Advancing Urban Mobility with Algorithm Engineering

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Valentin Buchhold

Tag der mündlichen Prüfung:	14. Juli 2021
Erste Referentin:	Prof. Dr. Dorothea Wagner
Zweite Referentin:	Prof. Dr. Sabine Storandt

Acknowledgments

Since I took a practical course on route planning in the first semester of my master's program, I have been fascinated by maps and navigation. I thank Dorothea Wagner for the opportunity to work in this area as a doctoral researcher after my studies. I had the chance to contribute to a number of interesting EU and industry projects, but still had enough freedom and leeway to move this thesis forward. I am also grateful to Peter Sanders for interesting discussions and his valuable input, and to Sabine Storandt for readily accepting to referee this thesis.

Moreover, I thank Daniel Delling for giving me the opportunity to spend a summer at Apple. I enjoyed not only an inspiring internship, but also numerous trips into the breathtaking nature of California, Nevada, and Arizona.

I also thank all of my colleagues at the Institute of Theoretical Informatics. It was a good time! In particular, I am grateful to my office mate Tobias Zündorf for many long nights of computer science, healthy food orders, illuminating discussions about the important things in life, and world-class foosball matches against our favorite opponents Guido Brückner and Jonas Sauer. I thank Lukas Barth for challenging my political beliefs and Jonas Sauer for granting asylum in his office to write these lines.

Last but not least, I would like to thank my family and friends for providing emotional and practical support when I needed it.

Deutsche Zusammenfassung

Die fortschreitende Urbanisierung ist ein globales Phänomen. Schon heute resultieren daraus diverse Probleme, wie etwa Staus, Unfälle, ein Mangel an Parkplätzen oder die Emission von Feinstaub, Stickoxiden und CO_2. Bis 2050 prognostiziert die OECD eine Verdreifachung des Personenverkehrs. In den kommenden Jahren und Jahrzehnten werden deshalb die durch urbane Mobilität verursachten Probleme weiter zunehmen. Gleichzeitig besteht die Aussicht, dass innovative Entwicklungen wie etwa Elektrifizierung, Automatisierung, Vernetzung und Informationstechnologie im Allgemeinen bei der Bewältigung dieser Herausforderungen behilflich sein werden. In dieser Dissertation entwerfen, implementieren und evaluieren wir praxisnahe Algorithmen für diverse Probleme aus dem Bereich der urbanen Mobilität.

Für die moderne Entwicklung von praxisnahen Algorithmen (*Algorithm Engineering*) sind realistische Experimente wesentlich. Die Verfügbarkeit von Benchmark-Daten ist deshalb eine notwendige Voraussetzung. Für Algorithmen aus dem Bereich der urbanen Mobilität enthält die Eingabe typischerweise ein Straßennetz und Nachfragedaten. Weltweite Daten zu Straßen und Gebäuden werden inzwischen von OpenStreetMap zur Verfügung gestellt. Frei verfügbare Nachfragedaten existieren jedoch kaum. Im ersten Themenbereich dieser Dissertation entwerfen wir deshalb skalierbare Algorithmen zur Generierung realistischer Nachfragedaten. Die theoretische Grundlage bildet das vor kurzem vorgeschlagene Radiation-Modell. Wir stellen mehrere Algorithmen zur Nachfrageberechnung gemäß dieses Modells vor, die eine zunehmend bessere Lösungsqualität und Rechenzeit aufweisen. Unser schnellster Algorithmus ist 100 000-mal schneller als eine naive Anwendung des Modells.

Die zeitaufwendigste Aufgabe bei unseren Algorithmen zur Nachfrageberechnung besteht darin, zu einem gegebenen Startpunkt einer Fahrt den nächsten Endpunkt zu bestimmen, der spezielle Eigenschaften erfüllt. Dies ist eine Verallgemeinerung des Problems der nächsten Nachbarn in Straßennetzen, bei dem der nächste Endpunkt zu einem gegebenen Startpunkt bestimmt wird. Zunächst stellen wir für das Problem der nächsten Nachbarn einen neuen Algorithmus vor, den wir in einem zweiten Schritt auf die Nachfrageberechnung anwenden. In unseren Experimenten verhält sich der neue Algorithmus in etwa genauso gut wie die besten existierenden Techniken für die Bestimmung nächster Nachbarn in Straßennetzen. Für die Nachfrageberechnung führt er zu einer beträchtlichen Beschleunigung.

Im zweiten Themenbereich dieser Dissertation beschäftigen wir uns mit Verkehrsumlegungen. Dabei ist die Aufgabe, für ein gegebenes Straßennetz und gegebene Nachfragedaten die Auslastung jeder Strecke im Netz zu berechnen. Verkehrsumlegungen sind seit Jahrzehnten ein allgegenwärtiges Werkzeug für verkehrsplanerische Untersuchungen. Dem Einsatz zum intelligenten Echtzeit-Verkehrsmanagement standen bisher relativ langsame Rechenzeiten im Wege. In dieser Dissertation entwerfen wir deshalb einen Algorithmus, der eine Verkehrsumlegung für eine Metropolregion mit mehreren Millionen Einwohnern innerhalb von wenigen Sekunden berechnen kann. Dies ist fast 40-mal schneller als der bisher beste Algorithmus und macht Verkehrsumlegungen zu einem Echtzeit-Werkzeug.

Mitfahrgelegenheiten, wie sie von Unternehmen wie Uber und Lyft angeboten werden, bilden den dritten Themenbereich dieser Dissertation. Gegeben ist dabei eine Flotte von Fahrzeugen und eine Menge von Fahrtanfragen, die den Fahrzeugen möglichst intelligent zugewiesen werden sollen, wobei Fahrten von mehreren Fahrgästen geteilt werden können. Für dieses Problem stellen wir einen neuartigen Algorithmus vor, der nicht nur 30-mal schneller als bisherige Verfahren ist, sondern im Gegensatz zu diesen auch beweisbar optimale Lösungen findet. Schnelle Rechenzeiten sind insbesondere für Verkehrssimulationen von großer Bedeutung, bei denen ein Szenario vielfach mit unterschiedlichen Modellparametern simuliert wird. Die Rechenzeit solcher Simulationen kann mit unserem Algorithmus von mehreren Tagen auf wenige Stunden verringert werden.

Das Herzstück der Algorithmen aus den drei zuvor genannten Themenbereichen bildet die Kürzeste-Wege-Technik Customizable Contraction Hierarchies (CCHs). Wie fast ausnahmslos alle Algorithmen zur Routenplanung wurde sie unter der vereinfachenden Annahme entworfen, dass Abbiegekosten und Abbiegeverbote außer Acht gelassen werden. Während Abbiegekosten für Fernreisende auf Autobahnen vernachlässigbar sein mögen, sind sie für innerstädtische Routen von großer Bedeutung. Deshalb untersuchen wir in dieser Dissertation, wie Abbiegekosten möglichst effizient in CCHs integriert werden können. Während sich eine naive Integration in 10-mal langsameren Rechenzeiten niederschlägt, kann unsere beste Variante dies auf einen in der Praxis akzeptablen Faktor 3 verringern.

Algorithmen zur Routenplanung führen typischerweise eine Vorberechnung durch, um nachfolgende Kürzeste-Wege-Anfragen zu beschleunigen. Alle anpassbaren (engl.: *customizable*) Techniken, zu denen auch der CCH-Algorithmus gehört, partitionieren das Straßennetz als ersten Vorberechnungsschritt. Zur Partitionierung eines statischen Straßennetzes existieren in der Literatur eine Vielzahl von Algorithmen. In der Praxis ändern sich Straßendaten jedoch überraschend häufig (in OpenStreetMap gibt es mehrere Millionen Änderungen pro Tag). Zum Abschluss dieser Dissertation entwerfen wir deshalb einen Algorithmus, der zur Partitionierung eines Straßennetzes die Partition einer älteren Version desselben Netzes ausnutzen kann. Dies verringert nicht nur die Rechenzeit, sondern auch die Unterschiede zwischen den beiden Partitionen, was für viele Anwendungen von Nutzen ist.

Contents

1 Introduction

Advancing urbanization is a global phenomenon. Already today, this results in various problems, such as traffic jams, accidents, a lack of sufficient parking space, or the emission of particulate matter, nitrogen oxides, and CO_2. Moreover, the Organization for Economic Co-operation and Development (OECD) predicts that urban traffic will triple by 2050 [ITF19]. The problems caused by urban mobility will therefore continue to increase in the coming years and decades. At the same time, there is a reasonable prospect that innovations in areas such as electrification, automation, connectivity, and information technology in general will help to cope with these challenges. In this thesis, we design, analyze, implement, and experimentally evaluate practical algorithms for various problems in the area of urban mobility, including the computation of mobility flows, traffic assignment, and dynamic ridesharing.

1.1 Algorithm Engineering

Our focus is on algorithms that are easy to implement and have good performance in practice. Therefore, we guide and justify our design choices by experiments rather than asymptotic worst-case analyses. While we are not so much interested in provable bounds on running time, we are aiming for provably optimal solutions (at least for problems that are polynomial-time solvable).

Classic *algorithm theory* focuses on designing algorithms with simple machine and problem models in mind, and giving theoretical performance guarantees by asymptotic worst-case analyses. While this methodology frequently results in el-

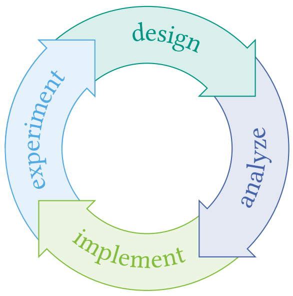


Figure 1.1: Visualization of the algorithm engineering methodology.

egant and timeless algorithms, these algorithms are often difficult to understand and implement. Moreover, worst-case analyses may not be very informative as to whether an algorithm works well in practice, since they ignore constant factors and lower-order terms, and the worst case may not arise in practice.

The more recent *algorithm engineering* methodology [San09, MS10, SW11] complements classic algorithm theory by experimental studies. To be able to conduct experiments, we first need to implement the algorithm, taking into account the features of modern computer architectures, such as cache memory and parallelism on different levels. Experiments give more insight into problem and algorithm, which helps us to design a revised version of the algorithm. Therefore, algorithm engineering can be seen as repeating cycles of design, analysis, implementation, and experimental evaluation (see Figure 1.1 for an illustration).

We employ the algorithm engineering methodology throughout the whole thesis. The iterative process is particularly pronounced in Chapters 3 and 4, in which we develop practical algorithms for generating travel demand data in a road network based on the recently introduced radiation model.

We start by designing a straightforward yet carefully tuned algorithm. Our analysis shows that its running time is $O(M^2 \log M)$, where M is a parameter that measures the size of the network. The superquadratic running time suggests that the straightforward algorithm is useful only for small input sizes. This is confirmed by our experiments, which show that the algorithm takes months on a continental network.

To come up with a more practical algorithm, we use a completely different approach during the second design phase. The analysis leads to an $O(M \log M \log\log M)$ bound. Since the output grows linearly with the size of the network, this bound comes close to the simple lower bound of $\Omega(M)$, and thus our new algorithm is almost optimal. Our experiments show that the algorithm can indeed handle networks of continental size, but still takes up to a few hours on them.

To decrease the running time even further, we choose to replace shortest-path distances by geometric distances, which are much easier to compute. This simplification allows us to come up with a more sophisticated algorithm during the third design phase. Our experimental results indeed show a big decrease in running time. Compared to the straightforward and near-optimal algorithm, the geometric algorithm takes only seconds rather than months and hours, respectively. However, the experiments also show a decrease in solution quality, and thus cannot fully justify the use of geometric rather than shortest-path distances.

Although the geometric algorithm yields only lower-quality solutions, its algorithmic heart proves to be fast. Therefore, we combine the main ideas from the geometric algorithm with shortest-path distances during the fourth design phase. Our experiments show that the new algorithm obtains high-quality solutions like the straightforward and near-optimal algorithm, but at much lower cost.

The algorithm engineering methodology is not as pronounced in all chapters as in the chapters on travel demand generation. In particular, performance guarantees are difficult to achieve for most of our algorithms. Where applicable, we still give theoretical guarantees for subroutines that are easier to analyze.

1.2 Main Contributions

This section gives a brief overview of the contributions of this thesis. Each of the following subject areas corresponds to a chapter of this thesis, in which the subject area is studied in detail, including extensive and thorough experimental studies.

Travel Demand Generation. Realistic experiments are essential for the algorithm engineering methodology. The availability of benchmark data is therefore a necessary requirement. Algorithms in the area of urban mobility typically take as input a road network and travel demand data. OpenStreetMap now provides data about roads and buildings all over the world. However, there is hardly any publicly available demand data. In the first subject area of this thesis, we thus design scalable algorithms for generating realistic demand data. The recently proposed radiation model serves as the theoretical foundation. We present multiple algorithms for travel demand generation according to this model, which obtain increasingly better solution quality and running times. Our fastest algorithm is 100 000 times faster than a straightforward application of the model, making it practical for continental networks.

Nearest-Neighbor Queries for Demand Generation. The most time-consuming task executed in our algorithms for travel demand generation is as follows. Given a starting point of a trip, we need to find the closest destination that satisfies certain properties. This is a generalization of the nearest-neighbor problem in road networks, which asks for the destination closest to a given starting point. We first present a novel algorithm for finding nearest neighbors, which we then, in a second step, apply to travel demand generation. In our experiments, the novel algorithm is on a par with state-of-the-art algorithms for finding nearest neighbors in road networks. For travel demand generation, it achieves significant speedups.

Traffic Assignment. In the second subject area of this thesis, we study traffic assignments. Given a road network and demand data, the goal is to compute the traffic flow on each road segment in the network. Traffic assignments have been a ubiquitous tool used in traffic engineering analyses for decades. So far, their use for intelligent real-time traffic management has been prevented by relatively slow running times. In this thesis, we therefore design an algorithm that can compute a traffic flow pattern on a metropolitan area with several million inhabitants in a few seconds. This is almost 40 times faster than the current state of the art and makes traffic assignments a real-time tool in the future.

Dynamic Ridesharing. Ridesharing services such as Uber and Lyft form the third subject area of this thesis. Here we are given a fleet of vehicles and a set of ride requests that are to be assigned to the vehicles as intelligently as possible, exploiting the fact that rides can be shared by multiple riders. We present a novel algorithm for this problem, which not only is 30 times faster than existing algorithms, but also finds provably optimal solutions (which is not the case for most existing algorithms). Fast running times are of particular importance for transport simulations, where a scenario is simulated many times with varying model parameters. Our algorithm can decrease the running time of such simulations from multiple days to a few hours.

Turn Costs and Restrictions. At the heart of the algorithms from the three subject areas mentioned above is the shortest-path technique customizable contraction hierarchies (CCHs). Like almost all algorithms for route planning, it was developed under the simplifying assumption that turn costs and restrictions are ignored. While turn costs may be negligible for long-distance travelers on highways, they are of utmost importance for inner-city routes. In this thesis, we therefore study how to incorporate turn costs into CCHs as efficiently as possible. While a naive integration results in a slowdown of an order of magnitude, our best variant can reduce this to a factor of 3, which is reasonable in practice.

Partitioning Evolving Road Networks. Algorithms for route planning typically use special preprocessing to accelerate shortest-path queries. All customizable techniques, including the CCH algorithm, partition the road network as a first preprocessing step. There is a large number of algorithms for partitioning static road networks in the literature. In practice, however, road network data changes surprisingly frequently (there are several million changes to OpenStreetMap each day). Therefore, we conclude this thesis by designing an algorithm for partitioning an evolving road network that exploits the partition of a previous network snapshot. This decreases not only the running time, but also the difference between the two partitions, which is useful for many applications in practice.

2 Fundamentals

We aim to make each chapter of this thesis as self-contained as possible while still avoiding repetition. Therefore, we prefer to describe fundamentals and related work in the chapter in which they are needed, as long as they are limited to a single chapter. Here we introduce some basic techniques that will play an important role throughout the whole thesis. This includes Dijkstra's shortest-path algorithm and the speedup technique contraction hierarchies (CHs), including its variants customizable contraction hierarchies (CCHs) and contraction hierarchies with buckets (BCHs).

In the simplest case, we treat a road network as a directed graph $G = (V, E)$ where each vertex represents an intersection and each edge represents a road segment. Each edge $(v, w) \in E$ has a nonnegative length $\ell(v, w)$ that represents the travel time from the tail v to the head w. The shortest-path distance (i.e., travel time) from v to w is denoted by $dist(v, w)$. For simplicity, we often assume that G is strongly connected.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] computes the shortest-path distances from a source s to all other vertices. For each vertex v, it maintains a *distance label* $d_s(v)$, representing the length of the shortest s–v path seen so far. Moreover, it maintains an addressable priority queue Q [SMDD19] of vertices, using their distance labels as keys. Initially, $d_s(s) = 0$ for the source s, $d_s(v) = \infty$ for each vertex $v \neq s$, and $Q = \{s\}$.

The algorithm repeatedly extracts a vertex v with minimum distance label from the queue and *settles* it by *relaxing* its outgoing edges (v, w). To relax an edge $e = (v, w)$, the path from s to w via v is compared with the shortest path from s to w found so far. More precisely, if $d_s(v) + \ell(e) < d_s(w)$, the algorithm sets $d_s(w) = d_s(v) + \ell(e)$ and inserts w into the queue. It stops when the queue becomes empty. Note that Dijkstra's algorithm has the label-setting property, i.e., each vertex is settled at most once. Therefore, when computing a point-to-point shortest path from a source s to a target t, we can stop the search when t is settled.

2.2 Contraction Hierarchies

Although Dijkstra's algorithm runs in almost linear time, it still takes a few seconds on continental road networks. Therefore, speedup techniques have been developed that rely on a slow preprocessing phase to enable fast queries. *Contraction hierarchies* (CHs) [GSSV12] are a two-phase speedup technique to accelerate point-to-point computations, which exploits the inherent hierarchy of road networks (their organization into residential roads, urban roads, highways, motorways, etc.). To differentiate them from customizable CHs, we sometimes call them *weighted* or *standard* CHs.

Preprocessing. The preprocessing phase heuristically orders the vertices by importance, and *contracts* them from least to most important. Intuitively, vertices that hit many shortest paths are considered more important, such as vertices on highways and other main roads. To contract a vertex v, it is temporarily removed from the graph, and *shortcut* edges are added between its neighbors to preserve distances in the remaining graph (without v). Note that a shortcut is only needed if it represents the only shortest path between its endpoints, which can be checked by running a *witness search* (local Dijkstra) between its endpoints. The output of preprocessing is the input graph plus the shortcuts added during contraction. We call this graph H.

