used to compute S. As always, we denote by $c(v)$ the cell that contains v according to the (underlying) partition. The internal vertices of a cell C are all non-separator vertices v such that $c(v) = C$. We move on to determine boundary vertices.

We maintain one list of boundary vertices for each cell and one list of incident cells for each separator vertex. For each vertex $u \in S$, we loop through its incident edges (u, v). If v is a non-separator vertex, we insert u into the list of boundary vertices of $c(v)$ and add $c(v)$ to u's list of incident cells. In Fig. 5.3, such edges are colored red. Now, assume that v is also a separator vertex. By construction of the separator, the edge (u, v) has to be a boundary edge in the partition and thus goes directly along the boundary between $c(u)$ and $c(v)$. Such edges are colored green in Fig. 5.3. We can put (u, v) in either $c(u)$ or $c(v)$, however, we need to be consistent during all stages of the algorithm. We decide to put edges (u, v) between two separator vertices in $c(\min\{u, v\})$.

5.3.3 The VS+PHAST Algorithm

This section finishes the chapter with a description of the GS+PHAST algorithm when using vertex separators. We call this variant *VS+PHAST algorithm*. It closely follows the distance oracle variant of ES+PHAST. The main difference is that VS+PHAST obtains the cell topology from a vertex separator instead of from an edge separator. Since most other algorithmic aspects are quite similar, we will not go into details again. In the following, we briefly point out some differences during the different stages.

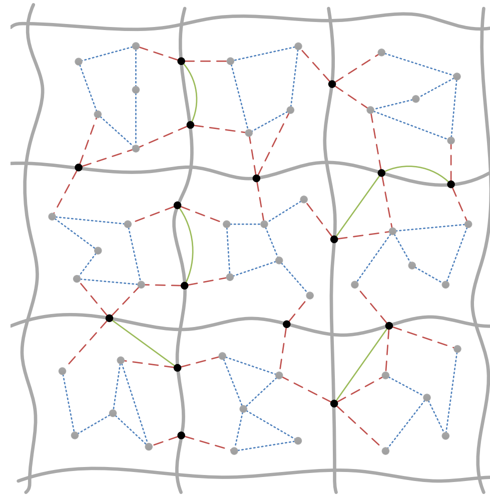


Figure 5.3 – An example of a topology obtained from a vertex separator. Black circles denote separator vertices, gray circles denote internal vertices. Edges between two internal (separator) vertices are colored blue (green). Red edges denote those between a separator and an internal vertex.

Preprocessing Stage. One advantage of VS+PHAST is that we can use a standard contraction hierarchy without delaying the contraction of core vertices. This allows better contraction orders and thus leads to lower preprocessing and query times. Note, however, that we still temporarily contract and reorder each cell in order to accelerate the computation of the backward diameters as before. If we computed the backward diameters by running Dijkstra's algorithm in the original graph, we would spend more time than we would save by avoiding the contraction of the cells.

Query Stage. The query stages are almost identical. If the source s is a non-core vertex, we look up the cell C_s that contains s and check the lower and upper bounds from C_s to all other cells. Note that if s is a core vertex, there are multiple cells that contain s. However, queries remain correct if we just take any of the cells that contain the source as C_s.

Determining the Output. Another advantage of VS+PHAST due to the absence of boundary edges is that queries do not need an additional fourth phase. All isochrone edges (or pairs) can be determined during the linear sweeps on the subgraphs that belong to active cells. However, since the vertex sets of the subgraphs are not distinct, we have to pay attention that we produce no duplicates. Recall that we output only edges where a special bit is set that marks them as original edges. We use the same mechanism to avoid duplicates at no extra cost. After extracting the relevant subgraph from the downward graph for a cell C, we simply loop through its edges and unset the aforementioned bit for all of them that are not contained in C.

6 Isochrones for Electric Vehicles

In this chapter, we adapt RangeDijkstra and the multilevel Dijkstra techniques such that they are capable of computing isochrones for electric vehicles. In the EV scenario, we want to obtain the region that is reachable with a certain battery charge level. Recall that we cannot simply use energy consumption instead of travel times as metric, since energy-optimal paths differ considerably from quickest paths [9]. Instead, we use two cost functions in the EV scenario: a *routing cost function*, namely travel times, and a separate *consumption cost function*, namely the energy consumption of the electric vehicle.

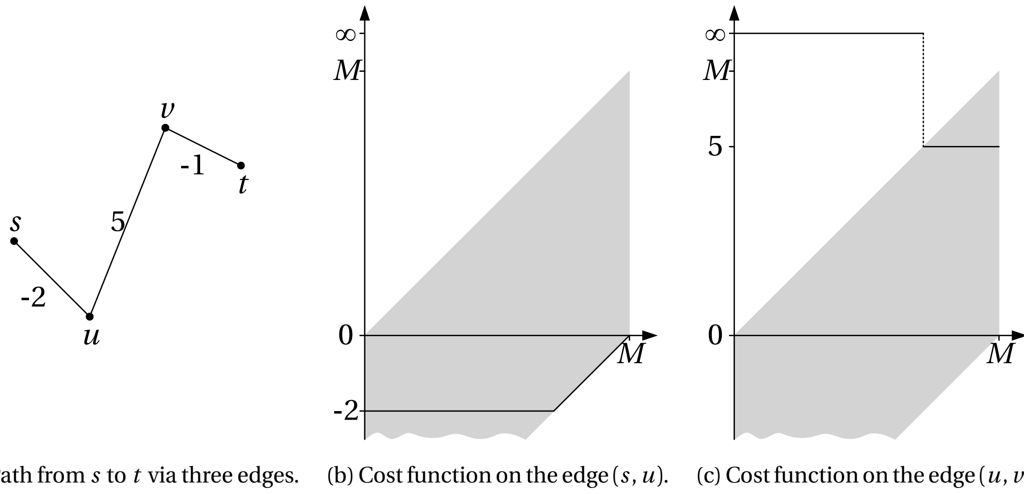
The first section of this chapter describes how the energy required to get across a path can be modeled as a cost function of bounded descriptive complexity, which was first observed by Eisner et al. [33]. Afterwards, we introduce some basic operations on such cost functions that we will use in our algorithms. In the last section, we finally adapt RangeDijkstra and the multilevel Dijkstra techniques for the EV scenario.

6.1 Modeling Energy Consumption

As before, each edge (u,v) is assigned a length $\ell(u,v)$ that represents the time it takes to travel along the edge. We use ℓ as routing metric. Additionally, we assign each edge (u,v) a consumption value $c(u,v)$. In contrast to travel times or travel distances, c is no non-negative metric, since electric vehicles are able to recuperate energy during deceleration phases or when going downhill. Hence, $c(u,v)$ takes on negative function values on some downhill edges. However, due to physical reasons there are no negative cycles.

In order for a vertex v to be reachable from s with an initial charge level b_s, the shortest $s-v$ path $P_{sv} = (s = v_0, \dots, v_k = v)$ has to obey the *battery constraints* of the electric vehicle. First, we must never run out of energy when driving along P_{sv}. More precisely, $b_{v_i} \geq 0$ for $i = 1, \dots, k$. Second, overcharging must never take place. That is, the battery charge b_{v_i} must not exceed the maximum charge level M of the electric vehicle for any $i = 1, \dots, k$. If there is an initial charge level such that a path obeys the battery constraints, we say it is *feasible*.

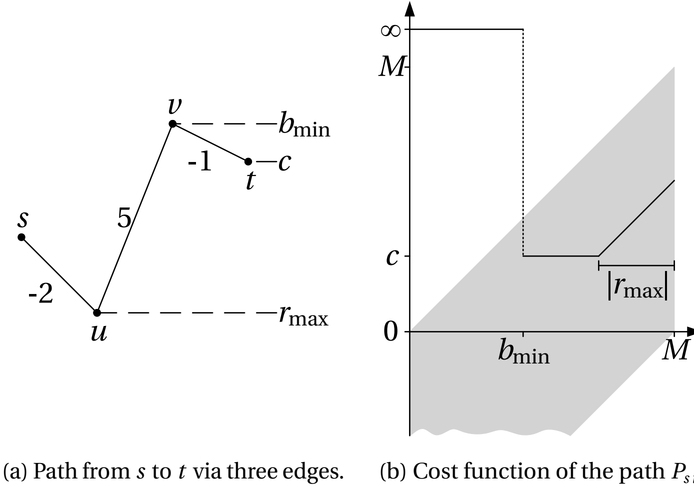
Due to the battery constraints, the actual amount of energy that is consumed or recuperated when driving along a path $P = (v = v_0, \dots, v_k)$ or even along a single edge (v, w) is not constant, but a function of the battery charge level b_v at v. These *cost functions* f_e on the edges e are of the form $f_e : [0, M] \to \mathbb{R} \cup \infty$. When going along an edge (v, w) with $c(v, w) < 0$, we are able to fully recuperate the energy only if $b_v \leq M + c(v, w)$. As soon as b_v exceeds $M + c(v, w)$, the amount of energy that is recuperated decreases linearly. Finally, if $b_u = M$, we are not able to recuperate any energy at all. See Fig. 6.1b for an example. When considering to drive along an edge $e = (v, w)$ with $c(v, w) \geq 0$, we are not allowed to take this road segment if $b_v < c(v, w)$. Hence, we set $f_e(b_v) = \infty$ for $b_v < c(v, w)$ and otherwise we set $f_e(b_v) = c(v, w)$. See Fig. 6.1c for an example.



(a) Path from s to t via three edges. (b) Cost function on the edge (s, u). (c) Cost function on the edge (u, v).

Figure 6.1 – Cost functions on edges with negative and positive consumption costs. Finite function values are only taken on in the shaded area.

As we have seen, cost functions on edges always consist of (at most) two pieces. Cost functions for whole paths $P = (v_0, \dots, v_k)$, i.e., sequences of edges, have almost the same descriptive complexity and consist of (at most) three pieces. See Fig. 6.2 for an example. Let $b_{\min}$ be the minimal battery charge level required at v_0 to get across P, that is, no cost function $f_e(b_u)$ of an edge $e = (u, v)$ on the path P takes on infinity for the battery charge level b_u at u. Then, by definition of $b_{\min}$, the cost function f_P takes on infinity in the interval $[0, b_{\min} - 1]$. Now, let $r_{\max}$ be the (possibly negative) amount of energy that is consumed at the lowest vertex in the consumption profile (see Fig. 6.2a). In the interval $[b_{\min}, M - |r_{\max}|]$, energy recuperation is not limited and thus f_P takes on $c = \sum_{i=0}^{k-1} c(v_i, v_{i+1})$ in the whole interval. For initial charge levels $b_{v_0} > M - |r_{\max}|$, recuperation is limited, which means that there is at least one vertex on P where the battery is fully loaded and thus the final charge level b_{v_k} at v_k is the same for all such b_{v_0}. Hence, if we start at v_0 with a higher initial charge level b_{v_0}, the consumed energy $b_{v_0} - b_{v_k}$ increases by the same amount. So the third piece of f_P is a line of slope one in the interval $[M - |r_{\max}|, M]$.



(a) Path from s to t via three edges. (b) Cost function of the path P_{st}.

Figure 6.2 – Cost function of a path consisting of edges with negative and positive consumption costs. The minimal battery charge required to get across the path is $b_{\min}(P_{st}) = 3$ and the maximal amount of energy that is recuperated while traversing the path is $r_{\max}(P_{st}) = -2$. The total consumption cost is $c(P_{st}) = 2$.

Each cost function (on edges or whole paths) can be fully specified by a triple $(b_{\min}, r_{\max}, c)$. This allows us to store any cost function as three 32-bit integers. Note that in the case of cost functions on single edges (u, v) with $c(u, v) < 0$, $b_{\min}$ is always zero. Conversely, for cost functions on single edges (u, v) with $c(u, v) \geq 0$, $r_{\max}$ is always zero. More precisely, the cost function that is induced by a triple $(b_{\min}, r_{\max}, c)$ is defined as

$$f(b) = \begin{cases} \infty, & b \in [0, b_{\min} - 1] \\ c, & b \in [b_{\min}, M - |r_{\max}|] \\ c + (b - (M - |r_{\max}|)), & b \in [M - |r_{\max}| + 1, M] \end{cases} .$$

6.2 Basic Operations on Edge Cost Functions

The previous section described how we model energy consumption as cost functions and how we represent them in memory. Before we move on to adapt our algorithms for the EV scenario, this section introduces three basic operations on edge cost functions that we will use in our algorithms. These operations include the evaluation of edge cost functions, linking two edge cost functions and checking for dominance between two edge cost functions.

6.2.1 Evaluation

The evaluation of an edge cost function $f_P = (b_{\min}, r_{\max}, c)$ at a battery charge level b is straightforward. See Alg. 6.1. If $b < b_{\min}$, the battery charge level does not suffice to get

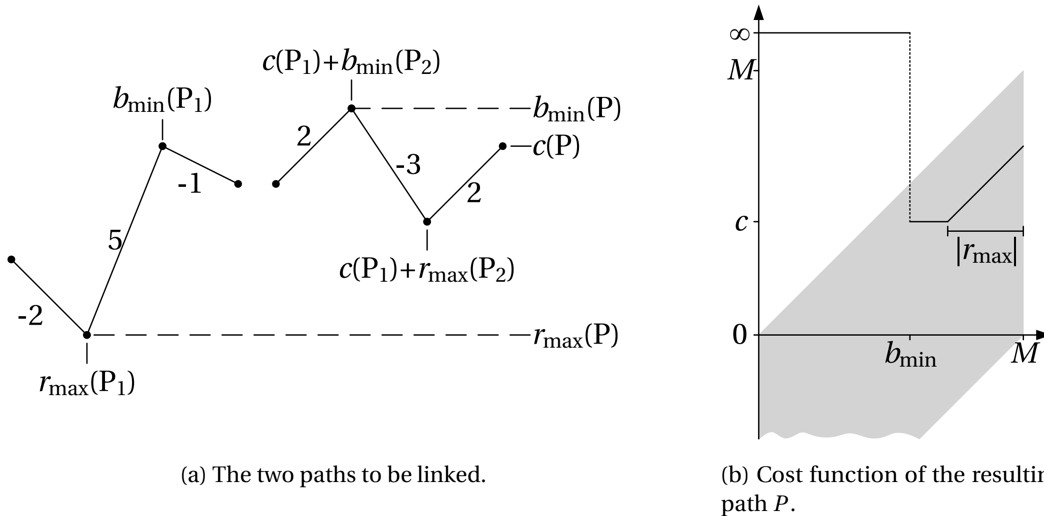
across P, that is, the electric vehicle runs out of energy. Hence, we return a cost of infinity. Next, if $b \leq M - |r_{\max}|$, energy recuperation is not limited and thus we return c. Otherwise, b is in the interval $[M - |r_{\max}| + 1, M]$ and the energy required to traverse P is computed by $c + (b - (M - |r_{\max}|))$, as described in the previous section.