Queries. The query phase performs a bidirectional Dijkstra search on H that only relaxes edges leading to vertices of higher *ranks* (importance). More precisely, let a *forward CH search* be a Dijkstra search that relaxes only outgoing upward edges, and a *reverse CH search* one that relaxes only incoming downward edges. A *CH query* runs a forward CH search from the source and a reverse CH search from the target until the search frontiers meet. The stall-on-demand [GSSV12] optimization prunes the search at any vertex v with a suboptimal distance label.

Levels. Sometimes it is helpful to assign *levels* to vertices. If a vertex v has no lower-ranked neighbors, then v is assigned a level of zero. Otherwise, v is assigned a level of $L+1$, where L is the level of the highest-ranked of v's lower-ranked neighbors. Levels can be computed as we contract the vertices during preprocessing.

2.3 Customizable Contraction Hierarchies

Customizable contraction hierarchies (CCHs) [DSW16] are a three-phase technique that splits CH preprocessing into a metric-independent part, taking only the network structure into account, and a metric-dependent part (the *customization*), incorporating edge weights (the *metric*). A fast and lightweight customization is a key requirement for important features such as real-time traffic updates and personalized metrics.

CCHs require the road network to be *bidirected*, i.e., for each edge (v, w), the reverse edge (w, v) must also be present. This restriction is not as severe as it may seem at first sight, since a one-way road segment from v to w can be modeled by setting $\ell(w, v) = \infty$. Before describing the three phases of CCHs, we must be familiar with the concepts of separator decompositions and nested dissection orders.

Separator Decompositions. A *separator decomposition* [BCRW16] of a strongly connected n-vertex bidirected graph $G = (V, E)$ is a rooted tree $\mathcal{T} = (\mathcal{X}, \mathcal{E})$ whose nodes $X \in \mathcal{X}$ are disjoint subsets of V and that is recursively defined as follows. If $n = 1$, then $\mathcal{T}$ consists of a single node $X = V$. If $n > 1$, then $\mathcal{T}$ consists of a root $X \subseteq V$ that separates G into multiple strongly connected subgraphs $G_0, \dots, G_{d-1}$. The children of X are the roots of separator decompositions of $G_0, \dots, G_{d-1}$. For clarity, an element $v \in V$ is always called *vertex* and an element $X \in \mathcal{X}$ is always called *node*. We denote by $\mathcal{T}_X$ the subtree of $\mathcal{T}$ rooted at X and we denote by G_X the subgraph of G induced by the vertices contained in $\mathcal{T}_X$. The vertex set of G_X is represented by $V(G_X)$, and the edge set by $E(G_X)$.

In general, a separator X should be small, and the resulting subgraphs $G_0, \dots, G_d$ should be balanced. Therefore, separator decompositions are typically obtained by recursive dissection. Modern graph dissection algorithms tailored to road networks include Inertial Flow [SS15], FlowCutter [HS18], and InertialFlowCutter [GHUW19].

Nested Dissection Orders. A separator decomposition $\mathcal{T}$ of G induces a (not necessarily unique) *nested dissection order* π on the vertices in G [Geo73]. To obtain one, we number the vertices in the order in which they are visited by a postorder tree walk of $\mathcal{T}$, where the vertices in each node are visited in any order. Note that the resulting order $\pi = \langle \pi_0, \dots, \pi_{d-1}, \pi_d \rangle$ is split into $d + 1$ contiguous subsequences π_i, where d is the number of children Y_j of the root X of $\mathcal{T}$. The subsequences $\pi_0, \dots, \pi_{d-1}$ are nested dissection orders on $V(G_{Y_0}), \dots, V(G_{Y_{d-1}})$ and π_d is an arbitrary vertex order on X. We denote by $\pi^{-1}(v)$ the *rank* of v in π.

Preprocessing. The metric-independent preprocessing phase computes a separator decomposition of G, determines an associated nested dissection order on the vertices in G, and contracts them in this order. To contract a vertex v, it is temporarily removed, and shortcut edges are added between its neighbors. In contrast to standard contraction hierarchies, we run no witness searches (which depend on the metric) but add every potential shortcut. Again, the output of preprocessing is the input graph plus the shortcuts added during contraction, and we call this graph H. We denote by $N_H^{\uparrow}(v)$ the set of neighbors of v in H ranked higher than v.

Customization. The customization phase computes the lengths of the edges in H by processing them in bottom-up fashion. To process an edge (u, w), it enumerates all triangles $\{v, u, w\}$ in H where v has lower rank than u and w, and checks whether the path $\langle u, v, w \rangle$ improves the length of (u, w).

Queries. There are two query algorithms. First, one can run a standard CH query without modification. In addition, there is a query algorithm based on the *elimination tree* of H. The parent of a vertex v in the elimination tree is the lowest-ranked vertex in $N_H^{\uparrow}(v)$. Bauer et al. [BCRW16] prove that the ancestors of a vertex v in the elimination tree are exactly the set of vertices scanned by a Dijkstra-based CCH search from v. An elimination tree search from v therefore scans all vertices in the CCH search space of v in order of increasing rank by traversing the path in the elimination tree from v to the root. Since elimination tree queries use no priority queues, they are usually faster than Dijkstra-based CCH queries.

2.4 Contraction Hierarchies with Buckets

The *bucket-based approach* by Knopp et al. [Kno+07] extends any hierarchical speedup technique such as CHs and CCHs to batched shortest paths. In the one-to-many shortest-path problem, the goal is to compute shortest paths from a source $s \in V$ to each target $t \in T \subseteq V$. A bucket-based CH (BCH) search maintains a tentative distance $D_s(t)$ from s to each t, initialized to ∞, and for each vertex h an initially empty bucket $B(h)$. First, the algorithm runs a reverse CH search from each t and inserts, for each vertex h settled, an entry $(t, d_t(h))$ into $B(h)$. Note that $(t, d_t(h))$ can be thought of as a shortcut from h to t with length $d_t(h)$. Then, the algorithm runs a forward CH search from s and loops, for each vertex h settled, over all entries $(t, d_t(h)) \in B(h)$. If $d_s(h) + d_t(h) < D_s(t)$, it sets $D_s(t) = d_s(h) + d_t(h)$. Many-to-one queries from each source $s \in S \subseteq V$ to a target $t \in V$ work analogously. In this case, each bucket $B(h)$ stores entries that represent shortcuts from several s to h.

3 Travel Demand Generation

Determining travel demand within a region of interest takes a considerable calibration effort, requiring transportation surveys, traffic counts, and empirical trip volumes. However, there is a need for demand calculation without substantial calibration, for example to generate large-scale benchmark data for evaluating transportation algorithms. In this chapter, we present several approaches for demand calculation that take as input only publicly available data, such as population and POI densities. Our algorithms build upon the recently proposed radiation model, which is inspired by job search models in economics. We show that a straightforward implementation of the radiation model does not scale to continental road networks, taking months even on a modern 16-core server. Therefore, we introduce more scalable implementations, substantially decreasing the running time by five orders of magnitude from months to seconds. An extensive experimental evaluation shows that the output of our algorithms is in accordance with demand data used in production systems. Compared to simple approaches previously used in algorithmic publications to generate benchmark data, our algorithms output demand data of better quality, take less time, and have similar implementation complexity.

Chapter is based on joint work with Peter Sanders and Dorothea Wagner [BSW19a].

3.1 Introduction

Determining travel demand within a region of interest is no automatic process, but requires calibration by transportation experts based on transportation surveys, traffic counts, and empirical trip volumes. Usually, such cost intensive and time-consuming travel forecasts are made for urban and transport planning purposes by counties, municipalities, or transport authorities, which do not make their data publicly available. On the other hand, there is a need for demand calculation with low calibration effort that does not require transportation surveys or traffic counts. Our main motivation is the generation of large-scale benchmark data for evaluating transportation algorithms. Another application is the prediction of human mobility flows in developing nations, where empirical data and measurements are not available. Such regions benefit even from slightly less accurate demand data, which supports them in transport planning and epidemic modeling [Col+07, Ves12, Tiz+14].

Our goal is to develop algorithms for demand calculation that require as input only a road network (modeled as a directed graph) and a population grid covering the region of interest. Both kinds of data are publicly available for large parts of the world. OpenStreetMap maintains data about roads all over the world. Population grids are often produced and freely published along with census data, at least in industrial nations. For example, Eurostat, the statistical office of the European Union, provides a population grid with a resolution of 1 km for all EU and EFTA member states[1]. In addition, some countries like Germany[2] and Switzerland[3] publish countrywide population grids with a higher resolution of 100 m. There are also projects (e.g., Global Human Settlement Layer[4]) trying to provide a population grid for the whole world, by combining census data with fine-scale satellite imagery.

Human mobility models have received considerable attention recently; see e.g. [Bar+18] for an overview. The three prevailing models, however, have been the gravity model [Zip46], intervening opportunities model [Sto40, Sch59], and radiation model [SGMB12]. Considering a region of interest divided into zones, each model provides a closed formula for the mobility flow T_{ij} from zone i to zone j, depending on the population of i and j, and the spatial relation between them. A big advantage of the radiation model compared to the two other mobility models is the absence of parameters to be calibrated. Therefore, we take it as the foundation for our algorithms.

Our Contribution. Generally, human mobility models like the gravity and radiation model are applied at an aggregated level [Bar+18]. That is, the region of interest is divided into several zones (e.g., counties or municipalities), and the models are used for predicting the mobility flows between all pairs of zones. In this chapter, we apply and evaluate the radiation model at the level of individuals. The output of our algorithms is a set of origin-destination (OD) pairs where each pair represents a trip taken by an individual between two specific locations (i.e., vertices in the road network). Such microscopic demand data is necessary to evaluate a large number of transportation algorithms, including methods for ridesharing [Gei+10, DL13] and

[1] https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/population-distribution-demography/geostat

[2] https://www.zensus2011.de/DE/Home/Aktuelles/DemografischeGrunddaten.html

[3] https://www.bfs.admin.ch/bfs/de/home/dienstleistungen/geostat/geodaten-bundesstatistik/gebaeude-wohnungen-haushalte-personen/bevoelkerung-haushalte-ab-2010.html

[4] https://ghsl.jrc.ec.europa.eu/datasets.php

autonomous vehicle dispatching [BM16]. We show that a variant of the radiation model known as *radiation model with selection* [SMN13] can be used for generating trips that are in accordance with data used in production systems.

On a macroscopic level, the mobility flows between all pairs of zones can be computed by evaluating the closed formula provided by the radiation model for each pair of zones. In its original publication [SGMB12], the radiation model is applied to the United States at the level of counties. Since there are 3141 counties, computing mobility flows between all pairs needs about 10 million evaluations of the formula, which is quite feasible on a modern machine. However, when switching to the microscopic level, we are confronted with road networks having tens or hundreds of millions of vertices [Bas+16]. Computing mobility flows between all pairs of vertices then requires trillions or quadrillions of evaluations. When executed sequentially, this would take a few years on a modern machine (see Section 3.6.3). Therefore, in this chapter, we present more scalable implementations of the radiation model, substantially decreasing the running time from minutes to milliseconds on our metropolitan instances, and from months to seconds on our continental instances. This makes applications of the radiation model to the largest metropolitan areas and even continental networks at the microscopic level practical.

Finally, we compare our new radiation-based algorithms with approaches previously used in algorithmic publications to generate benchmark data. We show that our algorithms output demand data of better quality (i.e., being in better accordance with demand data used in production systems), take less time, have similar implementation complexity, and require only publicly available data.

Outline. This chapter is organized as follows. Section 3.2 reviews the radiation model (with selection) and discusses previously used benchmark data. Section 3.3 describes two simple approaches to calculate demand data. Section 3.4 introduces our new implementations of the radiation model. Section 3.5 shows how to take advantage of multiple cores. Section 3.6 presents an experimental evaluation of our radiation-based algorithms. Section 3.7 concludes with final remarks.

3.2 Preliminaries

We treat a road network as a directed graph $G = (V, E)$ where vertices represent intersections and edges represent road segments. Each edge $(i, j) \in E$ has a nonnegative length $\ell(i, j)$ representing the travel time between i and j. The problem we consider is computing the number T_{ij} of trips between each pair (i, j) of vertices i, j.

Each vertex $i \in V$ has a nonnegative number m_i of inhabitants, and M denotes the total population in the graph. We denote by r_{ij} the shortest-path distance between

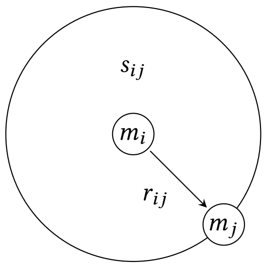


Figure 3.1: Input variables of human mobility models, such as the radiation model. Note that m_i denotes the population of vertex i, r_{ij} denotes the shortest-path distance between two vertices i and j, and s_{ij} denotes the population of all vertices in the shortest-path circle of radius r_{ij} centered on i.

two vertices i and j. Moreover, s_{ij} is the population of all vertices that are closer to i than j. In other words, s_{ij} is the population of all vertices in the shortest-path circle of radius r_{ij} centered on i (minus m_i and m_j). See Figure 3.1 for an illustration.

3.2.1 Radiation Model

The radiation model is inspired by job search models in economics [CMFB16, McC70, LM76]. There, a job searcher is assumed to consider job offers in increasing distance from his or her residence and stop when an offer is reached that provides working conditions (salary, insurance, office space, equipment, and so on) that fulfill the searcher's expectations. Since employers evaluate and value the searcher's skills differently, the working conditions obey a certain probability distribution. Similarly, the searcher's expectations also depend on his or her skills, and it is a natural assumption that they obey the same probability distribution (a searcher whose skills are highly valued by many prospective employers probably also has high expectations). The radiation model applies similar ideas to the choice of a traveler's destination.

Given a region of interest divided into several zones, the radiation model [SGMB12] assumes that each zone has a number of inhabitants and an amount of opportunities. In the simplest version, the number of opportunities is approximated by the population, i.e., there are M opportunities in a region with a population of M. The mobility flow out of each zone is proportional to its population. Destination selection is based on the following main idea: Each traveler assigns to all opportunities a fitness or attractiveness value, drawn independently from a common distribution. Then, the traveler selects the closest opportunity with a fitness higher than the traveler's fitness threshold, drawn from the same distribution. See Figure 3.2 for an illustration.

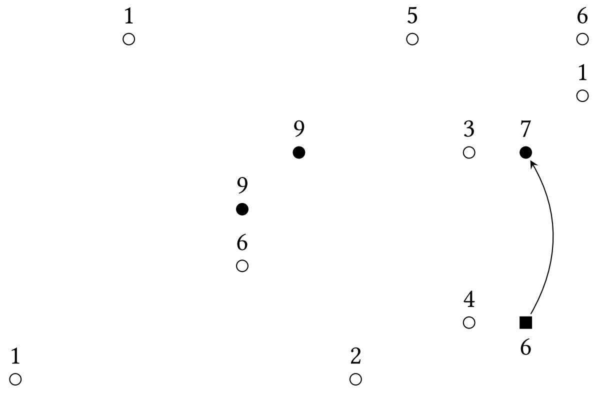


Figure 3.2: Destination selection in the radiation model. The traveler (square) assigns to all opportunities (circles) a fitness value. Among those opportunities with a fitness value higher than the traveler's fitness threshold (filled circles), the traveler selects the closest one as the destination of their trip.

Original Radiation Model. Under the assumptions mentioned above, Simini et al. [SGMB12] show that the mobility flow T_{ij} from zone i to zone j satisfies

$$T_{ij} = \underbrace{\gamma m_i}_{O_i} \underbrace{\frac{1}{1 - \frac{m_i}{M}}}_{c_i} \underbrace{\frac{m_i m_j}{(m_i + s_{ij})(m_i + m_j + s_{ij})}}_{p_{ij}} \tag{3.1}$$

Note that p_{ij} is the probability that a trip starting at i ends at j, under the assumption that there are infinitely many opportunities in the study area. It can be shown that the probability that a traveler starting at i does not find a sufficiently fit opportunity among the M closest opportunities is m_i/M [MSJB13]. Therefore, we normalize p_{ij} so that the probability that a travelers selects one of the M closest opportunities is 1, by dividing p_{ij} by $1 - m_i/M$. This is the purpose of the normalizing constant c_i. Intuitively, a traveler who failed the first time draws all fitness values and the fitness threshold again. Overall, the mobility flow T_{ij} from i to j is the product of the flow $O_i = \gamma m_i$ out of i (where γ is the proportionality constant) and the probability $c_i p_{ij}$ that a trip starting at i ends at j.

Radiation Model with Selection. While the radiation model obtains good results at the level of counties [SGMB12], it performs worse at finer levels [MSJB13]. To compensate for this drawback, Simini et al. [SMN13] propose to decrease the probability of selecting an opportunity by a factor of $1 - \lambda$. Intuitively, increasing λ increases the expected trip length. Then, the mobility flow T_{ij} satisfies

$$T_{ij} = \underbrace{\gamma m_i}_{O_i} \underbrace{\frac{1}{1 - \frac{1-\lambda^M}{1-\lambda^{m_i}} \frac{m_i}{M}}}_{c_i} \underbrace{\frac{\frac{1-\lambda^{m_i+s_{ij}}}{m_i+s_{ij}} - \frac{1-\lambda^{m_i+m_j+s_{ij}}}{m_i+m_j+s_{ij}}}{\frac{1-\lambda^{m_i}}{m_i}}}_{p_{ij}} \tag{3.2}$$

3.2.2 Previously Used Benchmark Data

Most experimental work [Bas+16, GH05, DGPW17, GSSV12, DSW16, ADGW11, ALS13, BFSS07] on algorithms for route planning in transportation networks has been evaluated on trips with the origin and the destination picked uniformly at random. Since a routing engine processes the trips (i.e., queries) independently from one another, this does not affect the validity of the evaluation. However, a meaningful evaluation of transportation algorithms that process the trips as a single unit, such as algorithms for traffic assignment, requires the trips to obey a realistic distribution.

One option is to consider real-world data sets. For example, Perederieieva et al. [PERW15] evaluate various traffic assignment algorithms on a set of standard benchmark instances. Those instances, however, have become outdated, i.e., their size does not reflect the size of current production data. Ten out of twenty instances have less than 1000 vertices, and even the largest instance consists of only 14 639 vertices. To compensate for this drawback, Schneck and Nökel [SN20] evaluate their traffic assignment algorithm on data taken from current production systems. The downside is that their data is proprietary and not publicly available.

Another option is to generate synthetic demand. For example, Luxen et al. [LS11] evaluate a traffic assignment algorithm on graphs representing the road networks of Belgium and Germany, drawing trip lengths from a geometric distribution.

3.3 Simple Approaches

In the past, several publications on different transportation algorithms resorted to very simplistic approaches to calculate travel demand data (cf. Section 3.2.2). This section describes our implementation of two simple ideas in detail.

3.3.1 Uniformly Distributed Endpoints

Arguably the simplest approach is to pick the origin and the destination uniformly at random. More precisely, we repeatedly choose a pair of uniform random vertices from the region of interest, compute the shortest-path distance between them, and stop as soon as the total network volume (i.e., the sum of the shortest-path distances computed so far) exceeds a certain threshold. For the shortest-path computations, we add the major roads in the surrounding region, to reduce boundary effects. This is fairly common when modeling traffic [HNA16]. We call this approach RAND.

Note that despite its simplicity, the RAND algorithm yields endpoints that tend to be in densely populated areas in the region of interest, since the population density correlates highly with the density of the road network.

Shortest paths can be computed with Dijkstra's algorithm [Dij59] or any other modern speed-up technique [Bas+16]. The method of choice depends on the scenario at hand. Generally, different speed-up techniques provide different trade-offs between preprocessing effort, space requirements, query time, and implementation complexity. On large road networks with high volumes, spending more time on preprocessing is justified. For example, contraction hierarchies [GSSV12] perform queries in about 100 microseconds with preprocessing times of a few minutes. Transit node routing [BFSS07, ALS13] and hub labeling [ADGW11] enable even faster queries, at the cost of increased preprocessing time and space. On the other hand, heavy preprocessing may not pay off on small road networks with low volumes, where the time to reach the total network volume with a lightweight shortest-path algorithm may be faster than a heavy preprocessing phase. In our experiments, we use bidirectional search to keep implementation complexity low.

3.3.2 Geometrically Distributed Distances

The RAND algorithm yields unrealistic distributions of the trip length. We can do better by drawing the trip length from a geometric distribution where the expected value μ is equal to the average trip length. More precisely, we generate one trip at a time. The origin vertex is picked uniformly at random from the region of interest. To choose the destination vertex, we draw the trip length from a geometric distribution with probability parameter $p = 1/(\mu + 1)$. We then run Dijkstra's algorithm from the origin until we settle a vertex whose distance label is greater than or equal to the trip length drawn before. That vertex is the destination we are looking for. When the priority queue becomes empty before we settle such a vertex, we draw the trip length again. We call this approach GEOM.

As in the RAND algorithm, we can add the major roads in the area surrounding the region of interest to the search graph, in order to reduce boundary effects. In this case, we stop the Dijkstra search as soon as we settle a vertex whose distance label is greater than or equal to the trip length drawn before *and* that vertex is contained in the region of interest. Again, the endpoints of the trips tend to be in densely populated areas, due to the correlation between the population and network density.

3.4 Radiation-Based Approaches

This section describes our main algorithmic contribution, several algorithms for demand calculation at the microscopic level that build upon the radiation model with selection discussed in Section 3.2. We start with a straightforward implementation of the model. Despite some careful engineering, it runs in superquadratic time, and thus does not scale to continental road networks with tens of millions of vertices. Afterwards, we present two output-sensitive, more scalable algorithms, decreasing the running time of the straightforward implementation on large road networks by five orders of magnitude, making the radiation model practical.

3.4.1 Straightforward Implementation

As described in Section 3.2, the radiation model defines the mobility flow T_{ij} between two zones i and j in the region of interest as the product of the flow O_i out of i and the probability $c_i p_{ij}$ that a trip starting at i ends at j. Hence, $T_{ij} = O_i c_i p_{ij}$; see also Equation (3.2). The most straightforward approach to calculate demand data based on the radiation model is to evaluate this formula for all pairs of zones. In our case, at the microscopic level, we view each vertex in the network as a zone of its own, and need to compute T_{ij} for all pairs of vertices.

There is one problem, however: we want the number of trips between any pair of vertices to be a natural number (including zero), but Equation (3.2) evaluates to a real number. At an aggregated level, we could simply round the mobility flow to the nearest integer. At the microscopic level, however, the mobility flow is almost always very close to zero. Rounding would probably result in no trips at all. Therefore, we actually view T_{ij} as a random variable that obeys a binomial probability distribution with O_i trials and success probability $c_i p_{ij}$. Note that the expected value of T_{ij} is actually $O_i c_i p_{ij}$, which is consistent with Equation (3.2).

The situation for the outflow O_i is analogous to the situation for T_{ij}. The radiation model defines O_i as γm_i, which may become too small at the microscopic level. Therefore, we actually view O_i as a random variable that obeys a binomial distribution with γM trials and success probability m_i/M. Again, we stress that the expected value of O_i is γm_i, which is consistent with Equation (3.2).

We now show how to implement the straightforward approach efficiently. In order to evaluate the formula for T_{ij} (more precisely, to draw T_{ij} accordingly), we need the population m_i of vertex i, the population m_j of vertex j, and the population s_{ij} of all vertices that are closer to i than j. Population counts are maintained as an array, indexed by vertex IDs, allowing efficient access to m_i and m_j. To obtain s_{ij}, we can run Dijkstra's shortest-path algorithm from i, stopping as soon as it scans j. Cumulating the population counts of all vertices scanned during the search yields s_{ij}.

Algorithm 3.1: Straightforward implementation of the radiation model, evaluating Equation (3.2) for all pairs of vertices.

```
Function $FRAD(G = (V, E), \gamma, \lambda)$
    foreach vertex $i \in V$ do
        if $m_i = 0$ then continue
        $O_i \leftarrow binomialVariate(\gamma M, m_i/M)$
        $s \leftarrow 0$
        $dijkstra.initialize(i)$
        $dijkstra.settleNextVertex()$
        while $dijkstra.queue \neq \emptyset$ do
            $j \leftarrow dijkstra.settleNextVertex()$
            if $m_j = 0$ then continue
            $s_{ij} \leftarrow s$
            $T_{ij} \leftarrow binomialVariate(O_i, c_i p_{ij})$
            for $k \leftarrow 1$ to $T_{ij}$ do
                output OD pair $(i, j)$
            $s \leftarrow s + m_j$
```

Note that our search graph again consists of the union of all roads in the region of interest and the major roads in the surrounding area.

Computing each s_{ij} from scratch is wasteful, and we can do better by not resetting Dijkstra's algorithm between different runs from the same source. For each vertex i in the region of interest (with a nonzero population, unpopulated vertices can be skipped), we run Dijkstra's algorithm until the priority queue is empty. During the search, we maintain the cumulated population s of all vertices scanned so far. In each iteration, we extract a vertex j from the queue, relax its outgoing edges, and check whether j has a nonzero population. If so, we draw T_{ij} from the binomial distribution described above (s_{ij} is equal to s at that point), and output as many trips between vertices i and j. Algorithm 3.1 gives pseudocode for the straightforward implementation, which we call FRAD (for formula-based radiation model implementation).

Asymptotic Analysis. By resuming the Dijkstra searches, we decrease the total number of them from $|V|^2$ to $|V|$. Since binomial variates can be generated in constant expected time, for example by using the ratio-of-uniforms method [KM77, Sta90], the total expected time of FRAD is $O(|V|\,\mathrm{Dij}(|V|, |E|))$. Using a binary heap [Wil64] or a Fibonacci heap [FT87], the running time $\mathrm{Dij}(|V|, |E|)$ of Dijkstra's algorithm becomes $O((|E| + |V|)\log|V|)$ or $O(|E| + |V|\log|V|)$, respectively. For sparse graphs with $|E| \in O(|V|)$, such as road networks, we obtain $O(|V|\log|V|)$ in both cases. Therefore, the FRAD algorithm runs in superquadratic expected time $O(|V|^2\log|V|)$.

3.4.2 Output-Sensitive Implementation

The superquadratic execution time restricts the applicability of the FRAD algorithm to relatively small networks. However, the output size (i.e., the number of trips) grows only linearly with the total population M, providing a lower bound $\Omega(M)$ on the execution time of our algorithms. In this section, we describe an output-sensitive algorithm that comes close to this lower bound, is thus almost optimal, and scales to continental networks having millions of vertices.

To obtain subquadratic time, we ignore the formula provided by the radiation model, and build directly upon the underlying assumptions of the model. Recall that the radiation model is based on the following main idea: Each traveler assigns to all opportunities a fitness or attractiveness value, drawn independently from a common distribution. Then, the traveler selects the closest opportunity with a fitness higher than the traveler's fitness threshold, drawn from the same distribution. The radiation model with selection decreases the probability of selecting an opportunity by a factor of $1 - \lambda$. In the simplest version, the number of opportunities is approximated by the population, i.e., there are M opportunities in a region with a population of M.

Our output-sensitive algorithm repeatedly generates a trip until we have a total of γM trips. We now show how to implement the generation of a single trip to run in almost constant expected time. First, we pick the origin vertex. The radiation model assumes that the flow O_i out of vertex i is proportional to the population m_i of i, i.e., that $O_i = \gamma m_i$. Therefore, we draw the origin O from a discrete distribution determined by the probability function $\Pr[O = i] = m_i/M$.

It remains to pick a destination vertex corresponding to the chosen origin vertex. Let X_0 be the fitness threshold of the traveler and let $X_1, \ldots, X_M$ be the fitness values of the M opportunities. We define O_{fit} as the number of opportunities with a higher fitness than the traveler's fitness threshold. Since $X_0, \ldots, X_M$ are independently and identically distributed, O_{fit} is a uniform random number in $0..M$. We define O_{sel} as the number of selectable opportunities, which obeys a binomial distribution with O_{fit} trials and success probability $1 - \lambda$. To pick the destination, we successively draw O_{fit} and O_{sel} from the respective distribution. If O_{sel} is zero, the traveler did not find any selectable opportunity. In this case, we draw O_{fit} and O_{sel} again, which is consistent with the constant c_i in Equation (3.2). Let $O_{\text{sel}}^{>0}$ be the value of the first nonzero O_{sel}.

Now, we know the total number of opportunities that can be selected by the traveler. Let O_{int} be the total number of opportunities that are closer to the origin vertex than any selectable opportunity. Since the selectable opportunities are uniformly distributed, O_{int} can be seen as the number of failures in a sequence of draws from a population of size M containing $O_{\text{sel}}^{>0}$ successes before a success occurs. That is, O_{int} obeys a negative hypergeometric distribution. To finally pick the destination, we draw O_{int} accordingly and run Dijkstra's algorithm from the origin, stopping as soon

Algorithm 3.2: Output-sensitive implementation of the radiation model, repeatedly generating a trip using Dijkstra.

```
1  Function $DRAD(G = (V, E), \gamma, \lambda)$
2      for $k \leftarrow 1$ to $\gamma M$ do
3          $O \leftarrow discreteVariate(\Pr[O = i] = m_i/M)$
4          $D \leftarrow O$
5          while $O = D$ do
6              $O_{\text{sel}} \leftarrow 0$
7              while $O_{\text{sel}} = 0$ do
8                  $O_{\text{fit}} \leftarrow uniformIntVariate(1, M)$
9                  $O_{\text{sel}} \leftarrow binomialVariate(O_{\text{fit}}, 1 - \lambda)$
10             $O_{\text{int}} \leftarrow negativeHypergeomVariate(1, O_{\text{sel}}, M)$
11             $dijkstra.initialize(O)$
12             while $O_{\text{int}} \geq 0$ do
13                 $D \leftarrow dijkstra.settleNextVertex()$
14                 $O_{\text{int}} \leftarrow O_{\text{int}} - m_D$
15         output OD pair $(O, D)$
```

as we have visited $O_{\text{int}} + 1$ opportunities. The last vertex scanned by the search is the destination we are looking for. Note that our search graph once again consists of all roads in the region of interest and the major roads in the surrounding area. Algorithm 3.2 gives pseudocode for the output-sensitive implementation, which we call DRAD (for Dijkstra-based radiation model implementation).

Asymptotic Analysis. We start by estimating the expected running time for the Dijkstra search. The stopping criterion of the search depends on the number O_{int} of opportunities that are closer to the origin than any selectable opportunity. More precisely, the search stops when $O_{\text{int}} + 1$ opportunities have been visited. Therefore, we need to estimate the expected value of O_{int}.

Recall that O_{int} is the number of failures in a sequence of draws from a population of size M containing $O_{\text{sel}}^{>0}$ successes before a success occurs. To obtain $O_{\text{sel}}^{>0}$, we repeatedly draw O_{sel} until a nonzero variate occurs. We ignore this complication for now. Let $\hat{O}_{\text{int}}$ obey the same negative hypergeometric probability distribution as O_{int}, with $O_{\text{sel}}^{>0}$ replaced by O_{sel}. We estimate the expected value of $\hat{O}_{\text{int}}$. Let $p = 1 - \lambda$ be the probability that a sufficiently fit opportunity is selected. We obtain

$$\mathrm{E}[\hat{O}_{\text{int}}] = \sum_{k=0}^{M} \Pr[O_{\text{fit}} = k] \, \mathrm{E}[\hat{O}_{\text{int}} \mid O_{\text{fit}} = k]$$

$$
\begin{aligned}
&= \sum_{k=0}^{M} \Pr[O_{\text{fit}} = k] \sum_{\ell=0}^{k} \Pr[O_{\text{sel}} = \ell \mid O_{\text{fit}} = k] \,\mathrm{E}[\hat{O}_{\text{int}} \mid O_{\text{fit}} = k, O_{\text{sel}} = \ell] \\
&= \sum_{k=0}^{M} \frac{1}{M+1} \sum_{\ell=0}^{k} \binom{k}{\ell} p^{\ell} (1-p)^{k-\ell} \frac{M-\ell}{\ell+1} \\
&= \sum_{k=0}^{M} \sum_{\ell=0}^{k} \left(\frac{1}{\ell+1} - \frac{1}{M+1} \right) \binom{k}{\ell} p^{\ell} (1-p)^{k-\ell} \\
&= \sum_{k=0}^{M} \sum_{\ell=0}^{k} \frac{1}{\ell+1} \binom{k}{\ell} p^{\ell} (1-p)^{k-\ell} - \frac{1}{M+1} \sum_{k=0}^{M} \sum_{\ell=0}^{k} \binom{k}{\ell} p^{\ell} (1-p)^{k-\ell} \\
&= \sum_{k=0}^{M} \sum_{\ell=0}^{k} \frac{1}{\ell+1} \binom{k}{\ell} p^{\ell} (1-p)^{k-\ell} - \frac{1}{M+1} \sum_{k=0}^{M} 1 \\
&= \sum_{k=0}^{M} \sum_{\ell=0}^{k} \frac{1}{\ell+1} \binom{k}{\ell} p^{\ell} (1-p)^{k-\ell} - 1 \\
&= \sum_{k=0}^{M} \sum_{\ell=0}^{k} \frac{1}{k+1} \binom{k+1}{\ell+1} p^{\ell} (1-p)^{k-\ell} - 1 \\
&= \frac{1}{p} \sum_{k=0}^{M} \frac{1}{k+1} \sum_{\ell=0}^{k} \binom{k+1}{\ell+1} p^{\ell+1} (1-p)^{(k+1)-(\ell+1)} - 1 \\
&= \frac{1}{p} \sum_{k=0}^{M} \frac{1}{k+1} \sum_{u=1}^{k+1} \binom{k+1}{u} p^{u} (1-p)^{(k+1)-u} - 1 \\
&= \frac{1}{p} \sum_{k=0}^{M} \frac{1-(1-p)^{k+1}}{k+1} - 1 \\
&= \frac{1}{p} \sum_{k=1}^{M+1} \frac{1-(1-p)^{k}}{k} - 1 \in \Theta\left(\sum_{k=1}^{M} \frac{1}{k} \right) = \Theta(\log M)
\end{aligned}
$$

Hence, if we allow O_{sel} to be zero, the expected number of opportunities closer to the origin than any selectable opportunity is $\Theta(\log M)$. Let us now consider the case where O_{sel} is not allowed to be zero, as is the case in the actual algorithm. We estimate the expected value of O_{int}. We have

$$
\mathrm{E}[O_{\text{int}}] = \Pr[O_{\text{sel}} = 0] \,\mathrm{E}[O_{\text{int}}] + \sum_{\ell=1}^{M} \Pr[O_{\text{sel}} = \ell] \,\mathrm{E}[\hat{O}_{\text{int}} \mid O_{\text{sel}} = \ell]
$$

$$
\begin{aligned}
&= \frac{\sum_{\ell=1}^{M} \Pr[O_{\text{sel}} = \ell] \, \mathrm{E}[\hat{O}_{\text{int}} \mid O_{\text{sel}} = \ell]}{1 - \Pr[O_{\text{sel}} = 0]} \\
&= \frac{\mathrm{E}[\hat{O}_{\text{int}}] - \Pr[O_{\text{sel}} = 0] \, \mathrm{E}[\hat{O}_{\text{int}} \mid O_{\text{sel}} = 0]}{1 - \Pr[O_{\text{sel}} = 0]}
\end{aligned}
$$

In order to proceed with the analysis, we need to determine the probability that O_{sel} is zero. Since O_{sel} obeys the binomial distribution described above, we obtain

$$
\begin{aligned}
\Pr[O_{\text{sel}} = 0] &= \sum_{k=0}^{M} \Pr[O_{\text{fit}} = k] \Pr[O_{\text{sel}} = 0 \mid O_{\text{fit}} = k] \\
&= \sum_{k=0}^{M} \frac{1}{M+1} \binom{k}{0} p^0 (1-p)^{k-0} \\
&= \frac{1}{M+1} \sum_{k=0}^{M} (1-p)^k \\
&= \frac{1}{M+1} \frac{1-(1-p)^{M+1}}{1-(1-p)} \\
&= \frac{1-(1-p)^{M+1}}{p(M+1)}
\end{aligned}
$$

Substituting this expression into the expression for the expected value of O_{int} yields

$$
\begin{aligned}
\mathrm{E}[O_{\text{int}}] &= \frac{\mathrm{E}[\hat{O}_{\text{int}}] - \frac{1-(1-p)^{M+1}}{p(M+1)} \frac{M-0}{0+1}}{1 - \frac{1-(1-p)^{M+1}}{p(M+1)}} \\
&= \frac{\mathrm{E}[\hat{O}_{\text{int}}] - \frac{1-(1-p)^{M+1}}{p} \frac{M}{M+1}}{1 - \frac{1-(1-p)^{M+1}}{p} \frac{1}{M+1}} \in \Theta(\mathrm{E}[\hat{O}_{\text{int}}]) = \Theta(\log M)
\end{aligned}
$$

Hence, forbidding O_{sel} to be zero has no impact on the expected number of intervening opportunities closer to the origin than any selectable opportunity. Under the assumptions that the number of opportunities is proportional to the size of the network and that the opportunities are evenly distributed over the network, the Dijkstra search runs in expected time $O(\log M \log\log M)$.