Algorithm 6.1: EvalEdgeCostFunction(f, b)

```
1 if b < f.b_min then                              // EV runs out of energy
2   return ∞
3 if b ≤ M + f.r_max then                          // overcharging does not take place
4   return f.c
5 return f.c + (b − (M + f.r_max))
```

6.2.2 Linking

During the customization of the multilevel Dijkstra techniques, we need to compute shortcut edges between the boundary vertices within each cell. A shortcut edge on level ℓ is the concatenation of shortcuts (and boundary edges) on level $\ell-1$. To obtain the cost function f_P for a path P that is the concatenation of two paths P_1 and P_2, we need to *link* their cost functions f_{P_1} and f_{P_2}. What we need to do to link two edge cost functions becomes clear when looking at their consumption profiles. See Fig. 6.3a for an example. We obtain the consumption profile of P by joining the profiles of P_1 and P_2 such that the P_1's last vertex touches P_2's first vertex. Obviously, we have to set $b_{\min}(P)$ to $\max\{b_{\min}(P_1), c(P_1) + b_{\min}(P_2)\}$ and $r_{\max}(P)$ to $\min\{r_{\max}(P_1), c(P_1) + r_{\max}(P_2)\}$. The total cost $c(P)$ when energy recuperation is not limited is simply the sum $c(P_1) + c(P_2)$.



(a) The two paths to be linked.

(b) Cost function of the resulting path P.

Figure 6.3 – Linking two cost functions. P_1 and P_2 denote the two paths taken as input. The resulting path is denoted by P.

Note that we have to check explicitly whether the resulting path is feasible, that is, if there is an initial charge level such that the path obeys the battery constraints. This is not the case if the distance between $r_{\max}(P_1)$ and $c(P_1) + b_{\min}(P_2)$ is greater than the maximum charge level M, since even a fully loaded battery then does not suffice to get across the resulting path. If we determine that the resulting path is not feasible, we manually set its minimum charge level and its total cost to infinity (see Algorithm 6.2).

Algorithm 6.2: LinkEdgeCostFunctions(f_1, f_2)

```
1 if f1.c + f2.bmin − f1.rmax > M then        // resulting edge cost function is not feasible
2     f.bmin ← ∞
3     f.rmax ← 0
4     f.c ← ∞
5 else                                          // resulting edge cost function is feasible
6     f.bmin ← max{f1.bmin, f1.c + f2.bmin}
7     f.rmax ← min{f1.rmax, f1.c + f2.rmax}
8     f.c ← f1.c + f2.c
9 return f
```

6.2.3 Dominance Check

Another operation that we will use when computing shortcut edges is to check for dominance between two cost functions. We say that f_1 *dominates* f_2 if there is no b such that $f_1(b) > f_2(b)$. Due to the special shape of the cost functions, this check can be done efficiently in constant time. Recall that each cost function f in general consists of three pieces. In the interval $[0, f.b_{\min} - 1]$, the cost function takes on infinity. The middle piece is a horizontal line that represents the cost when energy recuperation is not limited and finally a line of slope one forms the right piece of the cost function. Note that one or two of the pieces may vanish. Now, f_1 dominates f_2 if the following three conditions hold:

1. The minimal charge level of f_1 is no greater than the one of f_2, that is, $f_1.b_{\min} \leq f_2.b_{\min}$.
2. The right piece of f_2 (with slope one) does not lie below the right piece of f_1.
3. The middle piece of f_2 (with slope zero) does not lie below the middle piece of f_1.

It is clear that the first condition can be checked using a single comparison. The second condition can be seen as a geometric problem (see Fig. 6.4). Let f_1 denote the lower cost function and f_2 the upper cost function. The right piece of f_2 does not lie below the right piece of f_1 if the point r lies to the left or on the line through the points p and q. The points p and r are defined as follows.

$$p = \begin{pmatrix} M + f_1.r_{\max} \\ f_1.c \end{pmatrix} \qquad r = \begin{pmatrix} M + f_2.r_{\max} \\ f_2.c \end{pmatrix}$$

Let v be the vector from p to r, that is,

$$v = r - p = \begin{pmatrix} f_2.r_{\max} - f_1.r_{\max} \\ f_2.c - f_1.c \end{pmatrix}.$$

Since the line through p and q has slope one, its normal vector is

$$n_{pq} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}.$$

By definition of the dot product, if $v^{\mathsf{T}} \cdot n_{pq} \geq 0$, then $0° \leq \phi \leq 90°$ and thus r lies to the left or on the line through p and q. Hence, if $f_1.r_{\max} - f_2.r_{\max} - f_1.c + f_2.c \geq 0$, then the right piece f_2 does not lie below the right piece of f_1. So the second condition can be checked using one addition, two subtractions and one comparison. It is easy to see that the formula remains correct when one (or even both) of the right pieces vanishes.

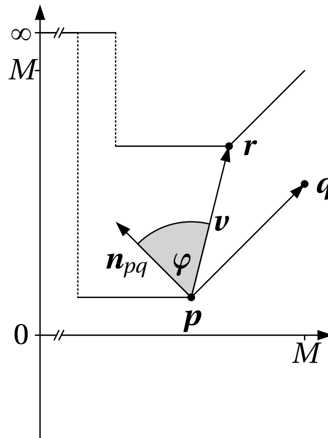


Figure 6.4 – Dominance check seen as a geometric problem.

In order to check the third condition, it suffices to ensure that $f_1(f_2.b_{\min}) \leq f_2(f_2.b_{\min})$ if we already know that the other two conditions hold. The cost functions are evaluated as described before. Algorithm 6.3 summarizes the steps to check whether f_1 dominates f_2.

Algorithm 6.3: Dominates(f_1, f_2)

1 **if** $f_1.b_{\min} > f_2.b_{\min}$ **then**
2   **return false**
3 **if** $f_1.r_{\max} - f_2.r_{\max} - f_1.c + f_2.c < 0$ **then**
4   **return false**
5 **if** EvalEdgeCostFunction(f_1, $f_2.b_{\min}$) > EvalEdgeCostFunction(f_2, $f_2.b_{\min}$) **then**
6   **return false**
7 **return true**

6.3 Extending Algorithms

We now adapt RangeDijkstra and the multilevel Dijkstra techniques for the EV scenario. We chose the multilevel Dijkstra techniques since they provide a reasonable trade-off between preprocessing time, customization time and query time. However, adapting the GS+PHAST algorithms is also possible and remains for future work. This section starts by describing the changes made to RangeDijkstra and then moves on to isoCRP and isoGRASP.

6.3.1 RangeDijkstra

The standard RangeDijkstra maintains for each vertex v a distance label $dlbl[v]$, which stores the length of the shortest path from the source s to v found so far. In the EV scenario, we keep the distance labels and additionally maintain for each vertex v a consumption label $clbl[v]$, which stores the (possibly negative) amount of energy that is consumed when driving along the shortest $s-v$ path found so far. If $clbl[v] < 0$, we recuperate more energy than we consume. As long as $clbl[v]$ is finite, the shortest $s-v$ path obeys the battery constraints. We also say that v is *ev-reachable* with the initial charge level b_s. If $clbl[v] = \infty$, the vertex v is not ev-reachable. Initially, we set $dlbl[s] = clbl[s] = 0$ and $dlbl[v] = clbl[v] = \infty$ for all $v \neq s$. We stress once again that we use two metrics in the EV scenario, namely travel times as routing metric and electric energy consumption as consumption metric. Hence, when we say shortest paths, we mean quickest paths, and not energy-optimal paths.

Just like the standard variant, the EV variant of RangeDijkstra maintains a priority queue of unscanned vertices with finite *dlbl* values, using their distance labels as keys. Vertices are scanned in increasing order of distance from s. To scan a vertex u, we relax all outgoing edges $e = (u, v)$. To do so, if $dlbl[u] + \ell(u, v) < dlbl[v]$, we set $dlbl[v] = dlbl[u] + \ell(u, v)$ and $clbl[v] = clbl[u] + f_e(b_u)$, where $b_u = b_s - clbl[u]$ is the battery charge level at u. If $dlbl[u] + \ell(u, v) = dlbl[v]$ and $clbl[u] + f_e(b_u) < clbl[v]$, we also update $clbl[v]$ accordingly.

Stopping Criterion

Recall that the standard RangeDijkstra stops when all elements in the priority queue have a distance label greater than the time limit. In other words, it stops when all vertices in the queue are not reachable (within the time limit). In the same way, we can safely stop RangeDijkstra in the EV scenario when all elements in the queue are not ev-reachable (with the initial charge level), that is, when all vertices in the queue have infinite *clbl* values.

One might want to implement the stopping criterion by only allowing ev-reachable vertices to enter the queue, and running the algorithm until the queue is empty. However, then we may not find the shortest path to some vertices and thus may erroneously consider some of these vertices as ev-reachable. Consider the following example. There are two vertices s and v that are connected by two distinct paths P_1 and P_2. Assume that P_1 is the quicker paths,

but does not obey the battery constraints. The other path P_2 is the slower path, but fulfills the battery constraints. If we did not allow ev-unreachable vertices to enter the queue, we would only insert the vertices on P_2 into the queue. Finally, we would erroneously consider v as ev-reachable, although the shortest $s-v$ path does not obey the battery constraints. Clearly, the algorithm would not be correct.

Hence, we need to insert each vertex that the search visits into the queue. To implement the stopping criterion, we use a counter that keeps track of the number of ev-reachable vertices in the queue. Whenever we insert an ev-reachable vertex into the queue, we increase the counter, and whenever we extract an ev-reachable vertex from the queue, we decrement the counter. Of course, we also may need to update the counter whenever we update the consumption label of a vertex that is contained in the queue. We stop the algorithm as soon as the counter becomes zero.

Determining the Output

Recall that the standard RangeDijkstra determines isochrone edges as follows. It extracts, after the search has stopped, each vertex v from the queue, loops through its incoming edges (u, v) and outputs each of them where u is reachable (within the time limit). This works correctly, since each unreachable vertex that is adjacent to at least one reachable vertex is still in the queue when the search stops. However, as we have seen in the previous section, in the EV scenario we may already extract and process ev-unreachable vertices during the search, in order to find the shortest path to all vertices. Hence, isochrone edges that are incident on such vertices cannot be determined during the postprocessing step after the search has stopped.

In the EV scenario, we keep the postprocessing step as in the standard scenario. Additionally, we also look for isochrone edges during the search itself. However, we have to be careful, since when scanning a vertex v, not all of v's neighbors have final distance and consumption labels. In order to be correct, we must not check for edges between v and neighbors that have not been scanned yet whether they are isochrone edges. Since we use non-zero edge lengths in the EV scenario, each neighbor u with $dlbl[u] \leq dlbl[v]$ has final distance and consumption labels, since each $s-u$ path found later would lead to a higher distance label.

More precisely, we determine isochrone edges as follows. After scanning a vertex u, we loop through its outgoing edges again if u is ev-reachable. If u is ev-unreachable, we loop through its incoming edges again. For each outgoing edge (u, v), we check if $dlbl[v] < dlbl[u]$. If it is, we output (u, v) if v is ev-unreachable. For each incoming edge (v, u), we check if $dlbl[v] \leq dlbl[u]$, and if it is, we output (v, u) if v is ev-reachable. Note that we choose $<$ for outgoing edges and $\leq$ for incoming edges in order to avoid duplicates. Also note that this choice of the comparison operators is consistent with the postprocessing step. If we had chosen them the other way round, the choice would have not been consistent with the postprocessing step, and might have checked some edges twice.

We may determine isochrone pairs in the same way. However, note that in this case, the EV variant of RangeDijkstra needs the same adaptions that the standard RangeDijkstra needs in order to determine isochrone pairs. Basically, after scanning a vertex u, we need to loop through its incoming edges (v, u) and check for each v whether it has a finite distance label (and thus was inserted into the queue once already). If is has not, we need to insert v with a key of infinity into the queue. See section 3.3 for further details.

6.3.2 isoCRP

We now move on to the EV variant of isoCRP. Note that the extensions that we developed for the EV variant of RangeDijkstra carry over to isoCRP straightforwardly. Hence, we may not go into each detail in this section. We concentrate on the two-phase variant, however, the single-phase variant can be adapted in the same way.

Customization Stage

During the customization of standard isoCRP, we compute shortest-path distances between the boundary vertices within each cell and store them in matrices (one per cell). In the EV scenario, we need to compute two matrices per cell. Besides the matrix which stores the lengths of the shortcuts between boundary vertices, we need an additional matrix that stores for each shortcut its cost function. These cost functions can be computed during customization with little overhead.

Basically, we need to adapt the Dijkstra searches during customization in a similar way as we adapted RangeDijkstra. Obviously, the searches maintain for each vertex v besides a distance label $dlbl[v]$ an additional consumption label $clbl[v]$. However, this consumption label is not a single 32-bit integer as before, but a cost function. This is necessary since we need to know the amount of energy that is consumed when driving along a shortcut for each initial charge level. As mentioned before, cost functions can be represented in memory by three integers.

Assume that we scan a vertex u on level ℓ and are about to relax the outgoing (original or shortcut) edge $e = (u, v)$. If $dlbl[u] + \ell(u, v) < dlbl[v]$, we set $dlbl[v] = dlbl[u] + \ell(u, v)$. Additionally, we need to link the cost function that is stored at $clbl[u]$ with the cost function f_e. The resulting function is stored at $clbl[v]$. When the Dijkstra search stops, the distance label of each boundary vertex w represents the length of the shortest path to w (within the cell), and w's consumption label contains the cost function of the shortcut to w.

There is one thing we need to care about. It may happen that there are multiple paths of the same length to a single vertex. Assume that we have just found a new path to a vertex v via an edge (u, v) such that $dlbl[u] + \ell(u, v) = dlbl[v]$. Then, we check whether the cost function of the new path dominates the cost function stored at $clbl[v]$. If it does, we can

safely replace $clbl[v]$ with the new cost function. If it does not, we check whether the cost function at $clbl[v]$ dominates the new cost function. If so, we dismiss the new cost function.