Since all of our variates can be generated in constant expected time, and the number of draws before obtaining a nonzero O_{sel} obeys a geometric distribution with probability parameter $1 - \Pr[O_{\text{sel}} = 0]$ and expected value

$$
\frac{\Pr[O_{\text{sel}} = 0]}{1 - \Pr[O_{\text{sel}} = 0]} = \frac{1-(1-p)^{M+1}}{p(M+1) - 1 + (1-p)^{M+1}} \in \Theta\left(\frac{1}{M}\right)
$$

the total expected time to generate a single trip is $O(\log M \log\log M)$. Since we generate γM trips, DRAD runs in expected time $O(M \log M \log\log M)$. Note that this bound comes close to our simple lower bound of $\Omega(M)$.

3.4.3 Scalable Implementation

Although the DRAD method is almost optimal, the constant factors involved are actually large. In this section, we describe an algorithm that improves the practical performance even further, taking only seconds on continental networks. The algorithm, which we call TRAD, works similar to DRAD: It repeatedly generates a trip until we have a total of γM trips. Like DRAD, it draws the origin vertex O from a discrete distribution determined by the probability function $\Pr[O = i] = m_i/M$, and generates the number O_{sel} of selectable opportunities as described above. However, it differs from DRAD in how it picks the destination vertex based on O_{sel}.

The general idea is to find the selectable opportunity closest to the origin using a nearest-neighbor search [FBF77] in a kd-tree [Ben75]. Each node in a kd-tree corresponds to a region of the plane. The region of the root is the whole plane and the leaves correspond to small disjoint blocks partitioning the plane. Given an origin vertex, we sample selectable opportunities only in regions that are close to the origin, and pick the closest opportunity among those. In order to do so, we need to know how many selectable opportunities we have to sample in which region.

We make the following crucial observation. Define $O_{\text{sel}}(v)$ as the number of selectable opportunities in the region corresponding to a node v and define $O_{\text{tot}}(v)$ as the total number of opportunities in the region corresponding to v. Consider a node p and let c_l and c_r be its left child and right child, respectively. Then, $O_{\text{sel}}(c_\text{l})$ can be seen as the number of successes in $O_{\text{sel}}(p)$ draws from a population of size $O_{\text{tot}}(p)$ containing $O_{\text{tot}}(c_\text{l})$ successes. In other words, $O_{\text{sel}}(c_\text{l})$ obeys a hypergeometric distribution. Consequently, $O_{\text{sel}}(c_\text{r})$ is $O_{\text{sel}}(p)$ - $O_{\text{sel}}(c_\text{l})$. In the following, we turn this observation into an algorithm and work out the details.

Building the Tree. Before we can generate the first trip, we need to build a kd-tree storing the vertices in our network. It suffices to build a kd-tree for all vertices with a nonzero number of opportunities, since only those vertices are potential destinations. We split the set of vertices with a splitting line into two subsets of roughly equal size and then recurse on each subset. The recursion ends when the resulting subsets are sufficiently small (we use a recursion threshold of 16 in our experiments, determined experimentally). We split with a vertical line at levels whose depth is even, and we split with a horizontal line at levels whose depth is odd.

Each kd-tree node stores the line chosen to split the region corresponding to the node. In addition, we store at each node v the total number $O_{\text{tot}}(v)$ of opportunities in the region. Moreover, we keep a list of all vertices with a nonzero number of opportunities, starting with the vertices in the region of the leftmost leaf, followed by the vertices in the region of the second leftmost leaf, and so on. For each interior and leaf node v, the vertices in the region corresponding to v are thus a contiguous sublist, whose starting and ending position we store at v. A second list stores the opportunity counts of all vertices with a nonzero number of opportunities in the same order. That allows efficient retrieval of the vertices in the region of the node v and their corresponding opportunity counts.

Finding the Closest Selectable Opportunity. The query algorithm (see Algorithm 3.3, with subroutines in Algorithm 3.4) traverses the kd-tree, starting at the root, and maintaining the closest selectable opportunity seen so far. Moreover, we maintain the number $O_{\mathrm{sel}}(v)$ of selectable opportunities in the region corresponding to the current node v. At the root r, $O_{\mathrm{sel}}(r)$ is simply the number O_{sel} of selectable opportunities in the whole region of interest, generated as in the DRAD algorithm. When the traversal reaches an interior node v with the left child c_{l} and the right child c_{r}, we draw $O_{\mathrm{sel}}(c_{\mathrm{l}})$ from the hypergeometric probability distribution described above, and set $O_{\mathrm{sel}}(c_{\mathrm{r}})$ to $O_{\mathrm{sel}}(v) - O_{\mathrm{sel}}(c_{\mathrm{l}})$.

We then recurse on the child whose region is closer to the origin, and when control returns, we recurse on the other child. Note that we use geographical instead of shortest-path distances here. There are two pruning rules: We prune the search at any node v with $O_{\mathrm{sel}}(v) = 0$, and at any node v whose region is further from the origin than the closest selectable opportunity seen so far. When the traversal reaches a leaf node, we proceed with the base case as follows.

Handling the Base Case. When reaching a leaf node v, we sample $O_{\mathrm{sel}}(v)$ selectable opportunities in the region corresponding to v. For each of those opportunities, we check whether it improves the closest selectable opportunity seen so far. If this is the case, we update the best solution found so far.

It remains to explain how we sample the selectable opportunities. More precisely, we wish to sample $O_{\mathrm{sel}}(v)$ vertices from all vertices $\{v_1, v_2, \ldots\}$ in the region of v. The probability that a vertex is chosen should be proportional to the number of not yet chosen opportunities at that vertex. Note that this problem is similar (but not identical) to weighted sampling without replacement [Dev86, WE80].

Let W denote an array where $W[j]$ stores the number of not yet chosen opportunities at v_j. Initially, we copy the (contiguous) sublist of opportunity counts at the v_j's to W. Besides, let $S(j)$ be the prefix sums $\sum_{k=1}^{j} W[k]$. To simplify notation, $S(0) = 0$.

To sample the i-th vertex, we generate a uniform random number r in $0..O_{\mathrm{tot}}(v) - i$. Then, we find j such that $S(j-1) \leq r < S(j)$ with a linear sweep over the array W.

Algorithm 3.3: Tree-based destination selection method using geometric distances, resembling a nearest-neighbor search in a kd-tree.

```
Global variables
    $q = (q_x, q_y)$ denotes the coordinates of the origin vertex
    $b^+ = (b^+_x, b^+_y)$ denotes the top-right of the current region
    $b^- = (b^-_x, b^-_y)$ denotes the bottom-left of the current region
    $D$ denotes the closest selectable opportunity seen so far
    $d$ denotes the distance from the origin vertex to $D$
Function $findClosestSelOpportunity(v, O_{sel}(v))$
    if $v$ is a leaf or recursion threshold is deceeded then
        $handleBaseCase(v, O_{\text{sel}}(v))$
        return
    $O_{\text{sel}}(c_l) \leftarrow hypergeomVariate(O_{\text{sel}}(v), O_{\text{tot}}(c_l), O_{\text{tot}}(v))$
    $O_{\text{sel}}(c_r) \leftarrow O_{\text{sel}}(v) - O_{\text{sel}}(c_l)$
    $dim \leftarrow splitDim(v)$
    $s \leftarrow splitVal(v)$
    if $q_{dim} \leq s$ then
        if $O_{\text{sel}}(c_l) > 0$ then
            $(tmp, b^+_{dim}) \leftarrow (b^+_{dim}, s)$
            $findClosestSelOpportunity(c_l, O_{\text{sel}}(c_l))$
            $b^+_{dim} \leftarrow tmp$
        if $O_{\text{sel}}(c_r) > 0$ then
            $(tmp, b^-_{dim}) \leftarrow (b^-_{dim}, s)$
            if $boundsIntersectDisk()$ then
                $findClosestSelOpportunity(c_r, O_{\text{sel}}(c_r))$
            $b^-_{dim} \leftarrow tmp$
    else
        if $O_{\text{sel}}(c_r) > 0$ then
            $(tmp, b^-_{dim}) \leftarrow (b^-_{dim}, s)$
            $findClosestSelOpportunity(c_r, O_{\text{sel}}(c_r))$
            $b^-_{dim} \leftarrow tmp$
        if $O_{\text{sel}}(c_l) > 0$ then
            $(tmp, b^+_{dim}) \leftarrow (b^+_{dim}, s)$
            if $boundsIntersectDisk()$ then
                $findClosestSelOpportunity(c_l, O_{\text{sel}}(c_l))$
            $b^+_{dim} \leftarrow tmp$
```

The vertex we are looking for is v_j. Finally, we decrement $W[j]$, since we have chosen one of the opportunities at vertex v_j, which has to be taken into account when sampling the $i+1$-th vertex in the next iteration.

Algorithm 3.4: Subroutines used by Algorithm 3.3.

```
Function handleBaseCase($v, O_{sel}(v)$)
    let $\{v_1, v_2, \dots\}$ be the vertices in the region of node $v$
    copy the opportunity counts at the $v_j$'s into array $W$
    for $i \leftarrow 1$ to $O_{\text{sel}}(v)$ do
        $r \leftarrow$ uniformIntVariate$(0, O_{\text{tot}}(v) - i)$
        $j \leftarrow 1$
        $s \leftarrow W[1]$
        while $s \leq r$ do $(j, s) \leftarrow (j + 1, s + W[j + 1])$
        $W[j] \leftarrow W[j] - 1$
        if $|coordinates(v_j) - q|^2 < d$ then
            $D \leftarrow v_j$
            $d \leftarrow |coordinates(v_j) - q|^2$
Function boundsIntersectDisk()
    $b \leftarrow 0$
    if $q_x < b_x^-$ then
        $b \leftarrow b + (q_x - b_x^-)^2$
    else if $q_x > b_x^+$ then
        $b \leftarrow b + (q_x - b_x^+)^2$
    if $q_y < b_y^-$ then
        $b \leftarrow b + (q_y - b_y^-)^2$
    else if $q_y > b_y^+$ then
        $b \leftarrow b + (q_y - b_y^+)^2$
    return $b < d$
```

Since we can efficiently retrieve the vertices in the region of any node, and their opportunity counts, we may call the base-case algorithm not only for leaves but also for interior nodes. Therefore, we should break the recursion as soon as handling the base case is no more costly than handling the recursion. The base-case algorithm runs in time $O(O_{\text{sel}}(v) \cdot k)$, where k is the number of vertices in the region corresponding to v, and thus we switch to it as soon as $O_{\text{sel}}(v) \cdot k$ drops below some threshold (we use a recursion threshold of 1024 in our experiments, determined experimentally).

3.4.4 Other Proxies for the Attraction Rate

In the simplest version, the radiation model assumes that the number of trips leaving a zone is proportional to its population, and that the attractiveness of a zone is also proportional to its population. Yang et al. [YHEG14] propose to distinguish the *production rate* of a zone from its *attraction rate*, i.e., to use distinct proxies for them.

They keep the population density as a proxy for the production rate, but use the number of points-of-interest (POIs) in a zone as its attraction value. POIs are all types of potential destinations, such as restaurants, post offices, groceries, schools, churches, and hospitals. Note that POIs are also maintained in OpenStreetMap.

We can easily incorporate distinct proxies for the production and attraction rate into our algorithms. Both rates are maintained as separate arrays, indexed by vertex IDs, allowing efficient access to the production and attraction value of any vertex. Adapting the FRAD algorithm is straightforward. When generating the flow O_i out of vertex i, we substitute the production rate of i for m_i. When computing the probability $c_i p_{ij}$ that a trip starting at vertex i ends at vertex j, we substitute the attraction rates of i and j for m_i and m_j, respectively, and let Dijkstra's algorithm cumulate the attraction values of scanned vertices to obtain s_{ij}. The adaptation of DRAD and TRAD is also straightforward. When picking the origin, we substitute the production rate of each vertex i for m_i. When choosing the destination, we use the attraction value of each vertex i as the number of opportunities at i.

This way, we generate only trips from residential areas to opportunities. To also take the reverse direction into account, we proceed as follows. Whenever we have generated a trip, we swap origin vertex and destination vertex with probability p_{swap}. The choice of p_{swap} depends on the traffic scenario. During the morning peak, trips mainly go from residential areas to opportunities, and during the evening peak, it is the opposite. In our experiments in Section 3.6, we thus use $p_{\text{swap}} = 0.3$ for the morning peak, $p_{\text{swap}} = 0.6$ for the evening peak, and $p_{\text{swap}} = 0.5$ for the whole day.

3.5 Parallelization

All algorithms described in this chapter are easy to parallelize with perfect speedups. Let c be the number of CPU cores available. RAND, GEOM, DRAD, and TRAD generate one trip at a time. Since the generations of the trips are independent from one another, we assign different trips to separate cores. Note that GEOM takes as input the number T of trips to be generated and the expected trip length μ, and DRAD and TRAD take the number T of trips and the model parameter λ. In all cases each core generates T/c trips. RAND takes as input the total network volume V, and thus each core generates trips until its network volume exceeds V/c.

The FRAD algorithm does not process one trip but one pair of vertices at a time. Since all pairs $(i, \cdot)$ are examined during the same Dijkstra search, those pairs have to be assigned to the same core. However, the examinations of pairs $(i, \cdot)$ and $(j, \cdot)$ with $i \neq j$ are independent from each other, and can be allocated to distinct cores.

To obtain lock-free implementations, the cores write their trips to temporary files. After generating all trips, we merge the temporary files into a single output file.

3.6 Experiments

This section presents an extensive experimental evaluation of all algorithms considered. First, we describe our experimental setup. Second, we evaluate the quality of the demand data calculated by our algorithms. Finally, we compare their performance.

3.6.1 Experimental Setup

Our publicly available code[5] is written in C++14 (with OpenMP for parallelization) and compiled with the GNU compiler 7.4 using the optimization level 3. We use 4-heaps [Joh75] as priority queues. To ensure a correct implementation, we make extensive use of assertions (disabled during measurements). Our benchmark machine runs openSUSE Leap 15.0 (kernel 4.12.14), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each with eight cores clocked at 3.50 GHz and 8×64 KiB of L1, 8×1 MiB of L2, and 24.75 MiB of shared L3 cache.

Inputs. Our main benchmark instance, taken from a real-world production system, is the Stuttgart Region [SHP11], Germany, encompassing about 2.6 million inhabitants. The experiments are run on the largest strongly connected component, which consists of 134 663 vertices and 307 759 edges. The length of an edge represents the travel time between its endpoints. Note that the network also contains the major roads in the area surrounding the Stuttgart Region.

The instance provides demand data for a whole week. The demand was originally forecasted using mobiTopp [MKV13, MV15], which was calibrated from a household travel survey [VRS11] conducted in 2009/2010. The raw data contains about 51.8 million trips between 1174 zones, encompassing various modes of transportation such as pedestrian, bicycle, public transit, and car. For our experiments, we only consider car trips, and extract three different traffic scenarios: a morning peak, an evening peak, and a whole day. We assume the endpoints to be uniformly distributed in the zones, and pick for each trip the origin vertex and the destination vertex uniformly at random from its origin and destination zone, respectively.

We take population densities from two different population grids. We mainly use the grid made available by the Federal Statistical Office of Germany. It has a resolution of 100 m and covers Germany. We also use the population grid made available by Eurostat, which has a resolution of 1 km and covers all EU and EFTA member states. Note that POI data is provided by the Stuttgart instance.

Methodology. We evaluate the quality of the demand data calculated by our algorithms and their performance. Measuring the latter is obvious. The solution quality is evaluated by comparing synthetic data provided by our algorithms to reference

[5] `https://github.com/vbuchhold/routing-framework`

data used in production systems. However, how to make such a comparison is nonobvious. We argue that two demand data sets are similar if they yield similar traffic patterns in the road network. Therefore, we proceed as follows. First, we assign the reference demand to the network (using the state-of-the-art traffic assignment algorithm that we will introduce in Chapter 5), which provides the traffic volume of each edge. Then, we assign the synthetic demand. Finally, we compare the saturation of each edge resulting from the synthetic demand to the one resulting from the reference demand, by plotting the synthetic edge saturation against the reference edge saturation. We prefer to compare saturation (i.e., the volume on an edge divided by its capacity) rather than pure volume, since for the purposes of transportation planning, it is particularly important to forecast the bottlenecks (i.e., strongly saturated road segments that impede traffic flow) on a transportation network. Note that accurate and reliable road capacities are also provided by the benchmark instances.

Moreover, we compare the trip duration distributions for the reference and synthetic demand. This gives further insight into the solution quality obtained.

Population and POI Assignment. Formally, we assume that each vertex in the road network has a nonnegative number of inhabitants. As input, however, we take population grids. Assigning the grid to the graph works as follows. For each inhabitant, we pick a vertex lying in that cell uniformly at random and assign the inhabitant to it. If there is no such vertex, we choose one lying in the Moore neighborhood of that cell with range $r = 1$. If there is still no such vertex, we gradually increase r up to some $r_{\max}$. In our experiments, we use $r_{\max} = 1$ for the German grid with a resolution of 100 m, and $r_{\max} = 0$ for the European grid with a resolution of 1 km.

In contrast, the POI data we are given is a list of geographical coordinates. We assign each POI to the closest vertex no further than $d_{\max}$. We set $d_{\max}$ to 200 m.

Random Variate Generation. In order to keep implementation complexity low, we use existing implementations of random variate generation algorithms. The Standard Template Library offers the three distribution classes `uniform_int_distribution`, `binomial_distribution`, and `geometric_distribution`. Unfortunately, the STL provides neither a hypergeometric nor a negative hypergeometric distribution. To generate hypergeometric variates, we use the stocc library[6]. However, we are not aware of any C++ library that offers a generator for negative hypergeometric variates. Therefore, when we need a negative hypergeometric variate, i.e., the number k of failures in a sequence of draws from a population of size N containing n successes before a success occurs, we approximate k by a geometric variate with parameter $p = n/N$. Note that the geometric distribution slightly underestimates the actual probability for small values of k, and slightly overestimates the probability for large values of k.