Sometimes, however, neither cost function dominates the other and thus we have to keep both in order to ensure correct queries. Note that neither a consumption label nor a matrix entry has room for more than one cost function. Hence, the Dijkstra searches maintain an additional array that stores the cost functions for each vertex v that has an ambiguous consumption label. At v's consumption label we store the starting and ending positions in the additional array of v's cost functions. Consider scanning a vertex u with multiple cost functions and assume that we are about to relax an outgoing edge $e = (u, v)$ such that $dlbl[u] + \ell(u, v) < dlbl[v]$. Then, we have to link each of u's cost functions with f_e and check for dominance between all resulting functions, including the cost function or functions that is stored at $clbl[v]$. If we are lucky, there is one cost function that dominates all others and we can continue at v with a single cost function. Otherwise, we have to store all non-dominated cost functions at v as described before.

Due to dominance checks, the number of shortcuts with ambiguous cost functions can be reduced drastically. In our experiments on the road network of Western Europe, only 0.00009 % of the shortcuts have ambiguous non-dominated cost functions. The total number of non-dominated cost functions that are stored in the additional array is 6028.

Finally, eccentricities on the bottom level are computed as follows. During the Dijkstra search from a boundary vertex v (restricted to v's bottom-level cell), we update the eccentricity $\mathrm{ecc}_1(v)$ whenever we scan a vertex u by setting $\mathrm{ecc}_1(v) = \max\{\mathrm{ecc}_1(v), b_{\min}\}$, where $b_{\min}$ is the minimal charge level required at v to get to u. Eccentricities on higher levels are computed in a similar way. Assume that we are computing the eccentricity for a vertex v on level ℓ. Whenever scanning a vertex u on level $(\ell - 1)$, we link the cost function stored at $clbl[u]$ with the cost function f_e, where e is an edge of length $\mathrm{ecc}_{\ell-1}(u)$. Then, we set $\mathrm{ecc}_\ell(v) = \max\{\mathrm{ecc}_\ell(v), b_{\min}\}$, where $b_{\min}$ is the minimal charge level of the chained cost function. Note that on higher levels, we actually compute upper bounds on the eccentricities.

Query Stage

Queries in the EV scenario are basically standard isoCRP queries, except that they use the EV variant of RangeDijkstra during the upward and downward phase. Hence, the upward phase runs the EV variant of RangeDijkstra on the graph consisting of the union $c_1(s) \cup \cdots \cup c_{L-1}(s) \cup H_L$, that is, the union of the top-level overlay graph H_L and for each level ℓ the cell $c_\ell(s)$ that contains s. During the search, we set the bit $descend[\ell][C]$ for each level-ℓ cell C if we scan a boundary vertex v of C with $clbl[v] + \mathrm{ecc}_\ell(v) > b_s$, and we set the bit $reach[\ell][C]$ for each level-ℓ cell C if we scan at least one boundary vertex of C that is ev-reachable (see also section 4.1.3). The downward phase consists of L subphases. The i-th subphase runs the EV variant of RangeDijkstra on each level-$(L - i + 1)$ cells C for which both $descend[L - i + 1][C]$ and $reach[L - i + 1][C]$ are set.

We need to make one change to the EV variant of RangeDijkstra. Stopping as soon as the counter of the ev-reachable vertices in the queue becomes zero may lead to incorrect results. See Fig. 6.5 for an example. Assume that the green edges obey the battery constraints, but the red edges have too high consumption costs. When scanning s, we insert u with a key of one and v with a key of two into the queue. Next, we scan u. Since $\mathit{clbl}[u] + \mathrm{ecc}(u) \leq b_s$, we do not mark u's cell as active. Afterwards, the queue contains only the ev-unreachable vertex v and thus we stop the search. Now, we erroneously consider w as ev-reachable, since we have not found the shortest $s - w$ path, and miss the isochrone edge (u, w).

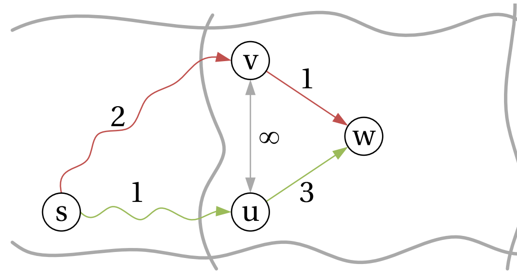


Figure 6.5 – Example where an isochrone edge is missed. Edge labels show the routing costs of the edges. Assume that the green edges obey the battery constraints, but the red edges have too high consumption costs. Then, we miss the isochrone edge (u, w).

In order to avoid the problem described above, we need to ensure that we find shortest paths to all vertices that we consider ev-reachable. To do so, we store for each boundary vertex v besides its normal (consumption) eccentricity additionally its *routing eccentricity* $\mathrm{recc}(v)$, which represents the length of the shortest path (restricted to the cell) from v to its farthest reachable vertex in the cell. Basically, the routing eccentricities in the EV scenario are the same as the normal (consumption) eccentricities in the standard scenario. During queries, we maintain (in addition to the counter) an upper bound $d_{\max}$ on the distance to the farthest ev-reachable vertex. Whenever scanning an ev-reachable vertex v, we set $d_{\max} = \max\{d_{\max}, \mathit{dlbl}[v] + \mathrm{recc}(v)\}$. Now, we can safely stop the search as soon as the counter becomes zero and the all elements in the queue have a distance label no less than $d_{\max}$.

Due to the loose stopping criterion, we may set $\mathit{descend}[\ell][C]$ for level-ℓ cells C with no ev-reachable boundary vertex. Of course, such cells cannot contain any isochrone edges or isochrone pairs and thus do not need to be processed. By descending only into level-ℓ cells C for which $\mathit{reach}[\ell][C]$ is set, we avoid a considerable amount of work. See also section 4.3.

Recall that isochrone edges are determined during the RangeDijkstra searches. Hence, this works exactly as described in the previous section. When determining isochrone pairs, note that we need to take the same measures as in the standard scenario in order to deal with weakly connected cells. That is, we need to set some (consumption) eccentricities to infinity or, alternatively, spend more time during customization. See also section 4.1.5.

6.3.3 isoGRASP

Basically, isoGRASP needs the same extensions as we developed for the EV variant of isoCRP. However, recall that during customization, the isoGRASP algorithm additionally needs to build the downward graph $G_GS^{\downarrow}$. As in the standard scenario, the downward graph contains for each boundary vertex u on level ℓ a shortcut edge from u to each level-($\ell-1$) boundary vertex v in the interior of $c_\ell(u)$. The length of such a shortcut edge (u, v) is set to the length of the shortest $u-v$ path within the cell $c_\ell(u)$. In the EV scenario, we additionally store at each shortcut edge the cost function for the shortest $u-v$ path. Again, there may be multiple shortest paths to v and their cost functions may not dominate each other. However, such situations are easy to manage in the downward graph, since there is room for an arbitrary number of downward edges per pair u, v. More precisely, if there are n shortest $u-v$ paths with non-dominating cost functions, we insert n downward edges into $G_GS^{\downarrow}$. These downward edges all have the same length, namely the length of the shortest path to v, but different (non-do minted) cost functions.

7 Experimental Results

This chapter provides an experimental evaluation of the different algorithms discussed in this thesis. We mainly focus on computing isochrone edges in the standard scenario. After describing the inputs and experimental setup, we compare the performance of our implementations of some basic building blocks with the running times reported in the corresponding original publications. Then, we conduct some experiments to tune the parameters of our implementations, for example the number of cells used by the GS+PHAST algorithms. Afterwards, we evaluate the tuned algorithms in different experiments. The evaluation finishes with considering the computation of isochrone pairs and the EV scenario.

7.1 Inputs and Experimental Setup

Our code is written in C++ (with OpenMP for parallelization) and compiled with the GNU C++ compiler 4.8.1 using optimization level 3. We run most of our evaluation on a multi-socket machine that has two 8-core Intel Xeon E5-2680 CPUs and 256 GiB main memory. It runs SuSE Linux 13.1 (kernel 3.11.10). Each core is clocked at 2.70 GHz and has 64 KiB of L1, 256 KiB of L2, and 20 MiB of shared L3 cache.

Our main benchmark instance is the road network of Western Europe made available for the 9th DIMACS Implementation Challenge [24]. It has 18 million vertices and 42 million edges. The cost of an edge represents the travel time (in seconds) between its endpoints. We refer to it as DIMACS Europe. In the EV scenario, we use a proprietary (and not publicly available) road network, kindly provided by PTV AG. It also represents the European road network, but provides more information for each edge, such as physical lengths, road categories and speed limits. Hence, we are able to compute travel times of arbitrary precision. In contrast to DIMACS Europe, we use travel times measured in tens of seconds in the EV scenario. This reduces the total number of ambiguous shortest paths and leads to fewer non-dominating cost functions on the ambiguous shortest paths that are still present. We refer to the instance used in the EV scenario as PTV Europe. Additionally, we obtain height information for the vertices from the Shuttle Radar Topography Mission (SRTM) data, made publicly available

by NASA. SRTM data for Europe are sampled at three arc-seconds, which is about 90 meters.

To obtain consumption costs, we use the simulation tool PHEM (Passenger car and Heavy duty Emission Model) [45]. One application of PHEM is to compute the energy an electric vehicle consumes when traversing a road segment, depending on its physical length, road category, speed limit and slope. PTV Europe provides for each edge its physical length and speed limit, and also its road category. In order to map these road categories to the ones of PHEM, we use the heuristic developed by Baum et al. [9]. The slope for each edge is computed from the SRTM data. Note that we remove all vertices from the the benchmark instance where no height information is available, and all edges whose road categories cannot be mapped to PHEM. The final (strongly connected) road network has 22 million vertices and 51 million edges.

PHEM supports a huge number of vehicle categories. In our experiments, we use the model of a Peugeot iOn. The battery capacity is chosen such that the longest shortest path in the benchmark instance obeys the electric vehicle's battery constraints when starting with a full battery. We consider different initial charge levels to produce isochrones of varying size. Our choice of the maximum charge level allows us to compute isochrones that cover arbitrary fractions of the instance.

Our CH implementation follows [40], however, we use during contraction the priority term proposed by Delling et al. [13]. That is, the priority of a vertex is given by $2ED(u)+CN(u)+H(u)+5L(u)$, where $ED(u)$ is the change in the number of edges caused by the contraction of u, $CN(u)$ is the number of deleted neighbors, $H(u)$ is the number of original edges represented by all shortcuts added, and $L(u)$ is the (tentative) level of u. As in [13], we bound $H(u)$ such that each incident edge on u can contribute at most 3. To limit local searches, we use staged hop limits [13]. Again, we adopt the parameters in [13]. While the average degree of the remaining graph is at most 5, we limit the number of edges on witness paths to 5. The hop limit is then set to 10 until the average degree 10. When it exceeds 10, the hop limit is set to infinity. We stress that our CH preprocessing implementation is not tuned for speed and is also not parallelized.

Our GRASP implementation follows [30]. It includes acceleration techniques such as implicit initialization [30] and (downward) edge reduction [32]. However, whereas [30] use (reduced) adjacency lists to represent the cliques for the cells, we resort to the matrix representation from [15]. Note that Efentakis et al. [30] proposed in the original publication to use 16 overlay levels. Later [31], they reported that using a "normal" CRP partition (with much fewer levels) is only slightly less efficient. Hence, we also resort to partitions with four to six levels.

7.2 Basic Building Blocks

The algorithms for computing isochrones rely on some basic building blocks. This section compares the performance of our Dijkstra, PHAST and RPHAST implementations with the

running times reported in [17]. Since their CPU is clocked at 3.33 GHz, we do not run the experiments in this section on our main benchmark machine, but on a workstation with an Intel Core i7-2600K CPU clocked at 3.40 GHz. We consider two scenarios: one-to-all and one-to-many queries. In the one-to-all scenario, we pick a source s at random and compute shortest-path distances from s to all other vertices in the graph. The methodology of [17] is adopted in the one-to-many scenario. We fix the number of targets $|T|$ at 16384 (2^{14}), pick a center c at random and run Dijkstra's algorithm from c until reaching $|T|$ vertices. The visited vertices form a ball B of size $|B| = |T|$. Then, we pick a random source s from B and compute shortest-path distances from s to all other vertices in B.

Table 7.1 shows the results. It reports the average running time over 1000 random one-to-all queries. In the one-to-many scenario, we test 100 different centers, each with 100 different sources. Both selection and query times are sequential. For each algorithm, the query times are more or less equal. The target selection phase of our RPHAST implementation is even slightly faster than the original implementation. Note that the number of levels in the contraction hierarchy has a huge impact on the query times of RPHAST. Since our CH implementation produces contraction hierarchies with relatively many levels, we use a different (precomputed) contraction order for RPHAST with much fewer levels.

Table 7.1 – Performance of the basic one-to-all and one-to-many building blocks. Execution times are sequential. The one-to-many algorithms compute distances to the $|T| = 2^{14}$ targets in a ball of size $|B| = 2^{14}$.

	SELECTION TIME [MS]		QUERY TIME [MS]	
algorithm	our impl.	ref. [17]	our impl.	ref. [17]
Dijkstra (one-to-all)	–	–	2 789.12	–
PHAST	–	–	144.12	136.92
Dijkstra (one-to-many)	–	–	7.74	7.43
RPHAST	1.35	1.80	0.16	0.17

7.3 Parameter Tuning

Next, we consider the impact of the multilevel partition on the performance of the different algorithms. We also compare the three scheduling strategies proposed in section 4.2.2 to parallelize queries of the multilevel Dijkstra techniques. All queries in this section compute isochrone edges in the standard (non-EV) scenario and are run on our main benchmark machine described above.

7.3.1 Multilevel Dijkstra Techniques

We start by measuring the impact of the multilevel partition on isoCRP. Delling et al. use for their (point-to-point) CRP implementation four levels in the conference version [14] and five levels in the journal version [15]. Efentakis et al. [31] use six resort to six levels for their

SALT implementation. Hence, we expect the optimal number of levels to lie between 4 and 6. We test our isoCRP implementation with the 4-level partition from [14], which was created by using PUNCH [16]. Besides, we use Buffoon [56] to create further partitions for varying number of levels (L) and different cell sizes.