[6] `https://www.agner.org/random/`

Table 3.1: Choice of the parameter λ for the Stuttgart Region.

attraction	FRAD	DRAD	TRAD
POP	0.999 982 8	0.999 982 1	0.999 979 3
POI	0.997 41	0.997 32	0.996 81

Parameters. The algorithms described in this chapter have several input parameters. RAND takes the total network volume V, and GEOM takes the number T of trips to be generated and the expected trip length μ. Moreover, FRAD, DRAD and TRAD also take the number T of trips and the model parameter λ. We set T to the number of trips in the reference demand, and V to $T\mu$.

It remains to determine μ and λ. We assume an average trip length of 10 min, and thus set μ to 10 min. Moreover, we determine λ such that the resulting average trip length is μ. Note that we do not fit λ to the reference demand data set; we only fit it to our rough trip length estimate of 10 min. After all, our goal is to show that our algorithms do not require detailed data for calibration.

Table 3.1 shows our choice of λ. When using POI densities as attraction rates, λ is much smaller, since the total number of POIs in our data is much smaller than the total number of inhabitants. Moreover, λ is somewhat smaller for DRAD than for FRAD due to our approximation of the negative hypergeometric distribution by a geometric distribution. As discussed in the previous section, the geometric distribution slightly underestimates the probability of short-range trips, which is compensated for by a somewhat smaller λ. Note that λ deviates more for TRAD because it uses a different metric (geographical instead of shortest-path distances).

3.6.2 Solution Quality

Figure 3.3 compares the quality obtained by our algorithms for a morning peak (7.30–8.30 on a Tuesday) within the Stuttgart Region. The radiation-based algorithms use POI densities as attraction rates. We observe that the radiation model produces demand data sets that are quite similar to the reference demand, and significantly outperform the simple approaches. Both RAND and GEOM grossly underestimate the traffic volume of strongly saturated edges, whereas particularly FRAD and DRAD obtain good estimates for edges of any saturation, and also reduce the amount of dispersion. Moreover, the trip length distributions obtained by the radiation-based algorithms approximate the reference distribution much better. While the curves of the radiation-based algorithms closely follow the reference curve, the curves of RAND and GEOM have a completely different shape.

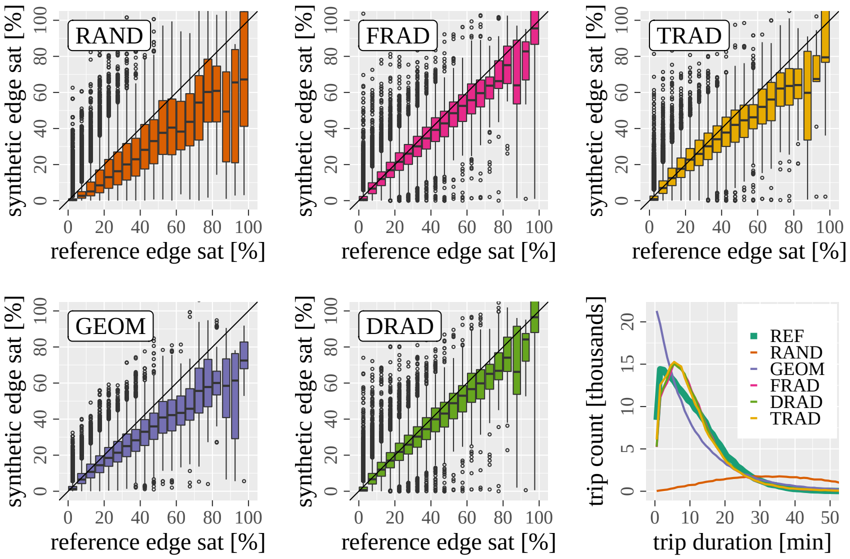


Figure 3.3: Quality of the demand data calculated by our algorithms for the morning peak within the Stuttgart Region. We plot the edge saturation resulting from the synthetic demand (calculated by the respective algorithm) against the edge saturation resulting from the reference demand (cf. Section 3.6.1). The bottom-right plot shows the distribution of the trip duration for the reference demand (thick line) and the synthetic demand (thin lines). The radiation algorithms use the population densities from the German grid as production rates and POI densities as attraction rates.

Note that the demand data sets produced by FRAD and DRAD are almost identical. This is expected, since we specifically designed DRAD to generate the same trips as FRAD, while spending much less time. The solution quality obtained by TRAD is slightly worse than the one obtained by FRAD and DRAD, since TRAD resorts to geographical instead of shortest-path distances. However, it still calculates demand data of better quality than the simple approaches, with a better trip length distribution.

Using Population Densities as Attraction Rates. Figure 3.4 shows the solution quality obtained by the radiation-based algorithms when using population instead of POI densities as attraction rates. We see that switching to population densities has a limited negative impact on the quality, however, particularly FRAD and DRAD still perform better than RAND and GEOM (cf. Figure 3.3). Still, POI densities should be preferred as attraction rates whenever they are available.

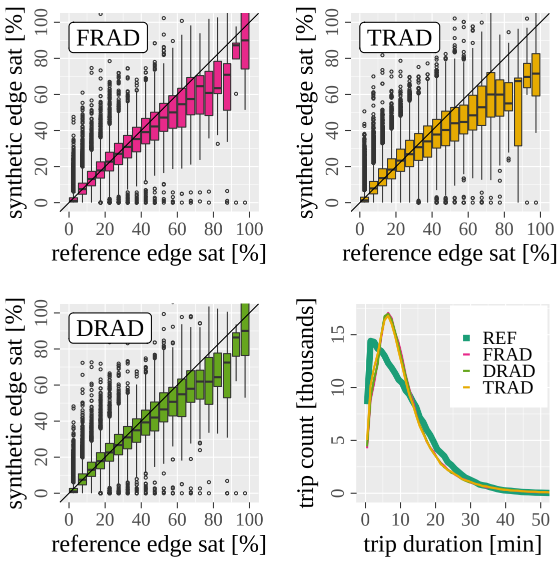


Figure 3.4: Impact on the solution quality when using population instead of POI densities as attraction rates. As in Figure 3.3, we consider the morning peak within the Stuttgart Region, and take the population densities from the German grid.

Other Analysis Period. Next, we consider a different analysis period within the Stuttgart Region. Figure 3.5 compares the solution quality obtained by our algorithms for a whole day (a Tuesday). We observe that this period seems easier to be forecasted, since all algorithms perform better than for the morning peak. We assume the reason is that the traffic is more evenly distributed when averaged over a whole day, and peaks are less pronounced. Still, there is a difference in quality between the radiation-based algorithms and the simple approaches. In particular, both FRAD and DRAD almost perfectly match the reference demand, with a small amount of dispersion.

Other Study Area: Greater London. Finally, we evaluate the solution quality on another region. Besides the Stuttgart Region, we take Greater London from a production system; see Table 3.2 for the key figures of the largest strongly connected component of this transportation network. Figure 3.6 shows the quality obtained by our algorithms for a peak hour. While all algorithms work quite well on average, both RAND and GEOM suffer from a large number of outliers, and a wide dispersion. Again, the radiation-based algorithms give reasonably accurate results, and the trip length distribution they obtain almost exactly match the reference distributions.

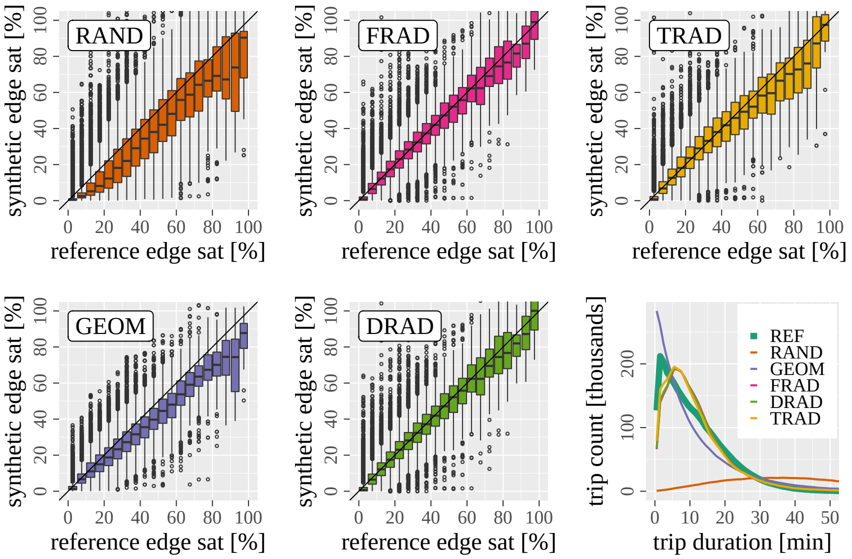


Figure 3.5: Solution quality for a whole day within the Stuttgart Region.[7]

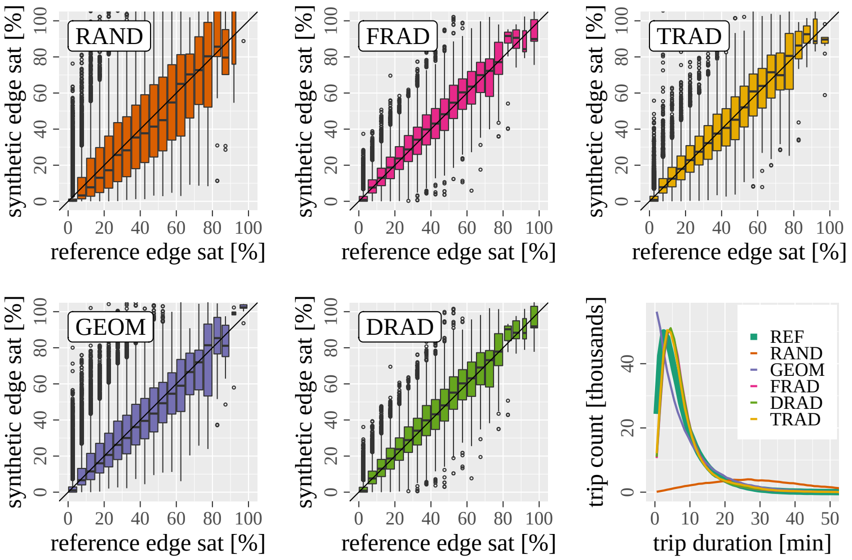


Figure 3.6: Solution quality for a peak traffic hour within Greater London.[8]

[7] As in Figure 3.3, the radiation-based algorithms use population densities from the German grid as production rates and POI densities as attraction rates.

[8] The radiation-based algorithms use population densities from the European population grid made available by Eurostat as both production and attraction rates.

Table 3.2: Key figures of our benchmark instances.

source	input	# vertices	# edges	deg	population	# POIs
PTV	Stuttgart	134 663	307 759	2.29	2 591 973	18 157
PTV	London	45 158	101 897	2.26	8 208 889	–
DIMACS	Belgium	462 843	1 112 155	2.40	11 094 164	–
DIMACS	Germany	4 377 307	10 736 198	2.45	79 168 050	–
DIMACS	Europe	18 017 748	42 560 275	2.36	372 443 839	–

3.6.3 Performance and Scalability

Our last experiment evaluates the performance of our algorithms on various benchmark instances. In addition to the Stuttgart Region and Greater London, we consider three country- and continent-sized road networks from the Ninth DIMACS Implementation Challenge [DGJ09], namely Belgium, Germany, and Western Europe. Table 3.2 shows the key figures of these road networks.

Table 3.3 reports the running time of our algorithms on each network. Our experimental observations strongly support the theoretical analyses in Section 3.4. While it is quite feasible to run FRAD on smaller instances, such as the Stuttgart Region and Greater London, it cannot scale to larger ones. It takes three days on Germany, and on Europe, it would take two months (using all 16 cores, single-threaded execution would take a few years). To estimate the running time of FRAD on Europe, we let it generate trips for all pairs (i, j) with $i \in S \subset V$ and $j \in V$, where the sample S contained 1 ‰ of the vertices in Europe. We then scaled the running time by a factor of 1000. Note that we estimated the running time of FRAD on Germany in the same way (in addition to running the algorithm in full on Germany), and observed that our projection differed only by 2 % from the actual running time. Therefore, we are convinced that our projections for Europe are quite accurate.

Overall, DRAD is faster than FRAD on the Western European road network by up to a factor of 10 000, and TRAD even by up to five orders of magnitude, decreasing the running time from months to seconds.

3.7 Conclusion

We introduced and evaluated three implementations of the recently proposed radiation model [SGMB12] used for predicting human mobility flows. We showed that a straightforward yet careful implementation, denoted by FRAD, does not scale to continental road networks, taking months on our largest network even when

Table 3.3: Running time (in seconds) of our algorithms on different benchmark instances. T denotes the number of trips to be calculated and μ the expected trip duration in minutes. We use population densities as both production and attraction rates. Figures marked with a * are projections.

	T	μ	RAND	GEOM	FRAD	DRAD	TRAD
Stuttgart	10^5	10	5	6	119	4	0.05
	10^5	20	9	18	119	15	0.04
	10^6	10	45	59	124	37	0.29
	10^6	20	90	181	124	152	0.23
London	10^5	10	3	3	19	2	0.03
	10^5	20	5	9	19	8	0.03
	10^6	10	27	32	19	23	0.24
	10^6	20	53	84	19	76	0.15
Belgium	10^6	10	120	52	3 263	28	0.46
	10^6	20	242	252	3 240	149	0.44
	10^7	10	1 184	497	3 343	265	3.58
	10^7	20	2 343	2 467	3 335	1 432	3.48
Germany	10^6	10	809	59	227 880	56	1.43
	10^6	20	1 616	322	249 875	251	1.39
	10^7	10	8 083	581	244 775	507	6.13
	10^7	20	16 134	3 231	264 938	2 405	5.91
Europe	10^7	10	13 949	498	5 136 007*	503	16.44
	10^7	20	27 669	2 659	5 166 284*	2 207	12.09
	10^8	10	138 546	4 932	5 298 487*	4 845	126.10
	10^8	20	275 712	26 509	5 391 445*	21 739	79.72

using 16 CPU cores. Therefore, we presented two output-sensitive, more scalable implementations. The first, denoted by DRAD, uses shortest-path distances, thus delivering the same solution quality as the straightforward implementation, but decreasing the running time from months to minutes. The second one, TRAD, resorts to geographical distances, thus sacrificing some solution quality in order to obtain even better running times. In total, this algorithm is five orders of magnitude faster than the straightforward implementation on our largest network representing Europe.

Our findings make applications of the radiation model to continental networks at the microscopic level practical and can be used to generate large-scale benchmark data for evaluating transportation algorithms. Compared to generation approaches previously used in algorithmic publications, our algorithms produce demand data of better quality, take less time, have similar implementation complexity, and rely only on publicly available data such as population grids.

Future work includes studying the output of our algorithms on other road networks. It would be particularly interesting to obtain production demand data from Switzerland, because the Swiss Federal Statistical Office publishes a whole host of grid data sets[9], including grid data on the number of inhabitants, workplaces and employees per hectare. We expect the number of employees to be an even better proxy for the attraction rate, at least during peak hours.

In the next chapter, we will develop an algorithm that combines the very efficient tree-based sampling technique with shortest-path instead of geographical distances.

[9] `https://www.bfs.admin.ch/bfs/de/home/dienstleistungen/geostat/geodaten-bundesstatistik.html`

4 Nearest-Neighbor Queries for Demand Generation

Customizable contraction hierarchies are one of the most popular route planning frameworks in practice, due to their simplicity and versatility. In this chapter, we present a novel algorithm for finding k-nearest neighbors in customizable contraction hierarchies. Compared to previous bucket-based approaches, our algorithm requires much less target-dependent preprocessing effort. Moreover, we use our novel approach in two concrete applications. The first application are *online* k-closest point-of-interest queries, where the points of interest are only revealed at query time. We achieve query times of about 25 milliseconds on a continental road network, which is fast enough for interactive systems. The second application is travel demand generation. We show how to accelerate the demand generators from the previous chapter by a factor of more than 50 using our novel nearest-neighbor algorithm.

This chapter is based on joint work with Dorothea Wagner [BW21].

4.1 Introduction

Motivated by route planning in road networks, the last two decades have seen intense research on speedup techniques [Bas+16] for Dijkstra's algorithm [Dij59], which rely on a slow preprocessing phase to enable fast queries. Particularly relevant to real-world production systems are customizable speedup techniques, which split preprocessing into a metric-independent part, taking only the network structure into account, and a metric-dependent part (the *customization*), incorporating edge weights (the *metric*). A fast and lightweight customization is a key requirement for important features such as real-time traffic updates and personalized metrics. The most prominent customizable techniques are *customizable route planning* (CRP) [DGPW17] and *customizable contraction hierarchies* (CCHs) [DSW16]. Both achieve similar performance but with different trade-offs, and both are in use in industry.

Modern map-based services must support not only point-to-point shortest-path queries but also many other types of queries. Over the years, both CRP and CCHs have been extended to numerous types of queries and problems. Efentakis and Pfoser [EP14] propose one-to-all and one-to-many algorithms within the CRP framework, and Efentakis et al. [EPV15] extend CRP to nearest-neighbor queries. Delling and Werneck [DW15] present alternative CRP-based algorithms for the one-to-many and nearest-neighbor problem. Baum et al. [BDPW13] extend CRP so that it can find energy-optimal paths for electric vehicles, and Kobitzsch et al. [KRS13] so that it can find multiple alternate routes from the source to the target.

Customizable contraction hierarchies and in particular standard *contraction hierarchies* (CHs) [GSSV12], the predecessors of CCHs, have also received considerable attention; we refer to [Bas+16] for a recent overview. Since each CCH *is a* CH, all algorithms operating on CHs carry over to CCHs. Delling et al. [DGNW13] introduce PHAST, a one-to-all algorithm on CHs. RPHAST [DGW11] is an extension to the one-to-many problem. Alternatively, one-to-many queries on CHs can be solved using the *bucket-based approach* by Knopp et al. [Kno+07]. Geisberger [Gei11] extends the bucket-based approach to the nearest-neighbor problem.

In this chapter, we introduce a novel algorithm for finding k-nearest neighbors in CCHs. The *k-nearest neighbor problem* takes as input a graph $G = (V, E)$, a source $s \in V$, a nonempty set $T \subseteq V$ of targets, and an integer k with $1 \leq k \leq |V|$. The goal is to find the k targets $t_i \in T$ closest to s, i.e., those that minimize $dist(s, t_i)$, where $dist(v, w)$ is the shortest-path distance from v to w. Modern nearest-neighbor algorithms tailored to road networks work in up to four phases [DW15, DGW11, ACT16]. *Preprocessing* takes as input only the network structure, *customization* incorporates the metric into the preprocessed data, *selection* (or *target indexing*) incorporates the set of targets into the data, and *queries* take a source and find the k targets closest to the source. Our algorithm follows this standard four-phase setup.