Table 7.2 reports sequential and parallel customization times. We test both a mid-range ($x = 100$ min) and a long-range time limit ($x = 500$ min). For each time limit, we report the average number of vertices scanned (delete-min operations on the priority queue) and the average sequential and parallel query times over 1000 random sources. Whereas sequential execution uses only one core, parallel customization and queries take advantage of all sixteen cores. The second column shows the number of cells per level, from bottom level to top level. The first row corresponds to the PUNCH partition mentioned above, the others correspond to Buffoon partitions. The best value in each column is highlighted.

Table 7.2 – Impact of the multilevel partition on the performance of isoCRP for varying number of levels (L) and different cell sizes. The first 4-level partition was created by using PUNCH. Buffoon was used for all other partitions.

		CUSTOM		QUERIES					
				limit $x = 100$ min			limit $x = 500$ min		
L	parameter	seq. [s]	par. [s]	# scans	seq. [ms]	par. [ms]	# scans	seq. [ms]	par. [ms]
4	[667963:20481:623:20]	**14.95**	**1.50**	**133 559**	**17.67**	**3.02**	**501 553**	**75.51**	**9.45**
	[2^{18}:2^{13}:2^{8}:2^{3}]	18.50	1.78	159 149	19.75	4.33	589 526	85.44	11.80
	[2^{19}:2^{14}:2^{9}:2^{4}]	18.72	1.81	147 652	21.34	3.55	561 172	92.83	10.81
5	[2^{19}:2^{15}:2^{11}:2^{7}:2^{3}]	17.28	1.82	144 438	20.66	6.09	532 437	86.92	12.13
	[2^{20}:2^{16}:2^{12}:2^{8}:2^{4}]	18.99	1.97	140 770	23.02	4.91	524 669	96.16	12.51
6	[2^{19}:2^{15}:2^{12}:2^{9}:2^{6}:2^{3}]	16.77	1.89	146 327	20.97	8.83	535 507	87.20	15.37
	[2^{20}:2^{16}:2^{13}:2^{10}:2^{7}:2^{4}]	18.91	2.13	142 696	23.39	6.82	527 437	96.85	13.28

We observe that the PUNCH partition leads to the best sequential and parallel performance, in terms of the customization times as well as queries times. We mainly tune for query times, however, the customization times (when using Buffoon partitions) are relatively stable. Moreover, customization scales reasonably well. With 16 cores, we see a speedup of around 10. When considering only the Buffoon partitions, we observe that four levels yield the fastest sequential and parallel queries for both mid-range and long-range time limits. Hence, 5- or 6-level PUNCH partitions would probably also lead to slightly slower queries than the 4-level PUNCH partition. For the remainder of our experiments, we thus use for isoCRP the 4-level partition created by using PUNCH.

Next, we consider the impact of the multilevel partition on isoGRASP. Since the (one-to-all) queries of GRASP tend to be faster when using somewhat more levels than we typically use for a CRP implementation [30], we expect that the 4-level partitions do not dominate as clearly as before. Table 7.3 shows the results. The structure of the table is identical to the one of Table 7.2. Again, we test mid-range and long-range queries and execute them

both sequentially and in parallel (using all sixteen cores). We report average values over 1000 random queries.

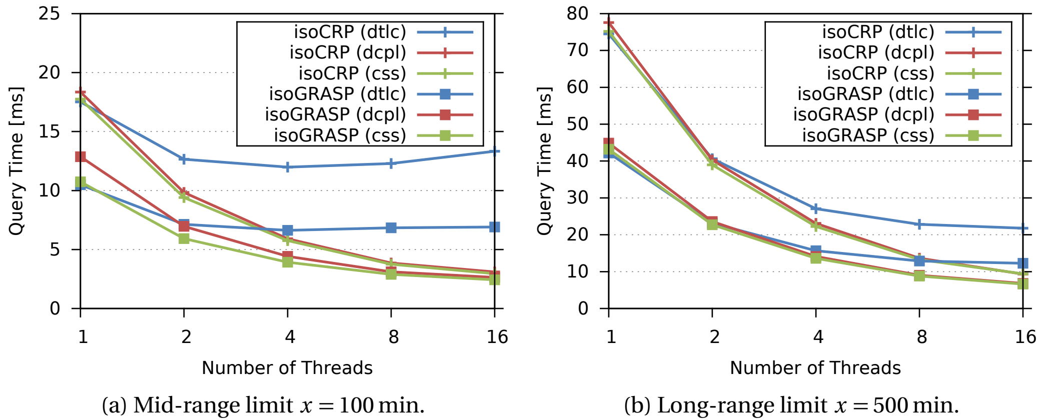
Table 7.3 – Impact of the multilevel partition on the performance of isoGRASP for varying number of levels (L) and different cell sizes. The first 4-level partition was created by using PUNCH. Buffoon was used for all other partitions.

		CUSTOM		QUERIES					
				limit $x = 100$ min			limit $x = 500$ min		
L	parameter	seq. [s]	par. [s]	# scans	seq. [ms]	par. [ms]	# scans	seq. [ms]	par. [ms]
4	[667963:20481:623:20]	**18.29**	**2.38**	157 516	10.92	**2.39**	548 632	43.89	**6.52**
	[2^{18}:2^{13}:2^{8}:2^{3}]	22.61	2.73	193 407	12.48	3.31	665 765	50.32	8.31
	[2^{19}:2^{14}:2^{9}:2^{4}]	22.35	2.65	173 028	12.05	2.73	611 826	49.96	7.47
5	[2^{19}:2^{15}:2^{11}:2^{7}:2^{3}]	20.92	2.59	161 058	10.62	3.36	552 532	43.23	7.30
	[2^{20}:2^{16}:2^{12}:2^{8}:2^{4}]	22.06	2.88	149 449	10.85	2.76	522 265	44.44	7.16
6	[2^{19}:2^{15}:2^{12}:2^{9}:2^{6}:2^{3}]	20.21	2.76	159 537	10.20	4.43	538 030	**40.43**	9.37
	[2^{20}:2^{16}:2^{13}:2^{10}:2^{7}:2^{4}]	21.49	3.10	**146444**	**10.04**	3.31	**501712**	40.66	8.19

As expected, the PUNCH partition still dominates in terms of customization times. Compared to the customization of isoCRP, we here have to build the downward graph in addition. However, we observe that the overhead is fairly reasonable. Sequential customization times slightly increase by 14-22 %, and parallel times increase by 42-59 %. Parallel times suffer more than sequential times, since the computation of shortcuts can be parallelized very well, by simply assigning cells to distinct cores. However, the computation of the downward graph always involves sequential work that cannot be parallelized (cf. section 4.3.2). For the same reason, customization of isoGRASP scales slightly worse then before.

In terms of sequential query times, the 6-level partitions now dominate the partitions with fewer levels, even the 4-level partition created by using PUNCH. This is consistent with the number of downward edges, which decreases from about 130 million (when using 4-level Buffoon partitions) to under 110 million (when using 6-level Buffoon partitions). Hence, the number of vertices scanned also decreases. Considering parallel query times, however, the 4-level PUNCH partition dominates. Since it leads, at the same time, to only slightly slower sequential queries, we also use it for isoGRASP for the remainder of our experiments. However, note that a 6-level partition created by using PUNCH may be the best choice for isoGRASP. Unfortunately, we have no such partition at hand.

Having fixed the partitions, it remains to compare the scheduling strategies proposed in section 4.2.2 to parallelize isoCRP and isoGRASP queries. Consider the following two plots. Fig. 7.1a shows the query times for a mid-range time limit $x = 100$ min and varying number of cores. Fig. 7.1b shows the same for a long-range time limit $x = 500$ min. There is no plot for a short-range time limit, since those queries are difficult to accelerate by using multiple cores. We report average values over 1000 random queries.



(a) Mid-range limit $x = 100$ min. (b) Long-range limit $x = 500$ min.

Figure 7.1 – Query times for varying number of cores using different scheduling strategies: distribute cells per level (dcpl), distribute top-level cells (dtlc) and combined scheduling strategy (css).

Basically, isoCRP and isoGRASP queries show the same behavior, both for mid-range and long-range queries. The combined scheduling strategy (CSS) and the strategy that distributes cells per level (DCPL) are almost equally fast, whereas the strategy that distributes the top-level cells (DTLC) is significantly worse when using a higher number of cores. We tried to combine the advantages of DCPL and DTLC in the combined scheduling strategy, thus we are not surprised to observe it performs best. Our main concern about DCPL was the synchronization overhead, since we need to synchronize the threads at each level. However, as we use only four levels, there are only three additional barriers, which has no impact on the overall performance. However, as suspected, the amount of parallelism when using DTLC is too small to fully occupy multiple cores. This is especially true for mid-range time limits, where it may happen that we do not leave the top-level cell at all. Note that we are not quite sure why sequential mid-range isoGRASP queries are slightly slower when using DCPL instead of one of the other strategies. For the remainder of our experiments, we parallelize isoCRP and isoGRASP queries using the combined scheduling strategy.

7.3.2 GS+PHAST Techniques

We move on to consider the impact of the partition on the GS+PHAST algorithms. Although we tune our implementations for query performance, we also want to show how different partitions influence the preprocessing time and space. Note, however, that our CH preprocessing routine is neither highly optimized nor parallelized. Hence, most preprocessing figures are probably far from perfect. Table 7.4 reports for each of the four GS+PHAST algorithms the preprocessing time and space when using different partitions. Note that the preprocessing space for both distance oracle variants does not include the space required to represent the downward subgraphs, since their size depends on their compression.

Table 7.4 – Impact of the number of cells on the (parallel) time [min:s] and space [MiB] for the preprocessing of the GS+PHAST techniques. Space usage of both distance oracle algorithms does not include the downward subgraphs, since their sizes depend on their compression.

		# CELLS							
algorithm	criterion	128	256	512	1 024	2 048	4 096	8 192	16 384
ES+PHAST (cd)	time	0:46	0:37	0:33	0:29	0:27	0:25	0:23	0:23
	space	764	767	770	774	777	782	787	796
ES+PHAST (cp)	time	20:52	22:17	20:58	20:49	18:21	17:58	17:30	16:13
	space	762	766	769	773	779	785	793	806
ES+PHAST (do)	time	20:56	22:24	21:08	21:06	18:53	19:01	19:39	20:17
	space	409	411	415	423	449	548	936	2 478
VS+PHAST	time	12:34	12:25	12:24	12:28	12:53	13:26	14:29	17:13
	space	403	404	405	411	435	531	915	2 451

We observe that the preprocessing time of the Core-Dijkstra variant decreases with the number of cells. This happens mainly because the time required to contract the cell graphs decreases from 36.53 seconds (when using 128 cells) to 12.68 seconds (when using 16384 cells), since the cell graphs become smaller and smaller. At the same time, the preprocessing space increases slightly, due to the larger core graphs.

The preprocessing of the Core-PHAST and distance oracle variant takes much longer. This is due to the contraction of the core graph, which makes up over 96 % of the total preprocessing time in case the number of cells is between 128 and 1024. Even when using 16384 cells, the core graph contraction still makes up 76 % of the total time. One reason for this is that, as already mentioned, our CH preprocessing routine is not parallelized, whereas the contraction of the cell graphs, the computation of the eccentricities and diameters, and the calculation of the distance oracle is done in parallel. However, even a parallelized CH preprocessing would take much longer than we are used to see. That is because we force the CH preprocessing to select a suboptimal contraction order, since we first block the contraction of core vertices. This not only leads to a worse contraction hierarchy, but also to slower preprocessing times.

The preprocessing time of the Core-PHAST variant slightly decreases with the number of cells. When using a partition with few cells, we contract more vertices while the contraction of some vertices is blocked. Hence, we “damage” the contraction order more than in the case of a partition with many cells, where we contract less vertices while some vertices are blocked. Note, however, that the preprocessing time of the oracle variant increase at some point again, since the computation of the distance oracle then also takes a significant amount of time (229.47 seconds in the case of 16384 cells). The distance oracle is also the reason for the steep increase in preprocessing space.

The preprocessing of VS+PHAST takes less time than the one of the distance oracle variant of ES+PHAST, since we perform a normal CH preprocessing without blocking the core vertices. Independent of the number of cells, our CH implementation takes about 660 seconds to contract DIMACS Europe. The preprocessing times increase with the number of cells, since

the computation of the distance oracle becomes more expensive. For the same reason, the preprocessing space also increases.

We now evaluate the impact of the compression optimization proposed in section 5.2.3 on the size of the downward subgraphs that we use for the oracle variant of the ES+PHAST algorithm. Table 7.5 reports the size of the compressed subgraphs for different partitions and varying choices of the parameter k (cf. section 5.2.3). The size of the downward subgraphs increases with the number of cells, since more and smaller cells lead to downward subgraphs that overlap significantly. In contrast, the size of the downward subgraphs decreases with the parameter k. The compression works very well even for small k. For example, when using a partition with 16384 cells and $k = 32768$ (after all, that are only 0.002 % of the vertices of the whole graph), the size decreases by a factor of about 25 from 15 GiB to 619 MiB.

Table 7.5 – Space [MiB] required to represent the downward subgraphs in ES+PHAST (do) for different partitions and varying choices of the parameter k.

core	# CELLS							
size	128	256	512	1 024	2 048	4 096	8 192	16 384
128	617	849	1 307	2 124	3 561	6 107	9 983	14 970
256	605	831	1 267	2 042	3 415	5 846	9 485	13 974
512	579	789	1 180	1 879	3 131	5 321	8 524	12 115
1 024	529	700	1 020	1 563	2 593	4 402	6 730	9 051
2 048	453	545	765	1 077	1 794	3 008	4 289	5 562
4 096	425	450	523	678	1 045	1 663	2 370	3 164
8 192	422	426	445	499	640	908	1 308	1 786
16 384	421	423	427	438	475	566	750	980
32 768	421	423	425	428	436	459	515	619

Table 7.6 reports the size of the compressed subgraphs that we use for the VS+PHAST algorithm. Note that these subgraphs are in general significantly smaller then their counterparts used for the oracle variant of ES+PHAST. Since we use a "standard" contraction hierarchy (with a normal contraction order) for VS+PHAST, the subgraphs extracted from the hierarchy overlap much less, resulting in lower space requirements.

Table 7.6 – Space [MiB] required to represent the downward subgraphs in VS+PHAST for different partitions and varying choices of the parameter k.

core	# CELLS							
size	128	256	512	1 024	2 048	4 096	8 192	16 384
128	479	536	640	802	1 117	1 838	3 356	6 029
256	474	523	616	755	1 040	1 659	2 985	5 375
512	463	502	574	681	901	1 353	2 344	4 247
1 024	450	474	524	588	742	1 009	1 586	2 754
2 048	439	453	484	525	612	772	1 062	1 614
4 096	431	440	458	485	534	627	780	1 085
8 192	426	432	442	457	485	534	615	766
16 384	422	426	432	441	457	483	525	602
32 768	420	422	426	432	441	456	481	523

Next, we evaluate the performance of the Core-Dijkstra and Core-PHAST variant when using different partitions. Table 7.7 reports sequential and parallel query times for both variants. For each combination, it shows the running time of short-, mid- and long-range queries. We use time limits of 10 minutes, 100 minutes and 500 minutes, respectively. Moreover, we compute for each combination the variant whose query times deviate the least from the respective best running times. These variants are highlighted in dark gray. We also highlight the variants with the least relative deviation in light gray. As always, we report average the running time over 1000 random queries.

Table 7.7 – Performance of ES+PHAST (cd) and ES+PHAST (cp) for varying number of cells. Queries are executed sequentially (seq.) and in parallel (par.) and given in milliseconds. The variant with the least absolute (relative) deviations is highlighted in dark (light) gray.

algorithm	exec.	limit [min]	# CELLS 128	256	512	1 024	2 048	4 096	8 192	16 384
ES+PHAST (cd)	seq.	10	1.96	1.11	0.67	0.43	0.31	**0.25**	0.26	0.32
		100	12.38	10.24	8.88	7.92	7.26	6.73	6.38	**6.32**
		500	70.89	58.58	49.57	42.24	38.15	**36.49**	38.17	43.72
	par.	10	1.76	0.90	0.48	**0.29**	**0.29**	0.49	0.94	2.32
		100	2.23	1.61	1.35	**1.33**	1.47	1.86	2.64	4.42
		500	8.92	**8.84**	9.50	10.70	12.93	16.56	22.15	31.48
ES+PHAST (cp)	seq.	10	3.55	**3.31**	3.59	4.22	5.27	6.93	9.87	13.47
		100	14.17	12.69	12.07	**11.96**	12.51	13.64	15.49	17.94
		500	71.58	59.04	49.51	40.94	35.14	30.91	28.74	**28.29**
	par.	10	3.55	**3.39**	3.75	4.28	5.61	7.04	10.18	13.55
		100	**4.15**	**4.15**	4.59	5.22	6.57	7.92	11.21	14.26
		500	8.67	**7.61**	7.88	7.71	8.58	9.49	12.50	15.37

We observe that sequential queries of the Core-Dijkstra variant are slow for partitions with both too few cells and too many cells. When using few cells, we cannot narrow down the cells that may contain isochrone edges very well. Hence, a large fraction of the cells is marked as active and needs to be processed, which slows down queries. Conversely, when using many cells, there is also a large number of core vertices and thus the whole core graph becomes rather large. As a consequence, the RangeDijkstra search on the core graph is a performance bottleneck. A medium-sized partition (in terms of the number of cells) provides a good trade-off between the fraction of cells that are marked as active and the amount of work that has to be done core graph size.

If multiple cores are available, we can accelerate the processing of the active cells significantly, however, the RangeDijkstra search on the core graph remains sequential. Hence, the core search becomes much earlier an performance bottleneck, so that parallel queries of the Core-Dijkstra variant are faster when using partitions with less cells than in the single-core setup. Although queries then scan much more vertices in total, the parallelization compensates for this drawback.

We notice a similar behavior for the Core-PHAST variant. Partitions with few cells lead to a large fraction of active cells. Conversely, partitions with many cells lead to large core graphs and thus the Core-PHAST becomes a bottleneck. In contrast to the Core-Dijkstra, which stops when all elements in the priority queue have a distance label greater than the time limit, the Core-PHAST has no early termination criterion, but always processes the whole core graph. As a consequence, short-, mid- and long-range queries are fastest at different partitions. For short-range queries, the Core-PHAST becomes earlier a performance bottleneck than for mid- or long-range queries.

As before, since we parallelize the processing of the active cells in a multi-core setup, but not the Core-PHAST, it becomes earlier a bottleneck. Hence, partitions with fewer cells also lead to faster mid- and long-range queries. For the remainder of our experiments, we use the variants with the least absolute deviation from the respective best query times, which are highlighted in Table 7.7 in dark gray.

Finally, we evaluate the performance of the distance oracle variants when using different partitions, starting with the oracle variant of ES+PHAST. Table 7.8 reports query times for different partitions and varying choices of the compression parameter k. For each combination, it again shows the running time of short-, mid- and long-range queries, using the same time limits as before. Moreover, we again highlight the variant with the least absolute deviation in dark gray, and the one with the least relative deviation in light gray. Running times are averages over 1000 random queries. Note that the queries use multiple cores. There is a similar table in the appendix that reports sequential query times.

Once again, we observe that the query times are slow for partitions with both too few and too many cells. The problem with using few cells is the same as described above. We cannot narrow down the active cells very well and thus need to process a large fraction of the cells. When using many cells, we have a large core graph and thus the fourth phase of the algorithm (cf. section 5.2.3) becomes a performance bottleneck, since it loops through all edges that connect two core vertices.

The compression parameter k does not only have a great impact on the size of the downward subgraphs, but also on the query performance. For small k, many vertices are contained in multiple subgraphs, and thus high-rank vertices are processed again and again during the same query. Obviously, this slows down queries. However, for large k, there are many "unnecessary" vertices among the topmost k vertices that we actually do not need to scan for the current query. Hence, query times also increase. Using a partition with 1024 cells and choosing $k = 4096$ or $k = 8192$ offers a good trade-off.

Let us finish this section with evaluating the performance of VS+PHAST when using different partitions. Table 7.9 again reports query times for different partitions and varying choices of the parameter k. The structure of the table is identical to the one of Table 7.8. Again, the queries use multiple cores. A table reporting sequential times is contained in the appendix.

Table 7.8 – Parallel query times [ms] of ES+PHAST (do) for different partitions and varying choices of the parameter k. The variant with the least absolute (relative) deviations is highlighted in dark (light) gray.

core	limit	# CELLS							
size	[min]	128	256	512	1 024	2 048	4 096	8 192	16 384
128	10	6.28	4.38	3.63	3.27	3.21	3.29	3.63	4.21
	100	6.91	5.45	5.20	6.26	7.83	10.05	12.72	15.54
	500	13.95	12.11	12.46	15.29	22.61	28.84	35.93	41.18
256	10	6.22	4.43	3.54	3.36	3.17	3.38	3.49	4.57
	100	6.88	5.45	5.06	6.21	7.48	9.77	12.17	15.03
	500	13.70	12.14	12.31	15.17	21.68	27.87	34.37	39.93
512	10	6.23	4.36	3.54	3.44	3.11	3.22	4.06	3.95
	100	6.83	5.36	5.03	5.92	7.28	9.27	12.27	13.77
	500	13.44	11.75	11.92	14.21	20.44	26.08	32.59	35.92
1 024	10	6.15	4.31	3.45	2.99	3.21	3.26	3.29	3.85
	100	6.71	5.23	4.83	5.48	6.59	8.42	9.86	11.65
	500	13.08	11.25	11.15	12.66	17.86	22.65	26.26	28.73
2 048	10	6.06	4.16	3.26	2.84	2.90	3.10	3.17	3.72
	100	6.58	4.96	4.56	4.81	5.71	6.99	7.78	8.81
	500	12.52	10.36	9.98	10.46	14.14	17.35	18.86	19.42
4 096	10	6.19	4.15	3.30	2.81	**2.72**	3.01	3.12	3.72
	100	6.70	5.02	4.53	**4.23**	4.79	5.55	6.12	7.03
	500	12.50	9.97	8.98	8.31	10.32	11.97	12.76	13.60
8 192	10	6.41	4.40	3.38	2.97	2.87	3.12	3.25	3.83
	100	6.88	5.25	4.62	4.31	4.52	5.05	5.26	6.04
	500	12.65	10.06	8.81	**7.94**	8.53	9.14	9.47	10.24
16 384	10	6.79	4.71	4.01	3.41	3.28	3.58	3.67	4.17
	100	7.29	5.53	5.07	4.77	4.79	4.92	10.44	5.72
	500	13.37	10.32	9.10	8.15	8.09	7.97	8.89	8.48
32 768	10	7.00	5.13	4.25	4.39	4.29	4.74	5.07	4.92
	100	7.51	5.98	5.49	5.46	5.51	5.82	5.84	6.20
	500	13.58	10.96	9.56	8.76	8.83	8.41	8.06	8.29

We observe similar effects as before. That is, too small parameters k lead to many vertices scanned multiple times, and too large k yield many unnecessary scans. Moreover, partitions with too few cells cause a large fraction of active cells. However, the VS+PHAST algorithm does not loop through all edges connecting two core vertices after processing the active cells. This allows us to use partition with more cells than we can use for the oracle variant of ES+PHAST, decreasing the fraction of cells that are marked as active. However, at some point, cells become too small, such that the higher accuracy in terms of active cells does not outweigh the additional overhead. After all, when using 16384 cells, there are only about 1100 vertices in each cell. For the remainder of our experiments, we use a partition with 8192 cells and also set $k = 8192$ for VS+PHAST.

Table 7.9 – Parallel query times [ms] of VS+PHAST different partitions and varying choices of the parameter k. The variant with the least absolute (relative) deviations is highlighted in dark (light) gray.

core size	limit [min]	# CELLS 128	256	512	1 024	2 048	4 096	8 192	16 384
128	10	4.12	2.54	1.71	1.16	0.98	0.89	0.93	1.12
	100	4.61	3.27	2.68	2.36	2.77	3.59	4.82	6.97
	500	10.17	8.22	7.60	7.19	8.76	10.89	14.89	20.72
256	10	4.12	2.52	1.68	1.14	0.92	0.88	0.95	1.20
	100	4.58	3.24	2.63	2.26	2.69	3.26	4.48	6.42
	500	10.08	8.07	7.37	6.82	8.39	10.07	13.51	18.77
512	10	4.10	2.48	1.63	1.10	0.89	0.84	0.87	1.10
	100	4.53	3.17	2.53	2.13	2.48	2.88	3.78	5.46
	500	9.91	7.81	6.95	6.29	7.53	8.63	11.03	15.37
1 024	10	4.06	2.43	1.59	1.05	0.84	0.78	0.83	1.04
	100	4.47	3.08	2.41	1.96	2.24	2.44	2.94	4.16
	500	9.69	7.47	6.49	5.57	6.74	6.96	8.12	10.76
2 048	10	4.02	2.40	1.57	1.04	0.85	**0.77**	0.81	1.01
	100	4.45	3.01	2.36	1.86	2.01	2.11	2.39	2.99
	500	9.53	7.25	6.07	5.08	5.86	5.82	6.12	7.19
4 096	10	4.05	2.44	1.60	1.07	0.90	0.82	0.91	1.06
	100	4.73	3.03	2.32	**1.83**	1.95	1.96	2.10	2.47
	500	9.57	7.15	5.87	4.83	5.48	5.18	5.09	5.57
8 192	10	4.13	2.52	1.70	1.17	1.00	1.00	1.03	1.19
	100	4.55	3.11	2.40	1.89	1.98	1.94	1.97	2.25
	500	9.49	7.15	5.85	4.76	5.33	4.86	4.57	4.65
16 384	10	4.31	2.71	1.88	1.41	1.37	1.43	1.47	1.54
	100	4.73	3.30	2.59	2.09	2.16	2.11	2.09	2.29
	500	9.63	7.29	5.97	4.89	5.38	4.83	4.43	**4.35**
32 768	10	4.66	3.07	2.28	2.07	2.04	2.09	2.15	2.22
	100	5.12	3.70	2.99	2.55	2.61	2.56	2.54	2.68
	500	9.98	7.65	6.32	5.22	5.60	5.14	4.71	4.52

7.4 Main Results

Table 7.10 summarizes the main results of our work. It compares the performance of the different algorithms discussed in this thesis on DIMACS Europe when computing isochrone edges. For each algorithm it reports preprocessing time and space, customization time and space (if applicable), and sequential and parallel query times. Preprocessing and customization are executed in parallel. The time required to create (multilevel) partitions is not included in the figures. Note that we report for the GS+PHAST algorithms the preprocessing time for the multi-core setup. For both sequential and parallel queries, we provide running times for mid-range and long-range time limits. The best value in each column is highlighted in bold.

Table 7.10 – Performance of the different algorithms on DIMACS Europe when computing isochrone edges. Preprocessing and customization times are given for multi-threaded execution, while queries are run single- (seq.) and multi-threaded (par.).

	PREPRO		CUSTOM		SEQ. QUERIES [MS]		PAR. QUERIES [MS]	
	time	space	time	space	limit [min]		limit [min]	
algorithm	[s]	[MiB]	[s]	[MiB]	$x = 100$	$x = 500$	$x = 100$	$x = 500$
RangeDijkstra	–	645	–	–	59.00	966.79	–	–
isoCRP (1-phase)	28.39	762	**1.50**	138	26.20	114.24	–	–
isoCRP	28.39	762	**1.50**	138	17.58	75.22	3.02	9.08
isoGRASP	28.39	762	2.38	1 093	10.73	43.23	2.43	6.20
ES+PHAST (cd)	36.93	767	–	–	**6.50**	35.17	**1.57**	8.33
ES+PHAST (cp)	1 336.70	766	–	–	12.74	29.30	4.09	7.50
ES+PHAST (do)	1 265.59	880	–	–	22.21	46.59	4.33	8.10
VS+PHAST	868.52	1 530	–	–	10.24	**27.77**	2.13	**4.09**

We observe that the multilevel Dijkstra techniques provide the better trade-off between preprocessing, customization and query times. Yet the GS+PHAST algorithms achieve the best query times. The Core-Dijkstra variant of ES+PHAST dominates for mid-range time limits, whereas VS+PHAST is the fastest algorithm for long-range time limits. As discussed in section 7.3.2, the slow preprocessing times of the last three algorithms in the table are mainly due to expensive graph contractions. Note the whereas the customization of isoGRASP is only slightly slower then the one of isoCRP (cf. section 7.3.1), it requires almost an order of magnitude more space to represent the downward graph, which contains about 110 million downward edges. Except RangeDijkstra, all algorithms enable queries that are fast enough for practical applications.