Note that there is already a nearest-neighbor algorithm by Geisberger [Gei11] which operates on CHs. However, its relatively heavy selection phase makes it only suitable for *offline* queries, where the set of targets is known in advance. This is the case for simple store locators of franchises. However, more common in interactive map-based services are *online* queries, where the set of targets is only revealed at query time. An example is the computation of the closest businesses whose name contains a user-defined keyword. We are not aware of any CH-based algorithm that can solve such queries fast enough for interactive services.

There is indeed an algorithm [DW15] for online nearest-neighbor queries within the CRP framework. As already mentioned, however, CRP and CCHs are on a par with each other and both used in industry with good reasons. For a production system based on the CCH framework, it is usually not desirable to simultaneously maintain a CRP setup to support nearest-neighbor queries. All different types of queries should be solvable within the CCH framework.

Related Work. We start by briefly reviewing the CH- and CRP-based nearest-neighbor algorithms mentioned above. Contraction hierarchies (CHs) [GSSV12] are a point-to-point route planning technique that is much faster than Dijkstra's algorithm (four orders of magnitude on continent-sized networks). CHs replace systematic exploration of *all* vertices in the network with two much smaller search spaces (forward and reverse) in directed acyclic graphs, in which each edge leads from a "less important" vertex to a "more important" one.

The basic idea behind the bucket-based nearest-neighbor algorithm [Gei11] is to precompute and store the reverse CH search spaces of the targets during the selection phase. More precisely, if v appears in the reverse search space from a target t with distance y, then (t, y) is stored in a *bucket* $B(v)$ associated with v. The bucket entries are sorted by nondecreasing distance. The query phase of the bucket-based nearest-neighbor algorithm computes the forward CH search space from the source s. For each vertex v in the search space from s with distance x, we scan the bucket $B(v)$. For each entry $(t, y) \in B(v)$, we obtain an s–t path of length $x + y$. The algorithm maintains the k closest targets seen so far and stops bucket scans when $x + y$ reaches the distance to the k-th closest target found so far.

Customizable route planning (CRP) [DGPW17] is a point-to-point route planning technique that splits preprocessing into a metric-independent part and a metric-dependent customization. Metric-independent preprocessing partitions the network into roughly balanced cells and creates *shortcuts* between each pair of boundary vertices in the same cell. Customization assigns costs to the shortcuts by computing shortest paths within each cell. Queries run a modification of bidirectional search that uses the shortcuts to skip over cells that contain neither the source nor the target. For better performance, we use multiple levels of overlays.

The CRP-based nearest-neighbor algorithm [DW15] marks all cells that contain one or more targets during selection. Queries run a modification of Dijkstra that skips over unmarked cells and descends into marked cells. Since the search discovers targets in increasing order of distance, we can stop when the k-th target is reached.

Of course, there are also nearest-neighbor algorithms tailored to road networks that are based on neither CRP nor CHs. Arguably the simplest one is *incremental network expansion* (INE) [PZMT03], which runs Dijkstra's algorithm until the k-th target is reached. Another straightforward approach is *incremental Euclidean restriction* (IER) [PZMT03]. The basic idea behind IER is to repeatedly retrieve the next closest target based on the straight-line distance (e.g., using an R-tree [Gut84]) and compute the actual distance to it using any shortest-path algorithm as a black box. IER stops when the geometric distance to the next closest target exceeds the shortest-path distance to the k-th closest target so far encountered.

Since IER had only been evaluated using Dijkstra's algorithm, its performance was generally regarded as uncompetitive in practice. In particular, IER combined with Dijkstra cannot possibly be faster than INE. More recently, IER was combined with the speedup technique *pruned highway labeling* [AIKK14], yielding one of the fastest nearest-neighbor algorithms in many cases [ACT16, AC17].

More sophisticated algorithms are SILC [SSA08, SAS05], ROAD [LLZT12, LLZ09], and G-tree [Zho+15, ZLTZ13]. Since previously published results had disagreed on the relative performance of these algorithms, Abeywickrama et al. [ACT16] carefully reimplemented and reevaluated them once more. While G-tree was faster than SILC and ROAD in most cases, the differences were relatively small. Delling and Werneck [DW15] compare the CRP-based nearest-neighbor algorithm to G-tree, claiming that CRP outperforms G-tree. To sum up, all algorithms have comparable performance, with selection and query times of the same order of magnitude. However, a big advantage of CRP (and also of our algorithm) compared to the other approaches is a fast and lightweight customization phase, enabling important features such as real-time traffic updates and personalized metrics.

Our Contribution. We introduce a novel algorithm for finding k-nearest neighbors that operates on CCHs. Our algorithm systematically explores the associated separator decomposition tree in a way similar to nearest-neighbor queries [FBF77] in kd-trees [Ben75]. Its selection phase is orders of magnitude faster than the one of previous bucket-based approaches, which makes it a natural fit for online k-closest point-of-interest (POI) queries. On the road network of Western Europe, we achieve selection times of about 20 milliseconds and query times of a few milliseconds or less. This enables *interactive* online queries, which need to run both the selection and query phase for each client's request. We are not aware of any other nearest-neighbor algorithm operating on CCHs that enables interactive online queries.

In addition to closest-POI queries, we also look at a second concrete application. We show how a slightly modified version of our nearest-neighbor algorithm can be used for travel demand generation (or mobility flow prediction). Here, the problem we consider is computing the number T_{vw} of trips between each pair (v, w) of vertices v, w in a road network. Depending on the expected length of the generated trips, we accelerate the demand generators from Chapter 3 by a factor of more than 50.

Outline. Section 4.2 describes our novel nearest-neighbor algorithm in detail. Section 4.3 continues with two concrete applications in which our algorithm can be used. Section 4.4 presents an extensive experimental evaluation of various closest-POI algorithms and travel demand generators. Section 4.5 concludes with final remarks. The notation and terminology of CCHs were introduced in Section 2.3.

4.2 Our Nearest-Neighbor Algorithm

Our algorithm for finding nearest neighbors in CCHs is inspired by the algorithm of Friedman et al. [FBF77] for finding nearest neighbors in kd-trees [Ben75]. (However, our description requires no knowledge of that algorithm.) During the search, we maintain the k closest targets seen so far in a max-heap $\hat{T}$ using their distances from the source as keys. Initially, $\hat{T} = \{\perp\}$ with $key(\perp) = \infty$. The basic idea is as follows: We systematically explore the separator decomposition tree, but visit only nodes X whose corresponding subgraph G_X contains vertices that are closer to the source than the k-th closest target found so far. For each visited node X, we compute the distance from the source to each target in the separator X, and update $\hat{T}$ accordingly.

The precise algorithm is most easily formulated as a recursive procedure (see Algorithm 4.1). It takes a node X in the separator decomposition tree as parameter. At the first call, X is the root of the separator decomposition. The first step of the procedure is to *examine* all targets $t \in T \cap X$ in the separator X. To examine a target t, we compute the shortest-path distance $dist(s, t)$ from s to t with a standard elimination tree search. If $dist(s, t)$ is less than the maximum key in $\hat{T}$, we insert t into the heap. If $\hat{T}$ now contains $k + 1$ elements, we delete the maximum element from the heap and discard it, since it cannot belong to the result.

Next, we loop over all children Y of X in the separator decomposition tree. If the subgraph G_Y induced by the vertices in $\mathcal{T}_Y$ contains any targets, we add a pair $(Y, dist(s, Y))$ to a set C. We denote by $dist(s, Y)$ the shortest-path distance from s to a closest vertex in G_Y, i.e., $dist(s, Y) = \min_{v \in V(G_Y)} dist(s, v)$. If G_Y contains the source vertex, this distance is zero. Otherwise, we have to compute it, which we will discuss in detail in the following sections.

Finally, we loop over all pairs $(Y, dist(s, Y)) \in C$ in ascending order of distance from the source. If $dist(s, Y)$ is less than the distance to the k-th closest target seen so far, we recurse on Y. Otherwise, $\mathcal{T}_Y$ cannot contain better solutions than those already known, and we can skip this subtree.

Note that when G_X is large but contains only a few targets, it is less costly to loop over all these targets than to explore $\mathcal{T}_X$ until the leaves are reached. Therefore, when the number of targets in G_X drops below a certain threshold, we stop the recursion and examine all targets $t \in T \cap V(G_X)$ in G_X (we use a recursion threshold of 8).

Algorithm 4.1: Recursive formulation of our nearest-neighbor algorithm. At the first call, the parameter X is the root of the separator decomposition.

```
Function searchSepDecomp($X$)
    if the recursion threshold is deceeded then
        examine all targets $t \in T \cap V(G_X)$ in the subgraph $G_X$
        return
    examine all targets $t \in T \cap X$ in the separator $X$
    $C \leftarrow \emptyset$
    foreach child $Y$ of $X$ do
        if $T \cap V(G_Y) \neq \emptyset$ then
            if $s \in V(G_Y)$ then
                $C \leftarrow C \cup \{(Y, 0)\}$
            else
                compute distance $dist(s, Y)$ from $s$ to a closest vertex in $G_Y$
                $C \leftarrow C \cup \{(Y, dist(s, Y))\}$
    foreach $(Y, dist(s, Y)) \in C$ in ascending order of $dist(s, Y)$ do
        if $dist(s, Y)$ is less than distance to $k$-th currently closest target then
            searchSepDecomp($Y$)
```

Accessing Vertices and Targets in Subgraphs. Given a node X in the separator decomposition tree, our algorithm requires easy access to the set of vertices and the set of targets in the subgraph G_X and in the separator X. Accessing the set of vertices in G_X and in X is particularly easy. To improve cache efficiency, the vertices in a CCH are reordered according to the order of contraction. That is, the vertices are numbered in the order in which they are visited by a postorder tree walk of $\mathcal{T}$, where the vertices in each node are visited in any order. Hence, for each $X \in \mathcal{X}$, the vertices in G_X are numbered contiguously, with the vertices in $G_X \setminus X$ appearing before the vertices in X. To support easy access to the vertices in G_X and in X, we only need to store three indices with each X: the vertex in G_X with the smallest index, the vertex in G_X with the largest index, and the vertex in X with the smallest index. An efficient representation of the separator decomposition already stores this information.

The set T of targets is represented by a sorted array. To make the targets in subgraphs (or separators) easily accessible, we use an auxiliary array A of size $|V| + 1$. The element $A[i]$, $0 \leq i \leq |V|$, stores the number of targets among the first i vertices. Note that A can be filled by a single sweep through T and A. In order to access the targets in G_X (or X), we first retrieve the index l of the first vertex and the index r of the last vertex in G_X (or X). The number of targets in G_X (or X) is then $A[r+1] - A[l]$, and the actual targets are stored contiguously in $T[A[l]], \ldots, T[A[r+1] - 1]$.

Computing Shortest Paths to Subgraphs. The most straightforward approach to compute the shortest-path distance $dist(s, X)$ from s to a closest vertex in G_X is a standard Dijkstra-based CCH query, where the reverse search is initialized with all vertices in G_X. Let d_r and Q_r be the distance labels and the queue of the reverse search, respectively. To initialize the reverse search, we set $d_r[v] = 0$ for each vertex $v \in V(G_X)$, $d_r[w] = \infty$ for each vertex $w \in V \setminus V(G_X)$, and $Q_r = V(G_X)$. This yields a correct but inefficient algorithm. However, we can do better.

We define the *boundary* $B(X)$ of G_X as the set of vertices in $V \setminus V(G_X)$ that are adjacent to G_X, i.e., $B(X) = \{w \in V \setminus V(G_X) : (v, w) \in E, v \in V(G_X)\}$. Note that the boundary of any G_X is easily accessible without any additional preprocessing.

Lemma 4.1. *Let u be any vertex in G_X. Then, $\pi^{-1}(b) > \pi^{-1}(u)$ for each $b \in B(X)$.*

Proof. Consider any vertex $b \in B(X)$. Let b be contained in the node $Y \notin V(\mathcal{T}_X)$. We claim that Y lies on the path in $\mathcal{T}$ from X to the root R. Assume otherwise, i.e., Y does not lie on the X–R path. Let Z be the lowest common ancestor of X and Y. Since Z separates G_X and G_Y in G_Z, there is no edge in G that connects G_X and G_Y. This contradicts that b is adjacent to a vertex in G_X. Thus, Y lies on the X–R path. Since the vertices are numbered in the order in which they are visited by a postorder tree walk of $\mathcal{T}$, the vertices in Y are assigned higher ranks than the ones in X. In particular, we have $\pi^{-1}(b) > \pi^{-1}(u)$, which completes the proof. □

Theorem 4.2. *Let u be the highest-ranked vertex in G_X. Then, $B(X) = N_H^{\uparrow}(u)$.*

Proof. Let b be a vertex in $B(X)$. We claim that $b \in N_H^{\uparrow}(u)$. Since G_X is by definition connected, there is a path $\langle u, v_0, \ldots, v_k, b\rangle$ in G with $v_i \in G_X$. Since $\pi^{-1}(v_i) < \pi^{-1}(u)$ by definition and $\pi^{-1}(u) < \pi^{-1}(b)$ by Theorem 4.1, all v_i's are contracted before u and b. Therefore, CCH preprocessing adds a shortcut (u, b), and thus $b \in N_H^{\uparrow}(u)$.

Conversely, let w be a vertex in $N_H^{\uparrow}(u)$, i.e., there is an edge (u, w) in H. Since u is the highest-ranked vertex in G_X and $\pi^{-1}(u) < \pi^{-1}(w)$, we have $w \in V(G) \setminus V(G_X)$. We claim that $w \in B(X)$. Assume otherwise, i.e., $w \in V \setminus (V(G_X) \cup B(X))$. Since $B(X)$ separates u and w in G, the shortcut (u, w) corresponds to some path $\langle u, \ldots, b, \ldots, w\rangle$ in G with $b \in B(X)$. By construction, the vertex b is contracted before u and w. This contradicts Theorem 4.1, completing the proof. □

If $s \in V(G_X)$, then $dist(s, X) = 0$. So, assume $s \notin V(G_X)$. Since $B(X)$ separates s and G_X, and all edge lengths are nonnegative, there is a closest vertex v^* in G_X such that there is a shortest s–v^* path $\langle s, \ldots, b, v^*\rangle$, $b \in B(X)$. Note that (b, v^*) is a shortest edge among all edges $(b, v) \in E, v \in V(G_X)$; otherwise, v^* would not be a closest vertex in G_X. Therefore, $dist(s, X) = \min_{b \in B(X)}(dist(s, b) + \min_{\{(b,v) \in E : v \in V(G_X)\}} \ell(b, v))$. That is, it suffices to initialize the reverse search of the query with all boundary vertices. More precisely, we set $d_\mathrm{r}[b] = \min_{\{(b,v)\in E: v\in V(G_X)\}} \ell(b,v)$ for each vertex $b \in B(X)$, $d_\mathrm{r}[w] = \infty$ for each vertex $w \in V \setminus B(X)$, and $Q_\mathrm{r} = B(X)$. This yields a reasonable algorithm, but we can do even better by exploiting elimination tree queries, which are usually faster than the Dijkstra-based CCH queries used so far.

Recall that the CCH search space $S(b)$ of a vertex b corresponds to the path in the elimination tree from b to the root r. An elimination tree search from b therefore scans all vertices in $S(b)$ in order of increasing rank by traversing the b–r path in the elimination tree. Given a set B of vertices, it is not clear how to enumerate all vertices in the union of the search spaces, since the union generally corresponds to a subtree rather than a path in the elimination tree. However, we can exploit the fact that in our case the set B is the boundary of G_X.

Theorem 4.3. *Let l be the lowest-ranked vertex in $B(X)$. Then, $S(l) = \bigcup_{b\in B(X)} S(b)$.*

Proof. Since $l \in B(X)$, we trivially have $S(l) \subseteq \bigcup_{b\in B(X)} S(b)$, so let $b \neq l$ be a vertex in $B(X)$. We claim that $S(b) \subseteq S(l)$. By Theorem 4.2, the highest-ranked vertex u in G_X is adjacent to both l and b. Since $\pi^{-1}(u) < \pi^{-1}(l) < \pi^{-1}(b)$, CCH preprocessing adds a shortcut (l, b) when u is contracted. We have $b \in S(l)$ and $S(b) \subseteq S(l)$. □

By Theorem 4.3, we can compute the shortest-path distance $dist(s, X)$ with a standard elimination tree query from s to the lowest-ranked vertex in $B(X)$, where we initially set $d_\mathrm{r}[b] = \min_{\{(b,v)\in E: v\in V(G_X)\}} \ell(b,v)$ for each vertex $b \in B(X)$. Since a lower bound on $dist(s, X)$ suffices to preserve the correctness of our nearest-neighbor algorithm, we can also initialize the distance labels to zero. The resulting lower bound is only slightly worse than the exact distance, but initialization is somewhat faster. We observed the lowest running times when using lower bounds.

Accelerating Shortest-Path Searches. Note that the forward searches of all elimination tree queries done during the same nearest-neighbor query start at the same source. Unless we use special pruning criteria (we will introduce one in Chapter 5), the forward searches compute identical distance labels. To further accelerate our nearest-neighbor algorithm, we run the forward search *once* before the systematic exploration of the separator decomposition tree. Whenever we compute the distance to a target or subgraph, we run only the reverse search, which accesses the precomputed distance labels of the forward search.

After scanning a vertex v, a standard elimination tree search immediately initializes the distance label of v to ∞, since it is not accessed anymore afterwards. We maintain this initialization approach for the reverse searches. The forward search, of course, must not immediately initialize the labels. Instead, after the exploration of the separator decomposition tree, we traverse the path in the elimination tree from the source to the root once again, and initialize the forward label of each visited vertex.

4.3 Applications

We continue with two substantially different applications in which our nearest-neighbor algorithm can be used. An obvious application are k-closest POI queries in map-based services. Afterwards, we look at a more abstract application (travel demand generation) where we make slight modifications to our algorithm.

4.3.1 Online Closest-POI Queries

Recall that modern closest-POI algorithms [DW15, DGW11, ACT16] work in up to four phases: preprocessing, customization, selection, and queries. We now divide the work our nearest-neighbor algorithm does into these standard phases. Note that our nearest-neighbor algorithm does nothing else but the standard CCH preprocessing and customization during the first two phases. To support easy access to the set of vertices in a subgraph or separator, we indeed need to associate three indices with each node $X \in \mathcal{X}$ but an efficient representation of the separator decomposition already stores this information. Therefore, we reuse the standard CCH preprocessing and customization, without further modifications.