Fig. 7.2 shows sequential query times for varying time limits when computing isochrone edges. Each point is the average of 1000 random queries. Note that the network diameter of DIMACS Europe is about 4710 minutes. We observe that the curves for all algorithms (except RangeDijkstra) follow the same pattern. First, query times increase with the time limit. For larger time limits, the isochrone area becomes larger and with it the length of isoline longer. Since at least all cells intersected by the isoline become active cells, the work and thus query times increase for larger time limits. However, at some point, the isoline meets the “periphery” or “boundary” of the road network. Beyond this point, the number of isochrone edges and active cells decreases again. If the time limit is no less than the network diameter, each vertex is time-reachable from each other vertex, and thus there are neither isochrone edges nor active cells. Hence, queries are very fast again for the largest time limits.

For short-range time limits, the multilevel Dijkstra techniques dominate. Queries of the oracle variant of ES+PHAST take even for the smallest (and largest) time limits over ten milliseconds, due to their fourth phase which loops through all edges connecting two core vertices, in order to determine isochrone edges among them (cf. section 5.2.3). This phase takes about nine milliseconds and thus slows down the queries significantly. The Core-PHAST variant of ES+PHAST is slowed down by the linear sweep through the core graph, which

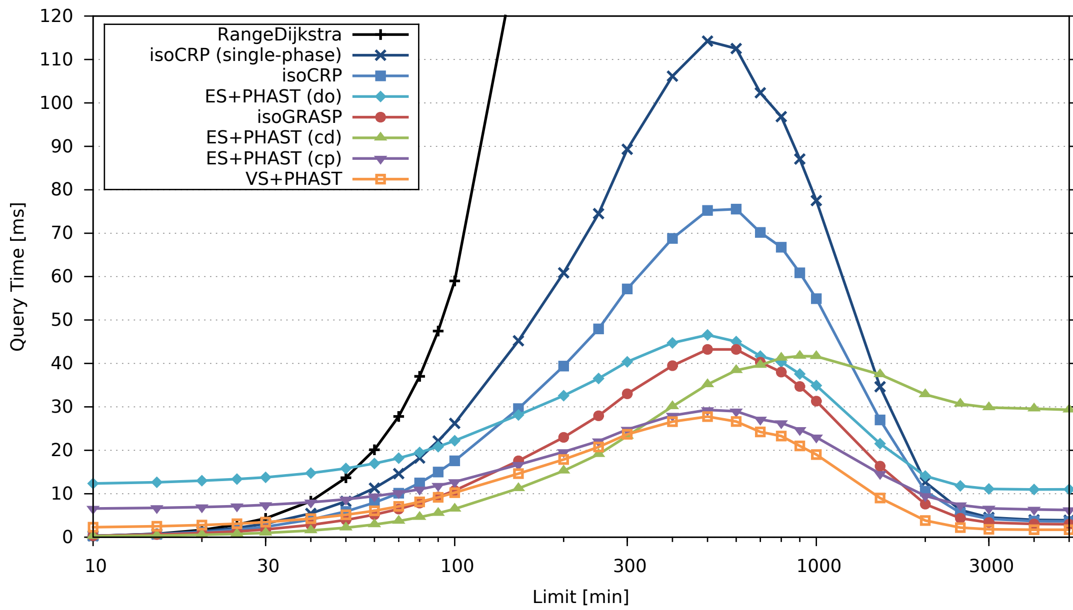


Figure 7.2 – Sequential query times for varying time limits when computing isochrone edges. The network diameter of the benchmark instance is about 4710 minutes.

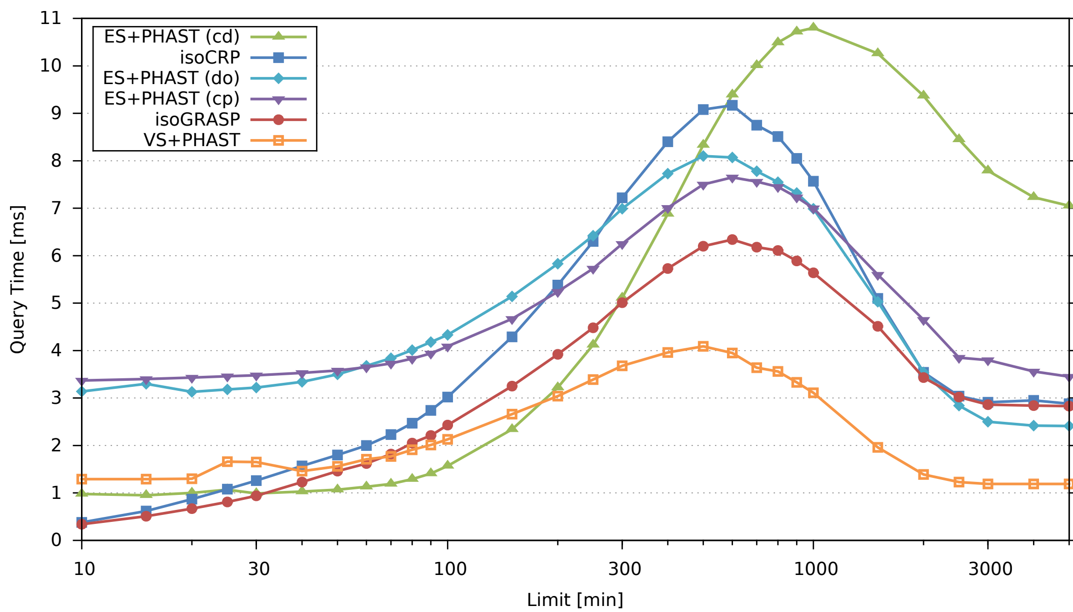


Figure 7.3 – Parallel query times using 16 cores for varying time limits when computing isochrone edges.

is independent from the time limit and takes about six milliseconds. Finally, VS+PHAST queries are somewhat slow for small time limits, since the distance oracle is not tight. Among the GS+PHAST algorithms, the Core-Dijkstra variant is the only one that is competitive for small time limits, since its search on the core graph has an early termination criterion. Unfortunately, the RangeDijkstra search does not scale to large time limits, causing queries of the Core-Dijkstra variant to be by far the slowest for those limits.

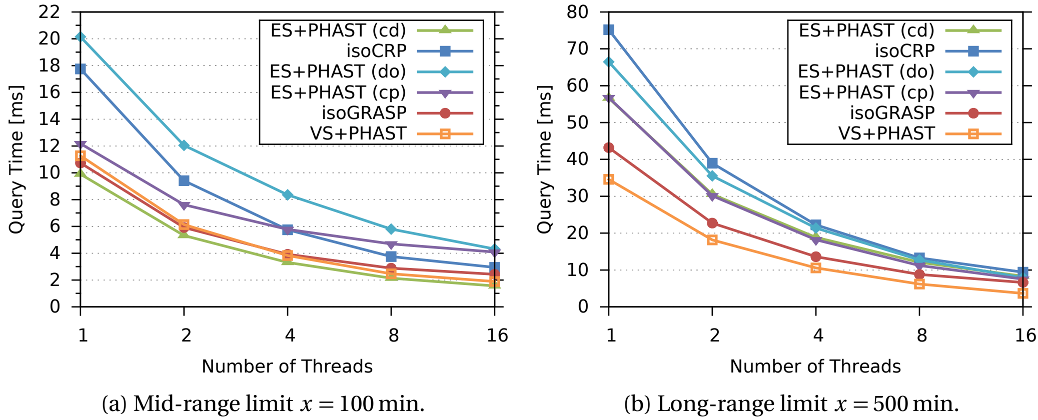
Moreover, we observe that single-phase isoCRP queries are even in a single-core setup significantly slower than their two-phase counterparts, due to bad locality and cache efficiency. However, even single-phase queries are fast enough for most practical applications. The isoGRASP algorithm is up to almost twice as fast as isoCRP. Hence, it provides a somewhat different trade-off customization space and query times. However, almost one gigabyte for storing the downward graph may be to much if each user should have an own cost function.

Fig. 7.3 shows parallel query times (using 16 cores) for varying time limits when computing isochrone edges. Again, each point is the average of 1000 random queries. Note that the plot does not contain curves for RangeDijkstra or single-phase isoCRP, since these algorithms are difficult to parallelize. The other curves mainly resemble the ones in the single-core setup. Queries of the Core-PHAST and distance oracle variant of ES+PHAST are still (relatively) slow for small time limits, due to the same reasons as described above. For the Core-PHAST variant, the linear sweep through the core graph still takes most of the query time. However, it is slightly faster than before, since we use smaller core graphs (partitions with fewer cells) in the multi-core setup (cf. section 7.3.2). For the distance oracle variant, its fourth query phase is still the main performance bottleneck, although the edges between core vertices are distributed among the cores (what explains the faster query times compared to the single-core setup).

The Core-Dijkstra variant is the best technique for mid-range queries again, however, as before, query performance get worse quickly for larger time limits, due to the scaling behavior of the RangeDijkstra search on the core graph. Note that the performance gap between isoCRP and isoGRASP is smaller when using multiple cores. Since isoCRP comes with more computational overhead than isoGRASP (e.g., much more operations on the priority queue), we conjecture that isoGRASP is limited by the memory bandwidth. The conjecture is supported by the fact that isoGRASP benefits significantly from storing a copy of the downward graph on each NUMA node. In contrast, the impact of keeping a copy of the clique matrices on each NUMA node on isoCRP queries is limited.

Finally, we evaluate how queries scale as the number of CPU cores increases. Fig. 7.4a plots mid-range query times against the number of cores used, and Fig. 7.4b does the same for long-range query times. As we expect after the previous experiments, the Core-Dijkstra variant of ES+PHAST is the best algorithm for mid-range time limits, and VS+PHAST is the fastest technique for long-range limits. All algorithms scale reasonably well (but not perfect) for long-range queries. With 16 cores, we see a speedup between 7 and 9. The isoGRASP

algorithm scales worst (speedup of 6.5), probably because it is memory bandwidth bounded. The VS+PHAST algorithm scales best (speedup of 9.4), since the only sequential parts are a forward upward CH search and questioning the distance oracle, that are both extremely fast. Hence, processing the active cells takes up to 99 % of the total running time for VS+PHAST queries, and this phase can be parallelized well. As one may expect, the speedups are slightly lower for mid-range queries.



(a) Mid-range limit $x = 100$ min.

(b) Long-range limit $x = 500$ min.

Figure 7.4 – Query times for varying number of cores when computing isochrone edges.

7.5 Computing Isochrone Pairs

Up to now, we have focused on computing isochrone edges. This section evaluates the performance of different algorithms when computing isochrone pairs. Recall that we proposed for the multilevel Dijkstra techniques two approaches to handle weakly connected cells (cf. section 4.1.5). Either we need to set the eccentricity of a boundary vertex u to infinity whenever there is at least one vertex v in the cell such that no $u-v$ path exists within the cell. Or we may not restrict searches during customization to cells, but allow them to cross cell boundaries. Whereas the first approach causes almost no overhead during customization, but slows down queries, the second approach leads to tighter eccentricities and thus faster queries, but slows down customization. In this section, we test the first approach. The evaluation of the second one remains for future work.

Table 7.11 reports some explicit sequential and parallel query times on DIMACS Europe when computing isochrone pairs. We omit preprocessing and customization times, since they are about the same as when computing isochrone edges. After all, we just add an additional check to the customization of the multilevel Dijkstra techniques that tests for each boundary vertex whether all other vertices in the cell are reachable within the cell. Fig. 7.5 shows sequential query times on DIMACS Europe for varying time limits. Each point is the average over 1000 random queries.

Table 7.11 – Performance of the different algorithms on DIMACS Europe when computing isochrone pairs. Query times are given for single- (seq.) and multi-threaded (par.) execution. Multilevel Dijkstra techniques use eccentricities of infinity, if necessary.

	SEQ. QUERIES [MS]			PAR. QUERIES [MS]		
	limit [min]			limit [min]		
algorithm	$x = 10$	$x = 100$	$x = 500$	$x = 10$	$x = 100$	$x = 500$
RangeDijkstra	**0.27**	59.11	973.45	–	–	–
isoCRP (1-phase)	0.40	36.95	404.91	–	–	–
isoCRP	0.33	24.96	255.77	0.41	3.94	24.48
isoGRASP	0.33	13.44	117.72	**0.35**	2.70	12.43
ES+PHAST (cp)	6.80	13.77	31.99	3.44	4.14	7.67
ES+PHAST (do)	12.72	22.31	46.27	3.13	4.39	8.16
VS+PHAST	2.47	**10.71**	**29.13**	1.75	**2.55**	**4.55**

We observe that the impact of the change from isochrone edges to isochrone pairs is limited for the GS+PHAST algorithms. However, queries of the multilevel Dijkstra techniques are much slower for large time limits than when computing isochrone edges. This happens due to the high number of weakly connected cells, resulting in many eccentricities of infinity. As a consequence, query times do not decrease when the time limit gets close to the network diameter. We conclude that our first approach to handle weakly connected cells is not practical. Note that there is a second plot in the appendix that shows parallel query times on DIMACS Europe when computing isochrone pairs.

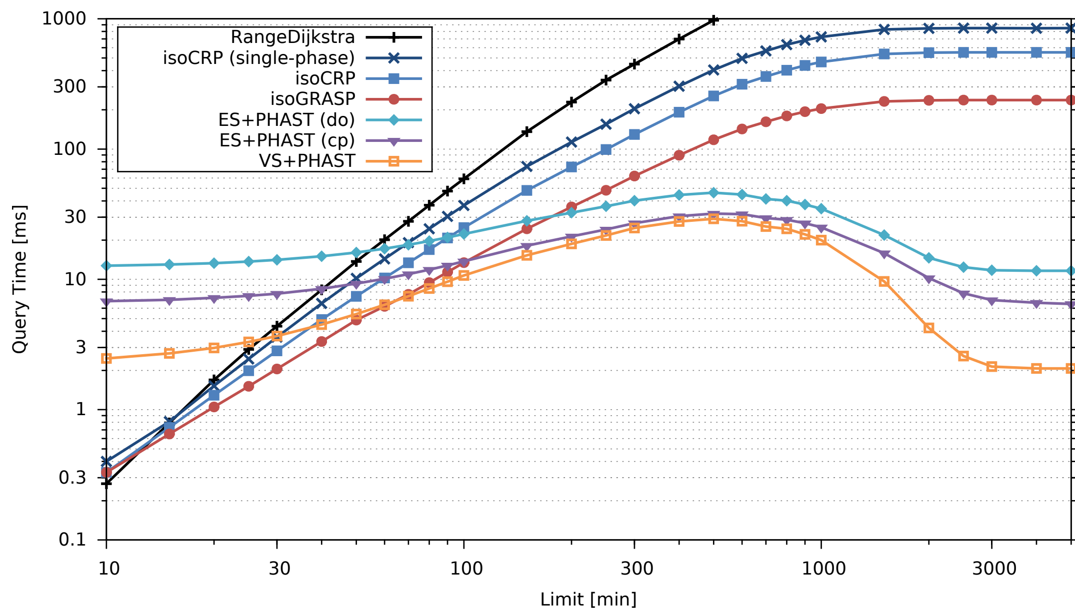


Figure 7.5 – Sequential query times for varying time limits when computing isochrone pairs.