The selection phase runs POI-dependent preprocessing. The only preprocessed data that depends on the set P of POIs is the auxiliary array A, which makes the POIs in a subgraph or separator easily accessible. As already mentioned, A can be filled by a single sweep through P and A. Finally, the query phase runs the systematic exploration of the separator decomposition tree (including the forward search immediately before the exploration and the initialization of the forward distance labels immediately after the exploration).

Note that our selection phase is lightweight and (as our experiments will show) orders of magnitude faster than the one of previous bucket-based approaches. This makes our nearest-neighbor algorithm a natural fit for *online* k-closest POI queries, where the POIs are only revealed at query time. In this case, we need to run both the selection and query phase for each client's request. Except for simple store locators of franchises, online queries are more common than offline queries in interactive map-based services. For example, whenever the set of POIs is obtained from user-defined keywords, we face online queries that should run in real time.

4.3.2 Travel Demand Generation

A substantially different application in which our nearest-neighbor algorithm can be used is travel demand generation. Here, the problem we consider is computing the number T_{vw} of trips between each pair (v, w) of vertices $v, w \in V$. This problem arises when we want to generate benchmark data for evaluating transportation algorithms, or when we want to predict mobility flows. This section shows how our algorithm can be used to accelerate the travel demand generators introduced in Chapter 3.

Radiation Model. Recall that the foundation for the aforementioned demand generators is the *radiation model* [SGMB12]. This model assumes that each vertex $v \in V$ has a nonnegative number m_v of inhabitants and a nonnegative amount n_v of opportunities. We denote by M the total population in G and by N the total number of opportunities in G. The mobility flow out of each vertex is proportional to its population. Destination selection is based on the following main idea: Each traveler assigns to all opportunities a fitness or attractiveness value, drawn independently from a common distribution. Then, the traveler selects the closest opportunity with a fitness higher than the traveler's fitness threshold, drawn from the same distribution. The *radiation model with selection* [SMN13] decreases the probability of selecting an opportunity by a factor of $1 - \lambda$. Intuitively, increasing λ increases the expected trip length. In the simplest version, the number of opportunities is approximated by the population, i.e., there are M opportunities in a graph with a population of M.

Previous Implementations. In Chapter 3, we introduced two practical implementations of the radiation model. *DRAD* obtains high-quality solutions based on shortest-path distances and *TRAD* obtains high performance but uses geometric distances. Both implementations generate one trip after another. First, they draw the origin O from a discrete distribution determined by the probability function $\Pr[O = v] = m_v/M$. Second, they choose the number O_{fit} of opportunities with a fitness higher than the traveler's fitness threshold uniformly at random in $0..N$. Third, they draw the number O_{sel} of selectable opportunities from a binomial distribution with O_{fit} trials and success probability $1 - \lambda$. It remains to find the selectable opportunity closest to O, given the total number O_{sel} of selectable opportunities in G. This is realized differently by the two implementations.

DRAD draws the number O_{int} of opportunities closer to O than any selectable opportunity from a negative hypergeometric distribution determined by O_{sel} and N, and runs Dijkstra's algorithm from O, stopping as soon as $O_{\text{int}} + 1$ opportunities are visited. The last vertex scanned by the search is the destination of the current trip.

The basic idea of TRAD is to find the selectable opportunity closest to O using a nearest-neighbor query [FBF77] in a kd-tree [Ben75]. Each node in a kd-tree corresponds to a region of the plane. The region of the root is the whole plane and the leaves correspond to small disjoint blocks partitioning the plane. The query algorithm traverses the kd-tree, starting at the root, and maintaining the number $O_{\text{sel}}(v)$ of selectable opportunities in the region corresponding to the current node v. Let $O_{\text{tot}}(v)$ be the total number of opportunities in the region of v.

When the traversal reaches an interior node v in the kd-tree, the algorithm draws the number $O_{\text{sel}}(l)$ of selectable opportunities in the region of the left child l from a hypergeometric distribution with $O_{\text{sel}}(v)$ draws without replacement from a population of size $O_{\text{tot}}(v)$ containing $O_{\text{tot}}(l)$ successes. The number $O_{\text{sel}}(r)$ of selectable opportunities in the region corresponding to the right child r is set to $O_{\text{sel}}(v) - O_{\text{sel}}(l)$. The algorithm then recurses on the child whose region is closer to O, and when control returns, it recurses on the other child. The search is pruned at any vertex v with $O_{\text{sel}}(v) = 0$, and at any vertex whose region is farther from O than the closest selectable opportunity seen so far during the query.

When the traversal reaches a leaf node v, the algorithm samples $O_{\text{sel}}(v)$ selectable opportunities in the region corresponding to v. For each of these opportunities, the algorithm checks whether it improves the closest selectable opportunity seen so far.

Our Implementation. We introduce a new implementation of the radiation model, called CRAD. Our implementation follows TRAD but uses nearest-neighbor queries in a CCH rather than in a kd-tree. In this way, we combine the efficient tree-based sampling approach from TRAD with shortest-path distances. As a result, our implementation obtains high-quality solutions like DRAD, but at much lower cost.

To use our nearest-neighbor algorithm in CRAD, we only need to make slight modifications to the procedure presented in Section 4.2 (see Algorithm 4.2 for the modified procedure). In addition to a node X in the separator decomposition tree, it now takes the number $O_{\text{sel}}(G_X)$ of selectable opportunities in G_X as second parameter. At the first call, X is the root of the separator decomposition and $O_{\text{sel}}(G_X)$ is the number O_{sel} of selectable opportunities in G, obtained as before in DRAD and TRAD. Let $Y_0, \ldots, Y_{d-1}$ be the children of X. As the first step, the procedure now distributes the O_{sel} selectable opportunities in G_X over the subgraphs $G_{Y_0}, \ldots, G_{Y_{d-1}}$ and the separator X. In contrast to the previous TRAD implementation where the opportunities are distributed among exactly two regions (left and right child), we now have $d+1$ regions (d children and the separator). Therefore, $O_{\text{sel}}(G_{Y_0}), \ldots, O_{\text{sel}}(G_{Y_{d-1}}), O_{\text{sel}}(X)$ obey a *multivariate* hypergeometric distribution.

In order to aid intuition, we can think of this discrete probability distribution as drawing $O_{\text{sel}}(G_X)$ balls without replacement from an urn containing $O_{\text{tot}}(G_{Y_i})$ balls of type i for $i = 0, \ldots, d-1$ and $O_{\text{tot}}(X)$ balls of type d. We obtain $O_{\text{sel}}(G_{Y_i})$ balls of type i for $i = 0, \ldots, d-1$ and $O_{\text{sel}}(X)$ balls of type d.

After obtaining $O_{\text{sel}}(G_{Y_0}), \ldots, O_{\text{sel}}(G_{Y_{d-1}}), O_{\text{sel}}(X)$, we sample $O_{\text{sel}}(X)$ selectable opportunities in the separator X, and check whether any of them improves the closest selectable opportunity so far encountered. Next, we loop over all children Y of X in the separator decomposition tree. If the subgraph G_Y contains any selectable opportunities, we add a pair $(Y, \mathit{dist}(O, Y))$ to a set C (recall that O is the origin vertex

Algorithm 4.2: Procedure for finding the closest selectable opportunity in the subgraph G_X, given the number of selectable opportunities in G_X.

```
Function findClosestSelectableOpportunity($X, O_{sel}(G_X)$)
    if the recursion threshold is deceeded then
        sample $O_{\text{sel}}(G_X)$ selectable opportunities in the subgraph $G_X$
        return
    $\langle O_{\text{sel}}(G_{Y_0}), \ldots, O_{\text{sel}}(G_{Y_{d-1}}), O_{\text{sel}}(X) \rangle \leftarrow$
      $multiHypergeomVariate(O_{\text{sel}}(G_X), \langle O_{\text{tot}}(G_{Y_0}), \ldots, O_{\text{tot}}(G_{Y_{d-1}}), O_{\text{tot}}(X) \rangle)$
    sample $O_{\text{sel}}(X)$ selectable opportunities in the separator $X$
    $C \leftarrow \emptyset$
    foreach child $Y$ of $X$ do
        if $O_{\text{sel}}(G_Y) > 0$ then
            if $O \in V(G_Y)$ then
                $C \leftarrow C \cup \{(Y, 0)\}$
            else
                compute distance $dist(O, Y)$ from $O$ to a closest vertex in $G_Y$
                $C \leftarrow C \cup \{(Y, dist(O, Y))\}$
    foreach $(Y, dist(O, Y)) \in C$ in ascending order of $dist(O, Y)$ do
        if $dist(O, Y)$ is less than distance to cur. closest sel. opportunity then
            findClosestSelectableOpportunity($Y, O_{\text{sel}}(G_Y)$)
```

of the current trip). The shortest-path distance $dist(O, Y)$ is computed as discussed in Section 4.2. Finally, we loop over all pairs $(Y, dist(O, Y)) \in C$ in ascending order of distance from the origin. If $dist(O, Y)$ is less than the distance to the closest selectable opportunity so far encountered, we recurse on Y.

4.4 Experiments

This section presents a thorough experimental evaluation of both applications. First, we describe our experimental setup, including our machine, the inputs, and implementation details. Next, we evaluate various closest-POI algorithms, with a focus on their selection and query phases. Finally, we compare CRAD to DRAD and TRAD.

4.4.1 Experimental Setup

Our publicly available code[10] is written in C++17 and compiled with the GNU compiler 9.3 using optimization level 3. We use 4-heaps [Joh75] as priority queues. To

[10] `https://github.com/vbuchhold/routing-framework`

ensure a correct implementation, we make extensive use of assertions. Our benchmark machine runs openSUSE Leap 15.2 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each with eight cores clocked at 3.50 GHz and 8×64 KiB of L1, 8×1 MiB of L2, and 24.75 MiB of shared L3 cache. Note that we consider only single-core implementations.

Inputs. Our benchmark instance is the road network of Western Europe. The network has a total of 18 017 748 vertices and 42 560 275 edges and was made available by PTV AG for the 9th DIMACS Implementation Challenge [DGJ09]. For the evaluation of the travel demand generators, we use the population grid[11] made available by Eurostat, the statistical office of the European Union. The grid has a resolution of one kilometer and covers all EU and EFTA member states, as well as the United Kingdom. We follow the approach in Chapter 3 to assign the grid to the graph. For each inhabitant, we pick a vertex lying in their cell uniformly at random and assign the inhabitant to it. If there is no such vertex, we discard the inhabitant.

Implementation Details. We use the recent network dissection algorithm Inertial Flow [SS15] to compute separator decompositions and associated nested dissection orders, with the balance parameter b set to $3/10$ (determined experimentally). CCH customization uses perfect witness searches [DSW16].

For comparison, we carefully reimplemented the bucket-based nearest-neighbor algorithm by Geisberger [Gei11], which we call BCH. CH preprocessing is taken from the open-source library RoutingKit[12]. Both the forward and reverse CH searches use the stall-on-demand optimization [GSSV12].

The bucket-based nearest-neighbor algorithm can be used as is on CCHs, without further modifications. For better performance, however, we use a tailored version where we replace the Dijkstra-based CH searches used during the selection and query phase by elimination tree searches. Note that in contrast to CH searches, CCH searches are faster without the stall-on-demand technique. On the other hand, stall-on-demand decreases the bucket sizes. Therefore, we use stall-on-demand only for the reverse searches. We call this version BCCH.

To keep implementation complexity of the demand generators low, we use existing implementations of random variate generation algorithms. The Standard Template Library (STL) offers the three distribution classes `uniform_int_distribution`, `binomial_distribution`, and `geometric_distribution`. The STL provides neither a hypergeometric nor a negative hypergeometric distribution. To generate hypergeometric variates, we use the stocc library[13]. Since we are not aware of any C++ library that offers a generator for negative hypergeometric variates, we approximate negative hypergeometric variates by geometric variates, as in Chapter 3.

[11] `https://ec.europa.eu/eurostat/web/gisco/geodata/reference-data/population-distribution-demography/geostat`

[12] `https://github.com/RoutingKit/RoutingKit`

[13] `https://www.agner.org/random/`

Table 4.1: Performance of different closest-POI algorithms for various POI distributions. For each distribution, we report the time to index a set of POIs (selection time), the space consumed by the index (selection space), and the time to find the $k = 1, 4, 8$ closest POIs (query time). For CRP, we take the figures for the online version from the original publication by Delling and Werneck [DW15].

	$\lvert P\rvert = 2^{12}, \lvert B\rvert = 2^{20}$					$\lvert P\rvert = 2^{14}, \lvert B\rvert = \lvert V\rvert$				
	selection		query time [µs]			selection		query time [µs]		
	space	time	POIs to be reported			space	time	POIs to be reported		
algo	[MiB]	[ms]	$k = 1$	$k = 4$	$k = 8$	[MiB]	[ms]	$k = 1$	$k = 4$	$k = 8$
Dij	–	–	846 210	855 438	873 716	–	–	113.4	439.3	883.7
BCH	72.4	134	20	20	21	83.6	481	5.0	8.5	10.7
BCCH	85.5	453	51	52	53	134.9	1 753	6.0	8.8	11.1
CCH	68.7	21	2 353	3 501	4 629	68.7	23	306.7	494.8	702.0
CRP	–	–	–	–	–	0.0	8	–	640.0	–

4.4.2 Online Closest-POI Queries

We start by comparing our nearest-neighbor algorithm (simply called CCH in this section) to Dijkstra's algorithm, BCH, BCCH, and CRP. Note that the performance of closest-POI algorithms is affected not only by the number of POIs but also by their *distribution*. For example, the set of *all* restaurants may be distributed evenly over the whole network, whereas a certain franchise may operate in a local region. To model this, we follow the methodology used by Delling et al. [DGW11] to evaluate different one-to-many shortest-path algorithms.

To obtain our problem instances, we first pick a center c uniformly at random. We then use Dijkstra's algorithm to grow a ball B of size $\lvert B\rvert$ centered at c. Finally, we pick a POI set P of size $\lvert P\rvert$ from B. By varying the parameters $\lvert B\rvert$ and $\lvert P\rvert$, we can model the aforementioned situations. For each combination, we generate 100 POI sets. Each POI set is evaluated with 100 sources picked uniformly at random. That is, each data point is an average over 10 000 queries.

Main Results. Table 4.1 shows the performance of different closest-POI algorithms for two POI distributions on the road network of Western Europe. We observe that Dijkstra's algorithm has reasonable performance when the POIs are evenly distributed over the whole graph ($\lvert B\rvert = \lvert V\rvert$). In this case, any potential source is relatively close to some POI, and thus the Dijkstra search can always stop early. However, Dijkstra is not robust to the POI distribution. When $\lvert B\rvert = 2^{20}$, many potential sources are relatively far from any POI, and the average running times are around one second, which is too slow for interactive map services.

BCH achieves the best (offline) query times for both POI distributions. Note, however, that BCH is no competitor to BCCH, CCH, and CRP, since it operates on *standard* contraction hierarchies, which cannot handle frequent metric updates. We only include BCH for comparison with BCCH, since the bucket-based nearest-neighbor algorithm has not yet been tested on *customizable* contraction hierarchies.

Although we tailored the bucket-based algorithm to CCHs, BCCH is still somewhat slower than BCH. This is expected, since CCHs contain more shortcuts and are thus denser than CHs. The slowdown is a factor of about 3.5 for selection. When $|B| = |V|$, BCCH has only slightly higher (offline) query times than BCH, since the queries relax only a few edges. However, BCCH queries are roughly 2.5 times slower than BCH queries when the targets are in a local region ($|B| = 2^{20}$).

We observe that our nearest-neighbor algorithm (simply called CCH in this section) has considerably higher offline query times than BCCH. On the other hand, CCH achieves much faster selection times. For example, when $|P| = 2^{14}$, offline CCH queries are slower by a factor of 51–63 but CCH selection is faster by a factor of 77. Note that although CCH queries are significantly slower than BCCH queries, they are still slightly faster than CRP-based queries.

Online queries need to run both the selection and query phase for each client's request. Therefore, the time taken by an online query is the sum of selection and query time. We observe that BCCH is not suitable for online queries. When $|P| = 2^{12}$, BCCH takes half a second to answer an online query, and it takes even 1.8 seconds when $|P| = 2^{14}$. In contrast, CCH takes only about 25 milliseconds.

Table 4.1 includes various alternative closest-POI algorithms. In addition, it seems natural to adapt existing one-to-many shortest-path algorithms to the closest-POI problem. Promising candidates that are not based on buckets are CTD [Eis+11, DGW11] and RPHAST [DGW11]. However, since CTD and RPHAST selection take more than 100 milliseconds when $|B| = |V|$, online closest-POI queries based on CTD or RPHAST would be at least four times slower than ours.

Impact of the POI Distribution. Our next experiment considers the impact of the ball size on the performance of the different closest-POI algorithms. Figure 4.1 plots selection and (offline) query times for various ball sizes. We omit online query times for clarity. Since the online query times are dominated by the selection times, online query times would closely follow the selection curves. Except for Dijkstra's algorithm, all selection and query times are very robust to the ball size. While all query algorithms benefit from an even distribution of the POIs (for the aforementioned reasons), this effect is most pronounced for Dijkstra.

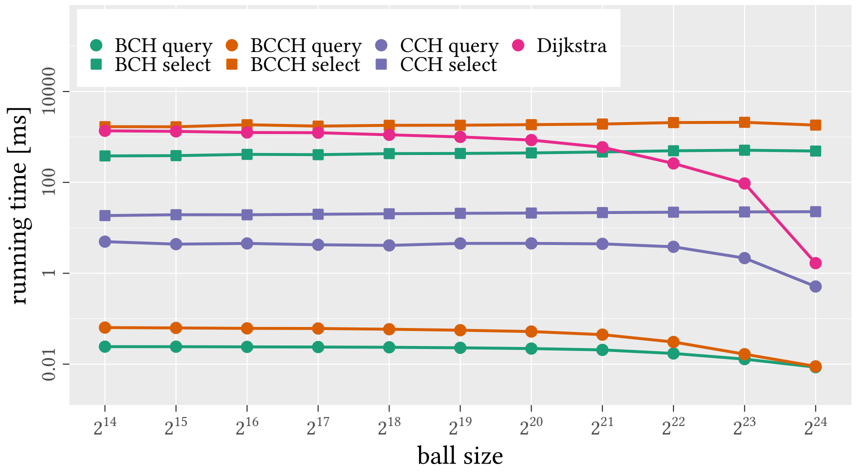


Figure 4.1: Selection and query times of various closest-POI algorithms with $|P| = 2^{14}$ POIs picked at random from a ball of varying size $|B|$. Queries find the 4 closest POIs.