We finish this section with Fig. 7.6, which plots the number of isochrone edges and the number of isochrone pairs against the time limit. As the plots seen so far suggest, the number of isochrone edges increases until the time limit is 500 minutes. Beyond this point,

the number decreases again. If the time limit is no less than the network diameter, there are no isochrone edges at all. The curve for the number of isochrone pairs follows the same pattern. By definition, the number of isochrone pairs must never be less than the number of isochrone edges. For long-range time limits, it is up to 3 % higher.

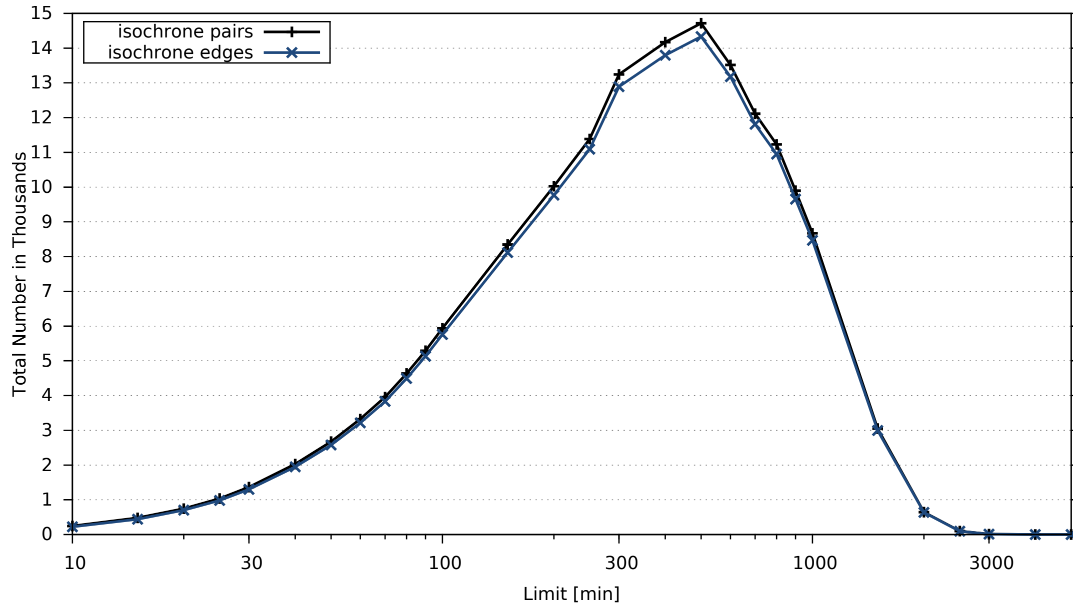


Figure 7.6 – Number of isochrone edges and isochrone pairs for varying time limits.

7.6 Electric Vehicle Scenario

This section finishes the experimental evaluation with considering the EV scenario. Table 7.12 compares the performance of the algorithms adapted to the EV scenario on PTV Europe when computing isochrone edges. For each algorithm, it reports preprocessing time and space, customization time and space, and sequential and parallel query times. As in section 7.4, preprocessing and customization are executed in parallel. Again, the time required to create the multilevel partition is not included in the preprocessing times. For both sequential and parallel queries, we provide the running time of mid- and long-range queries, using initial charge levels of 30 kWh and 250 kWh, respectively. The best value in each column is highlighted in bold.

We need to be careful when comparing the results from the EV scenario with the figures from the standard scenario, since we use a different benchmark instance here. However, as it has 23 % more vertices and 21 % more edges, we expect it to be “harder” for our algorithms. We use the 4-level partition from [9], which was created by using PUNCH. The higher preprocessing time and space are not directly related to the additional consumption metric, but are due to the fact that the instance is different. Although PTV Europe is about 20 % larger than DIMACS Europe and we need to maintain consumption cost functions, customization is

Table 7.12 – Performance of the different algorithms in the electric vehicle (EV) scenario. Preprocessing and customization times are given for multi-threaded execution, while queries are run single- (seq.) and multi-threaded (par.).

	PREPRO		CUSTOM		SEQ. QUERIES [MS]		PAR. QUERIES [MS]	
	time	space	time	space	limit [kWh]		limit [kWh]	
algorithm	[s]	[MiB]	[s]	[MiB]	$x = 30$	$x = 250$	$x = 30$	$x = 250$
RangeDijkstra	–	1 558	–	–	184.45	2 543.00	–	–
isoCRP (1-phase)	77.59	1 642	**1.60**	550	15.03	61.92	–	–
isoCRP	77.59	1 642	**1.60**	550	14.67	50.74	4.60	10.07
isoGRASP	77.59	1 642	3.22	3 020	**9.54**	**30.83**	**4.23**	**9.23**

only slightly slower. However, the space required to represent the clique matrices increases by a factor of about 4. This is plausible since instead of one 32-bit integer, we now need four integers to store the length and the consumption cost function for each shortcut. Besides, we need some space for the array that stores the cost functions for shortcuts that have multiple non-dominated cost functions. As mentioned in section 6.3.2, when using travel times in tens of seconds, only 0.00009 % of the shortcuts have multiple cost functions, resulting in a total number of additional cost functions of 6028 (71 KiB). Note that the space required to represent an downward edge increases from two 32-bit integers (tail, length) to five integers (tail, length, consumption cost function). Hence, the downward graph increases by a factor of about 2.5.

Fig. 7.7 shows sequential query times on PTV Europe for varying initial charge levels. Note that with the largest initial charge level, each vertex is ev-reachable from each other vertex and thus there are no isochrone edges. Each point is the average over 1000 random queries. We observe the the curves follow a similar pattern as in section 7.4. Queries of isoGRASP are still the fastest, followed by two-phase queries of isoCRP. The single-phase counterparts of the latter perform worst. However, what is interesting is that the query times of the multilevel Dijkstra techniques are faster by a factor of about 1.5, compared to the running times in the standard scenario. This happens because isochrones for electric vehicles contain much less isochrone edges. As shown in section 7.5, there are up to about 14000 isochrone edges in the standard scenario, however, we have only up to about 3600 isochrone edges when computing isochrones for electric vehicles. Hence, there are less active cells, which leads to improved query performance. Comparing the shape of isochrone areas in both scenarios (using visualization algorithms) remains for future work. We think it is possible that isochrones are star-shaped for large time limits, but rather circular when using electric energy as consumption metric. Note that there is a second plot in the appendix that shows parallel query times for varying initial charge levels when computing isochrone pairs.

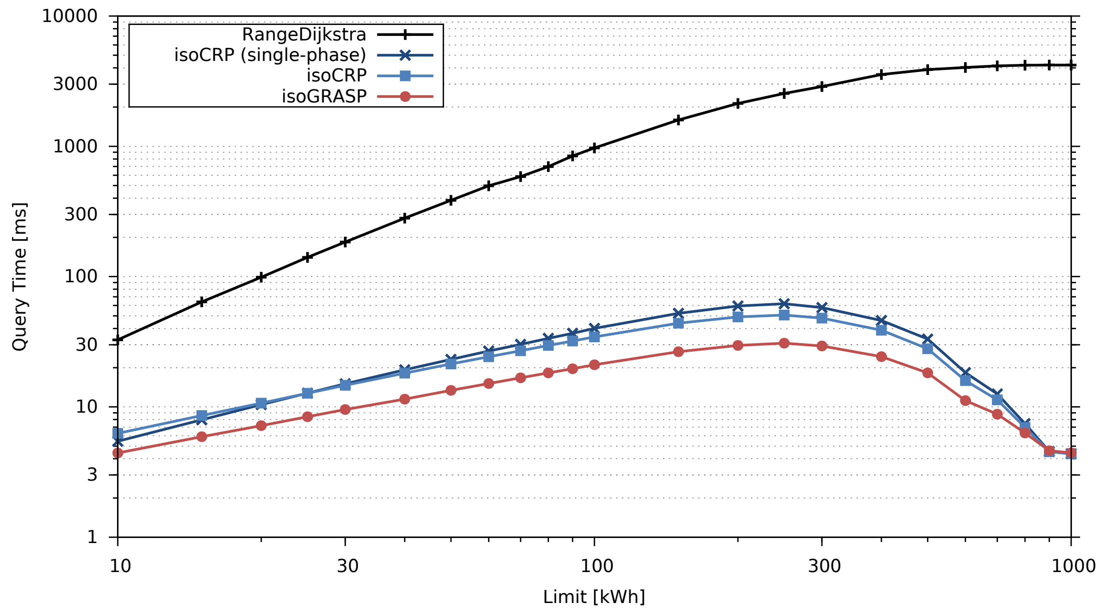


Figure 7.7 – Sequential query times for varying initial battery charge levels when computing isochrone edges. The network diameter of the benchmark instance is about 986 kWh.

8 Conclusion

This chapter finishes the thesis by summarizing our contributions and results. Afterwards, we outline possible future work in the context of isochrones.

8.1 Summary

We have studied the problem of computing isochrones in road networks. Practical applications include reachability analysis in urban planning, geomarketing, and displaying a electric vehicle's remaining cruising range. Although one may have a fairly good intuitive understanding of what isochrones are, the formal definitions varies among the literature. We systematically collated different definitions and introduced the notion of isochrone edges and isochrone pairs, which are two possible outputs for algorithms computing isochrones.

We started by describing RangeDijkstra, a Dijkstra variant that outputs isochrone edges or, if required, isochrone pairs. Since its performance is reasonable only for small time limits, we considered different speedup techniques. First, we explored multilevel overlay graphs to accelerate RangeDijkstra. Besides proposing a basic multilevel algorithm that is one order of magnitude faster than RangeDijkstra, we revisited and extended the isoCRP and isoGRASP algorithm, two known techniques for computing isochrones. We found that all multilevel algorithms provide a reasonable trade-off between preprocessing time, customization time and query time.

Motivated by promising query times of RPHAST, we proposed the novel family of GS+PHAST algorithms. They decompose the road network into several cells using k-way graph separators, determine which of the cells may contain isochrone edges (or isochrone pairs), and process each of those cells using the (R)PHAST algorithm. We tried edge separators as well as vertex separators and discussed their advantages and drawbacks. Our current GS+PHAST implementations are not customizable, however, their queries outperform the multilevel Dijkstra algorithms for many time limits.

Since our main motivation for computing isochrones is to display a electric vehicle's remaining cursing range, we dedicated a chapter to isochrones for electric vehicles. We modeled electric energy consumption as cost functions of bounded descriptive complexity, as described by Eisner et al. [33]. This allowed us to adapt RangeDijkstra and the multilevel algorithms to the EV scenario.

We finished with an experimental evaluation of the algorithms described in this thesis. We found that, except RangeDijkstra, all algorithms are fast enough for practical applications. The speedups on multi-core machines were fairly good (although not perfect). We observed that computing isochrone pairs is somewhat more difficult, especially for the multilevel algorithms. They need to spend more time during either customization or queries. Finally, we showed that computing isochrones in the EV scenario does not take longer than in the standard scenario.

8.2 Future Work

Our current multilevel Dijkstra implementations achieve rather poor query times for computing isochrone pairs, due to our handling of weakly connected cells. For now, we set the eccentricity of a boundary vertex u to infinity whenever there is at least one vertex v in the cell such that no $u-v$ path exists within the cell. This approach causes almost no overhead during customization, but significantly slows down queries. Alternatively, we may not restrict searches during customization to cells, but allow them to cross cell boundaries. This approach leads to tighter eccentricities and thus faster queries. It is an interesting question how much it slows down customization.

The GS+PHAST algorithms do not distinguish between metric-independent preprocessing and metric-dependent customization. Future work includes extending them to support the three-phase workflow introduced with CRP. A key ingredient could be Customizable Contraction Hierarchies [25]. For preliminary experiments, we built a customizable implementation of PHAST, using a nested dissection order [6] to build the upward and downward graph. Whereas the number of edges in the downward graph doubled, query times increased only by a factor of one and a half. Since computing eccentricities takes only about a second on a 16-core server, customizable implementations may be possible at least for the Core-Dijkstra and Core-PHAST variant. It is probably more difficult to provide customizable implementations for the oracle variants, due to the expensive oracle computation.

It would also be interesting to adapt the GS+PHAST algorithms to compute isochrones for electric vehicles. There should be no fundamental problems, since we can augment the downward graph used by PHAST in the same way as the downward graph used by GRASP (with consumption cost functions). However, the linear sweeps may become significantly slower, since the amount of data accessed would quadruple (32 bits for storing the length of a (shortcut) edge plus three times 32 bits to represent its consumption cost function).

Our main motivation for computing isochrones is to display the region that is reachable within a certain amount of time or, in the context of electric vehicles, to display the remaining cruising range. Hence, we need to develop algorithms that transform isochrones into displayable isochrone areas or isolines.

Bibliography

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A hub-based labeling algorithm for shortest paths on road networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

[2] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The shortest path problem revisited: Optimal routing for electric vehicles. In Rüdiger Dillmann, Jürgen Beyerer, Uwe D. Hanebeck, and Tanja Schultz, editors, *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, volume 6359 of *Lecture Notes in Computer Science*, pages 309–316. Springer, September 2010.

[3] 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. Technical Report abs/1504.05140, ArXiv e-prints, 2015.

[4] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.

[5] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing speed-up techniques is hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[6] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. Search-space size in contraction hierarchies. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13)*, volume 7965 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2013.

[7] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[8] Veronika Bauer, Johann Gamper, Roberto Loperfido, Sylvia Profanter, Stefan Putzer, and Igor Timko. Computing isochrones in multi-modal, schedule-based transport networks. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08)*, GIS '08, pages 78:1–78:2. ACM Press, 2008.

[9] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-optimal routes for electric vehicles. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 54–63. ACM Press, 2013.

[10] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2 edition, 2001.

[12] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.

[13] 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.

[14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning. In Panos M. Pardalos and Steffen Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[15] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 2015.

[16] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. Graph partitioning with natural cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011.

[17] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Faster batched shortest paths in road networks. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 52–63, 2011.

[18] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Query scenarios for customizable route planning, April 2014. US Patent Application 13/649,114.

[19] Daniel Delling, Moritz Kobitzsch, and Renato F. Werneck. Customizing driving directions with GPUs. In *Proceedings of the 20th International Conference on Parallel Processing (Euro-Par 2014)*, volume 8632 of *Lecture Notes in Computer Science*, pages 728–739. Springer, 2014.

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

[21] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.

[22] Daniel Delling and Renato F. Werneck. Faster customization of road networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 2013.

[23] Daniel Delling and Renato F. Werneck. Customizable point-of-interest queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 27(3):686–698, March 2015.

[24] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[25] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*, volume 8504 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2014.

[26] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[27] Herbert Edelsbrunner, David G. Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Transactions on Information Theory*, 29(4):551–559, July 1983.

[28] Alexandros Efentakis, Nikos Grivas, George Lamprianidis, Georg Magenschab, and Dieter Pfoser. Isochrones, traffic and DEMOgraphics. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 548–551. ACM Press, 2013.

[29] Alexandros Efentakis and Dieter Pfoser. Optimizing landmark-based routing and preprocessing. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pages 25:25–25:30. ACM Press, November 2013.

[30] Alexandros Efentakis and Dieter Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, volume 8737 of *Lecture Notes in Computer Science*, pages 358–370. Springer, September 2014.

[31] Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. SALT. A unified framework for all shortest-path query variants on road networks. In Evripidis Bampis, editor, *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science*, pages 298–311. Springer, 2015.

[32] Alexandros Efentakis, Dimitris Theodorakis, and Dieter Pfoser. Crowdsourcing computing resources for shortest-path computation. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pages 434–437. ACM Press, 2012.

[33] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large network. In Wolfram Burgard and Dan Roth, editors, *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, August 2011.

[34] David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08)*, pages 1–10. ACM Press, 2008.

[35] Lester R. Ford and Delbert R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[36] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[37] Johann Gamper, Michael Böhlen, Willi Cometti, and Markus Innerebner. Defining isochrones in multimodal spatial networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2381–2384. ACM Press, 2011.

[38] Johann Gamper, Michael Böhlen, and Markus Innerebner. Scalable computation of isochrones with network expiration. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management*, SSDBM'12, pages 526–543. Springer, 2012.

[39] Robert Geisberger. Contraction hierarchies. Master's thesis, Universität Karlsruhe, 2008. http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf.

[40] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[41] Andrew V. Goldberg. A practical shortest path algorithm with linear expected time. *SIAM Journal on Computing*, 37:1637–1655, 2008.

[42] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[43] Andrew V. Goldberg and Renato F. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[44] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[45] Stefan Hausberger. *Simulation of Real World Vehicle Exhaust Emissions.* Habilitation, Technische Universität Graz, 2003.

[46] Michael K. Hidrue, George R. Parsons, Willett Kempton, and Meryl P. Gardner. Willingness to pay for electric vehicles and their attributes. *Resource and Energy Economics*, 33(3):686–705, 2011.

[47] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.

[48] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 156–170. SIAM, 2006.

[49] Markus Innerebner, Michael Böhlen, and Johann Gamper. Isoga: A system for geographical reachability analysis. In *Proceedings of the 12th International Conference on Web and Wireless Geographical Information Systems*, W2GIS'13, pages 180–189. Springer, 2013.

[50] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using highway hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

[51] Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 19–40. American Mathematical Society, 2009.

[52] Sarunas Marciuska and Johann Gamper. Determining objects within isochrones in spatial network databases. In *Proceedings of the 14th East European Conference on Advances in Databases and Information Systems*, ADBIS'10, pages 392–405. Springer, 2010.

[53] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[54] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11:430–452, 1990.

[55] Peter Sanders and Dominik Schultes. Engineering highway hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1):1–40, 2012.

[56] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 16–29. SIAM, 2012.

[57] Peter Sanders and Christian Schulz. Karlsruhe high quality partioning: User guide. Technical report, Karlsruhe Institute of Technology, 2015.

[58] Christian Schulz. *High Quality Graph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, July 2013.

[59] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4), 2014.

[60] Dorothea Wagner and Thomas Willhalm. Speed-up techniques for shortest-path computations. In *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07)*, volume 4393 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2007. Invited Talk.

[61] J.W.J. Williams. Algorithm 232: Heapsort. *Journal of the ACM*, 7(6):347–348, June 1964.

A Appendix

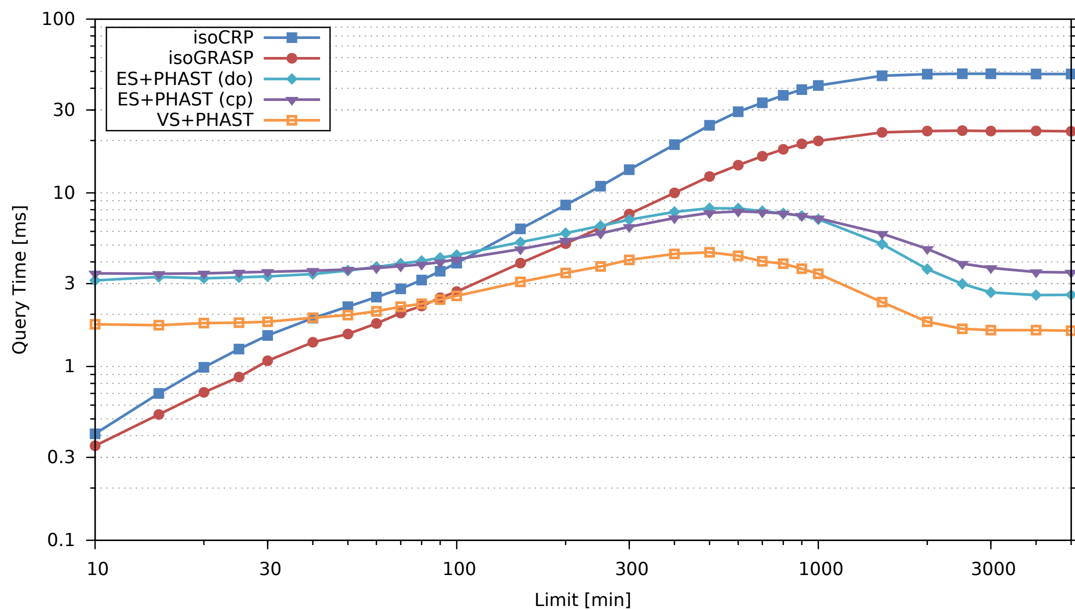


Figure A.1 – Parallel query times using 16 cores for varying time limits on DIMACS Europe when computing isochrone pairs.

Table A.1 – Sequential query times [ms] of ES+PHAST (do) for different partitions and varying choices of the parameter k. The variant with the least absolute (relative) deviations is highlighted in dark (light) gray.

core	limit	# CELLS							
size	[min]	128	256	512	1 024	2 048	4 096	8 192	16 384
128	10	15.97	11.33	9.48	9.35	10.39	12.38	15.00	16.83
	100	37.62	33.54	34.87	40.77	53.13	72.52	94.92	112.57
	500	140.42	133.28	140.29	155.25	189.91	241.25	299.12	326.39
256	10	15.85	11.23	9.47	9.22	10.27	12.26	14.84	16.65
	100	37.33	33.13	34.32	39.81	51.69	70.59	91.72	106.33
	500	139.09	131.61	137.43	151.08	184.15	233.27	286.93	312.07
512	10	15.58	11.04	9.26	9.00	10.03	11.96	14.55	16.18
	100	36.67	32.47	33.12	37.96	48.87	66.22	84.80	93.75
	500	136.55	128.51	132.35	143.09	172.89	217.03	265.15	267.09
1 024	10	15.17	10.64	8.84	8.52	9.52	11.46	13.87	15.45
	100	35.43	30.75	30.87	34.31	43.61	58.55	71.38	76.57
	500	131.32	121.49	121.96	127.33	151.58	188.10	215.39	207.94
2 048	10	14.50	9.90	8.18	7.76	8.81	10.67	13.08	14.78
	100	33.57	28.06	27.27	28.67	35.63	46.55	55.31	56.91
	500	123.73	108.70	104.88	102.18	118.74	143.01	159.06	143.81
4 096	10	14.40	9.54	7.55	7.16	8.01	9.87	12.33	14.31
	100	33.04	26.32	23.55	23.50	27.01	33.62	37.19	41.61
	500	120.81	100.43	87.86	80.10	84.91	95.58	96.31	96.85
8 192	10	14.56	9.72	7.57	**7.09**	7.79	9.57	12.07	14.13
	100	33.11	26.10	22.51	21.25	22.37	25.32	27.93	31.40
	500	120.50	98.49	82.16	70.25	66.60	67.44	66.67	67.39
16 384	10	15.42	9.98	8.00	7.49	8.16	9.89	12.26	14.28
	100	34.89	26.29	22.66	20.82	**20.75**	22.02	23.51	25.68
	500	127.28	98.53	81.40	67.12	59.22	54.72	51.04	50.39
32 768	10	15.67	10.59	8.54	8.26	9.17	10.86	13.03	14.88
	100	35.14	27.68	23.03	21.23	20.97	21.56	22.27	23.68
	500	127.47	103.56	81.31	66.80	57.65	50.79	45.07	**42.98**

Table A.2 – Sequential query times [ms] of VS+PHAST for different partitions and varying choices of the parameter k. The variant with the least absolute (relative) deviations is highlighted in dark (light) gray.

core size	limit [min]	# CELLS 128	256	512	1 024	2 048	4 096	8 192	16 384
128	10	10.52	6.10	3.88	2.71	2.25	2.23	2.68	3.33
	100	27.40	22.23	20.58	20.98	24.68	33.94	54.24	82.83
	500	104.50	93.02	88.56	86.42	96.13	121.41	185.43	268.22
256	10	10.43	6.01	3.80	2.61	2.15	2.08	2.48	3.07
	100	27.14	21.79	19.98	19.95	23.27	31.20	48.89	74.96
	500	103.28	90.85	85.54	81.68	89.64	110.38	166.40	240.61
512	10	10.28	5.85	3.63	2.45	1.98	1.83	2.11	2.63
	100	26.65	21.10	18.86	18.38	20.73	26.52	40.04	61.94
	500	101.13	87.51	80.00	74.19	78.21	91.74	132.30	193.11
1 024	10	10.12	5.65	3.45	2.25	1.78	1.56	1.69	2.03
	100	26.02	20.13	17.49	16.22	17.64	20.91	29.05	43.76
	500	98.36	82.92	73.16	64.42	64.86	69.30	92.21	129.08
2 048	10	9.97	5.52	3.33	2.16	1.64	1.40	1.42	1.55
	100	25.47	19.34	16.30	14.65	14.73	16.22	20.46	26.81
	500	95.99	79.26	67.71	57.36	53.82	53.46	63.43	77.58
4 096	10	9.93	5.51	3.31	2.16	1.63	1.37	**1.34**	1.40
	100	25.05	18.82	15.50	13.51	12.92	13.23	15.37	18.14
	500	94.35	76.95	64.06	52.82	46.87	43.71	47.09	52.46
8 192	10	10.03	5.63	3.45	2.29	1.76	1.48	1.44	1.45
	100	24.91	18.68	15.10	12.86	11.81	11.33	12.16	13.00
	500	93.20	75.61	61.84	49.97	42.72	37.48	37.66	37.52
16 384	10	10.31	5.93	3.74	2.59	2.07	1.78	1.74	1.72
	100	25.02	18.70	15.07	12.67	11.44	10.55	10.86	10.79
	500	92.68	74.87	60.74	48.51	40.60	34.32	33.19	30.46
32 768	10	10.87	6.50	4.30	3.16	2.62	2.34	2.30	2.29
	100	25.43	19.10	15.44	13.01	11.63	10.54	10.66	**10.14**
	500	92.44	74.55	60.37	47.93	39.72	32.99	31.20	**27.50**

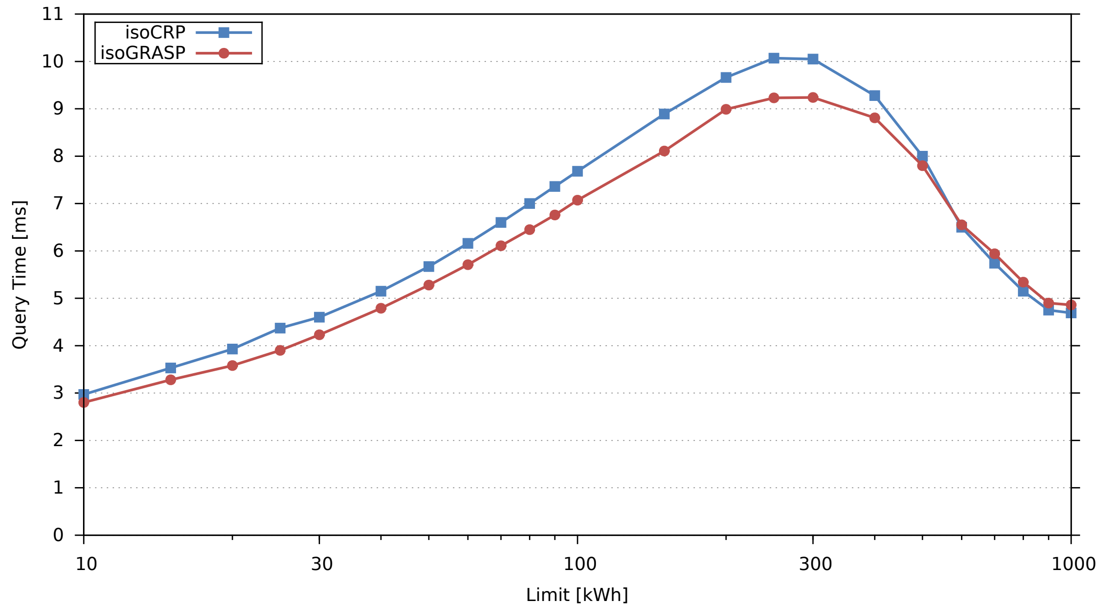


Figure A.2 – Parallel query times using 16 cores for varying initial charge levels on PTV Europe when computing isochrone edges.