Impact of the Number of POIs. Next, we evaluate the impact of the number of POIs on the performance of Dijkstra's algorithm, BCH, BCCH, and CCH. Figure 4.2 plots selection and (offline) query times for various numbers of POIs. As before, online query times would closely follow the selection curves. We observe that the CCH selection time is independent of the number of POIs, whereas the BCCH selection time grows linearly. For $|P| = 2^{14}$, CCH selection is 76 times faster than BCCH selection. The speedup increases to more than three orders of magnitude for $|P| = 2^{18}$, the largest number of POIs tested in our experiment.

Once again, queries tend to become faster as $|P|$ gets larger, since they can stop (in the case of Dijkstra-based searches) or prune (in the case of elimination tree searches) earlier. The exception are CCH queries, which become slower initially. The reason is that for very small values of $|P|$, we do not explore the separator decomposition tree but trigger the base case at the root (which simply finds $|P|$ point-to-point shortest paths by running standard elimination tree queries from the source to each POI).

4.4.3 Travel Demand Generation

Next, we evaluate CRAD, including a comparison to DRAD and TRAD. Since CRAD uses shortest-path distances rather than geometric distances, it obtains high-quality solutions like DRAD. We verified this experimentally by rerunning the experiments

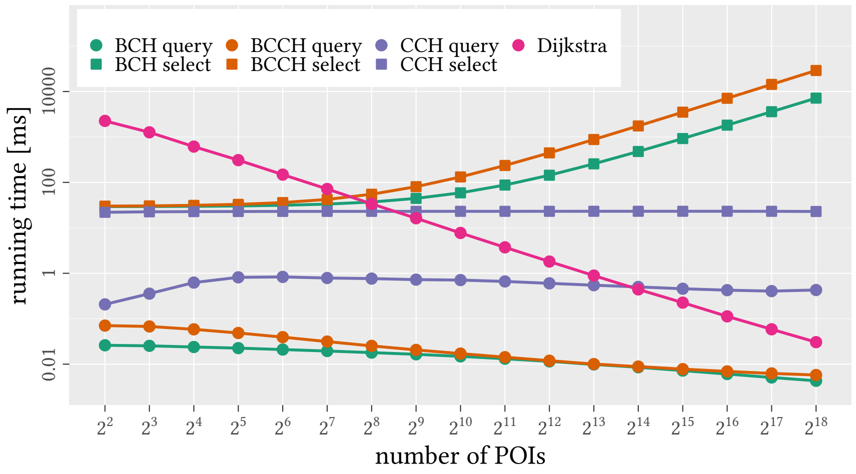


Figure 4.2: Selection and (offline) query times of different closest-POI algorithms with a varying number $|P|$ of POIs picked uniformly at random from a ball of fixed size $|B| = |V|$. Queries find the $k = 4$ closest POIs.

in Chapter 3 for CRAD, using the same instances and methodology. We refer to Chapter 3 for a comparison of the quality with shortest-path and geometric distances.

In this chapter, we focus on the performance of the three implementations of the radiation model. Since DRAD is at its heart a Dijkstra search from the trip's origin to its destination, the performance depends heavily on the expected length of the generated trip (which is controlled by the parameter λ; see Section 4.3.2). In contrast, TRAD and CRAD are robust to the trip length.

Figure 4.3 plots the time to generate a single trip for various values of λ. Note that a value of $\lambda = 1 - 10^{-4}/1 = 0.9999$ leads on our instance to an average trip length of 9 minutes, and a value of $\lambda = 1 - 10^{-4}/100 = 0.999999$ to an average trip length of 72 minutes. Between two data points, the average trip length increases by about 7 minutes. All data points are averages over 100 000 trip generation executions.

We observe that CRAD outperforms DRAD for each value of λ tested. Since TRAD resorts to geometric distances, it still is faster than CRAD by a factor of 28–74. As it obtains worse solutions, however, TRAD is no competitor to CRAD. For an average trip length of about 23 minutes, CRAD gains an order of magnitude over DRAD, and for the largest value of λ tested in our experiment, we see a speedup of 59. Note that this increase in speed is quite useful in practice. While travel demand generation does not need to run in real time, its performance should remain reasonable. However,

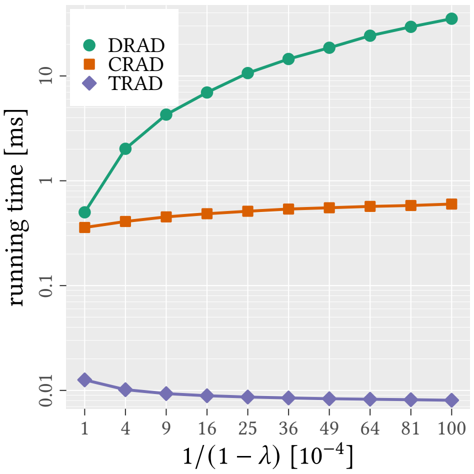


Figure 4.3: Time (in milliseconds) to generate a single trip with different travel demand generators for various values of λ.

DRAD takes about 7 hours to generate one million one-hour trips. In contrast, CRAD takes less than 10 minutes, a significant speedup.

4.5 Conclusion

We presented a novel k-nearest neighbor algorithm that operates on CCHs. With selection times of about 20 milliseconds and query times of a few milliseconds or less, it is the first nearest-neighbor algorithm operating on CCHs that is fast enough for interactive online queries. Interestingly, our algorithm achieves similar performance as the online nearest-neighbor queries by Delling and Werneck [DW15] within the CRP framework. This confirms that CCHs and CRP are on an equal level and solve many different types of problems equally well.

Moreover, we used our nearest-neighbor algorithm to significantly accelerate the demand generators from Chapter 3. We proposed CRAD, a new implementation of the radiation model that combines the advantages of the two previous implementations DRAD and TRAD. CRAD obtains high-quality (shortest-path based) solutions like DRAD, but follows a more efficient tree-based sampling approach like TRAD.

Future work includes accelerating our nearest-neighbor algorithm even further. Note that we compute distances to subgraphs corresponding to the topmost nodes in the separator decomposition more often than distances to subgraphs corresponding to leaves. It would be interesting to see if it pays to precompute the reverse search spaces of the topmost subgraphs. Another possible approach would be to keep frequently used reverse search spaces in an LRU cache. Another interesting project is a parallel version of our algorithm that uses for example task-based parallelism to explore the separator decomposition tree. Finally, it would be interesting to port other point-of-interest algorithms to CCHs, for example best-via queries.

5 Traffic Assignment

Given an urban road network and a set of origin-destination pairs, the traffic assignment problem asks for the traffic flow on each road segment. Common solution algorithms require a large number of shortest-path computations. In this chapter, we significantly accelerate the computation of flow patterns, enabling interactive transportation and urban planning applications. We achieve this by building a traffic assignment procedure upon customizable contraction hierarchies (CCHs), revisiting and carefully engineering CCH customization and queries, and adapting CCHs to compute batched point-to-point shortest paths. Although motivated by the traffic assignment problem, our optimizations apply to CCHs in general. In contrast to previous work, our evaluation uses real-world production data for all parts of the input. On a metropolitan area encompassing about 2.7 million inhabitants, we decrease the flow-pattern computation for a typical 1-hour morning peak (a quarter million trips) from 90.9 to 14.1 seconds on one core and 2.4 seconds on a 16-core machine. This represents a speedup of 37 over the state of the art and more than three orders of magnitude over the Dijkstra-based baseline.

This chapter is based on joint work with Peter Sanders and Dorothea Wagner [BSW18, BSW19b].

5.1 Introduction

The number of drivers traveling along a road segment within a given period is the result of many individual decisions. The common behavioral assumption in practice is that motorists driving between a given origin and destination choose the path with the minimum travel time (known as *Wardrop's first principle* [War52]). This seems natural, since travel is usually not a goal in itself, but entails disutility. However, the travel time on a path depends on the route choice of all other drivers, who themselves are trying to choose minimum travel time routes. Due to congestion, the travel time on a road segment increases with the traffic flow on it. As a result, some drivers choose at some point alternative routes, which can also get congested, and so on. When no driver can improve his travel time by unilaterally changing routes, each route used between a given origin and destination has the same travel time. This condition is known as the *user equilibrium*, and the corresponding flow pattern is called the *equilibrium flow pattern* [She85].

We study the efficient computation of equilibrium flow patterns in road networks. More formally, given an urban road network (represented by a weighted directed graph) and a set of origin-destination (OD) pairs, we want to compute the traffic flow on each road segment at equilibrium. This is known as *traffic assignment*, and it is one of the major problems faced by transportation engineers and urban planners [She85]. In this chapter, we accelerate traffic assignments significantly (by a factor of 37), thereby achieving our goal to enable *interactive* transportation and urban planning applications. Real-time performance is particularly important for applications based on traffic assignments running at road traffic centers, which control and monitor road traffic in real time (for example by opening the hard shoulder for vehicles).

Related Work. The traffic assignment problem has been studied for over 60 years, and motivated extensive research in the operations research community. A comprehensive introduction is offered by the textbook by Sheffi [She85], and the text by Florian and Hearn [FH95]. Perederieieva et al. [PERW15] give a recent overview of practical traffic assignment algorithms. In this chapter, we focus on the static deterministic traffic assignment problem, which was first formulated as a mathematical program in 1956 [BMW56], and is still a ubiquitous tool for traffic and transport analysis. The solution algorithms are often classified into three families [PERW15, FH95], depending on how the solution is represented.

The first family includes *link-based* algorithms, which explore the space of link flows, i.e., the solution is represented by variables f_e denoting the flow on each link e in the road network. The prototypical method among these approaches is the *Frank-Wolfe (FW) algorithm* [FW56], a feasible-direction method for solving convex optimization problems with linear constraints. Starting at an initial solution, the FW algorithm repeatedly generates a feasible direction of descent, and shifts the current solution along the descent direction. The algorithm terminates when some stopping criterion is met. Other link-based algorithms are variants of the FW algorithm, such as the *conjugate FW algorithm* and the *biconjugate FW algorithm* [ML13]. These methods generate better descent directions, by taking into account the directions from the last and last two iterations, respectively.

The second family includes *path-based* algorithms, which explore the space of path flows. The solution therefore is represented by variables F_k denoting the flow on each simple path k in the road network. For each OD pair p, path-based algorithms maintain a set K_p^+ of promising paths between the origin location and destination location. In each iteration, each OD pair p is processed in succession. First, K_p^+ is updated by removing unpromising paths (paths having no flow in the current solution) and inserting new promising paths (paths being cheaper in the current solution than any path in K_p^+). Then, K_p^+ is equilibrated, i.e., flow is shifted between the paths in K_p^+ to equalize their costs. Path-based algorithms differ in how they equilibrate K_p^+. The *PE algorithm* [Daf68, FH95] equalizes the costs of the cheapest and costliest path in K_p^+. The *GP algorithm* [JTPR94] distributes flow to the cheapest path from all other paths in K_p^+. The *PG algorithm* [FCF09] shifts flow from paths costlier than average to paths cheaper than average. Similarly, the *ISP algorithm* [KP11] divides the paths in K_p^+ into two parts, one that contains paths whose cost exceeds a certain threshold, and one that contains all other paths. Flow is distributed from paths in the first part to paths in the second one.

The third family includes *bush-based* algorithms, which explore the space of origin flows, where the solution is represented by variables f_{eo} denoting the flow on link e that originates at origin o. While path-based algorithms maintain a set K_p^+ of promising simple paths for each OD pair p, bush-based algorithms maintain a bush B_o for each origin o. A *bush* B_o is a directed acyclic graph that comprises promising paths from the origin o to all destinations. Bush-based algorithms work similar to path-based methods. In each iteration, each origin o is processed in succession. First, B_o is updated by removing links that have no flow in the current solution, and inserting new links that give rise to cheaper paths. Then, B_o is equilibrated. Again, bush-based algorithms differ in how they equilibrate B_o. *Algorithm B* [Dia06, Nie10] examines all destinations d in reverse topological order, equalizing the costs of the cheapest and costliest o–d path in B_o. The *LUCE algorithm* [Gen14] generates a feasible direction of descent by solving a quadratic minimization program for each destination, and shifts flow along the descent direction. Some authors, e.g., Perederieieva et al. [PERW15], also classify the *TAPAS algorithm* [Bar10] as a bush-based method. Although TAPAS does not maintain bushes, the solution is also represented by origin flows.

The variety of algorithms provide different trade-offs between convergence rate, space requirements and implementation complexity, and all families are implemented in commercial software. In this chapter, we build upon link-based algorithms (more precisely, the CFW algorithm), due to their simplicity and low space consumption. Moreover, although all families require a large number of shortest-path computations, the amount of work for everything else done during link-based algorithms is particularly negligible. Hence, they benefit the most from advances in route planning.

The past decade has seen intense research on speedup techniques [Bas+16] for Dijkstra's algorithm [Dij59], which rely on a slow preprocessing phase to enable fast queries. One of the most prominent and versatile techniques among these are contraction hierarchies (CHs) [GSSV12], which exploit the inherent hierarchy of the network. A fairly recent development in the area of route planning are customizable speedup techniques [DGPW17, DSW16, EP13], which split preprocessing into a slow metric-independent part, taking only the graph structure into account, and a fast metric-dependent part (the *customization*), incorporating new edge costs (the *metric*). Customizable route planning (CRP) [DGPW17] and customizable contraction hierarchies (CCHs) [DSW16] are the most prominent among them.

Note that CRP and CCHs do not have the fastest known queries for road networks. Transit node routing (TNR) [BFSS07, ALS13] and hub labeling (HL) [ADGW11] achieve even faster query times. The downside is that their preprocessing is much heavier than the quick and lightweight customization of CRP and CCHs. Therefore, TNR and HL are less suited for a dynamic scenario such as traffic assignment, where the edge costs change quite frequently due to flow shifts.

Despite the utmost importance of shortest-path computations for traffic assignment algorithms, there seems to be only a single paper [LS11] that solves the traffic assignment problem using a state-of-the-art shortest-path algorithm (standard CHs in their case). Even recent experimental studies, e.g. [PERW15], resort to the 50-year-old A* algorithm [HNR68] to compute shortest paths.

Our Contribution. The contribution of this chapter is twofold. First, we accelerate the state of the art in the area of traffic assignment. On our main benchmark instance, our procedure building upon CCHs gains a speedup of 37. This is more than three orders of magnitude faster than the Dijkstra-based baseline. However, the optimizations we propose to achieve this are also independent contributions, not restricted to the traffic assignment problem, but generally applicable to CCHs (and thus a wide variety of shortest-path problems [Bas+16]). Our three main optimizations are as follows: (1) We thoroughly reengineer the CCH customization phase, obtaining substantial speedups for both the single- and multi-threaded versions. We especially focus on the third customization subphase, which received less attention in the original CCH publication. (2) Currently, there are two CCH query algorithms, one based on Dijkstra's algorithm and one based on elimination trees (a structure encoding the search space of each vertex). We carefully engineer the elimination tree search, providing a unified query algorithm that combines good local-query with good global-query performance. (3) We introduce a *centralized* elimination tree search for computing batched point-to-point shortest paths fast. While there is a large amount of work on one-to-all, one-to-many, many-to-many, and point-of-interest queries [DGNW13, DGW11, EP14, EPV15, Kno+07, DW15], we are the first that accelerate batched *point-to-point* shortest paths. All optimizations are extensively experimentally evaluated using solely real-world data, whereas previous work fell back on synthetic origin-destination pairs [LS11].

Outline. This chapter is organized as follows. Section 5.2 provides a precise definition of the traffic assignment problem and briefly reviews the Frank-Wolfe method. Section 5.3 shows how to incorporate customizable contraction hierarchies into the Frank-Wolfe algorithm. Section 5.4 describes the original CCH customization and shows how to accelerate it. Section 5.5 discusses the original elimination tree search and our improved variant. Section 5.6 describes our optimizations for *batched* point-to-point shortest paths. Section 5.7 presents an extensive experimental evaluation of our traffic assignment procedures, and also evaluates our engineered customization and elimination tree search on its own on a well-known benchmark instance. Section 5.8 concludes the chapter with final remarks.

5.2 Preliminaries

We now formally define the traffic assignment problem and briefly review the main algorithms we build upon. First, we describe the Frank-Wolfe algorithm. Then, we discuss how previous work accelerated Frank-Wolfe by replacing Dijkstra's shortest-path algorithm with contraction hierarchies.

Note that we treat a road network as a directed graph $G = (V, E)$ where vertices represent intersections and edges represent road segments. In the literature on the traffic assignment problem, edges are also called *links*. We use these two terms as synonyms. Each edge $(u, v) \in E$ has a constant nonnegative cost representing the travel time between u and v under a certain fixed flow pattern.

Mathematical Program. The traffic assignment problem has been formalized by Beckmann et al. [BMW56] as a mathematical program, known as *Beckmann's transformation*, whose solution is the equilibrium flow pattern we are looking for. It is a convex minimization program with linear constraints.

Before we formulate the mathematical program known as Beckmann's transformation, it is helpful to introduce some notation and terminology. We denote by P the set of OD pairs, by K_p the set of simple paths between OD pair $p \in P$, and by t_p the number of trips between p during the period of analysis. Let f_e be the flow on link $e \in E$, and $c_e(f_e)$ the cost function of e. The latter maps the flow on a link into a cost. The flow on path k is denoted by F_k. The graph structure of the urban road network is given by the indicator variable

$$\delta_e^k := \begin{cases} 1 & \text{if link } e \text{ belongs to path } k, \\ 0 & \text{otherwise.} \end{cases}$$

The equilibrium flow pattern can be obtained by solving the following convex minimization program with linear constraints:

$$\min z(f) := \sum_{e \in E} \int_0^{f_e} c_e(x)\, dx \tag{5.1a}$$

subject to

$$\sum_{k \in K_p} f_k = t_p \qquad \forall\, p \in P \tag{5.1b}$$

$$F_k \geq 0 \qquad \forall\, k \in K_p, p \in P \tag{5.1c}$$

$$f_e = \sum_{p \in P} \sum_{k \in K_p} \delta_e^k F_k \qquad \forall\, e \in E \tag{5.1d}$$

Note that the objective function (5.1a) has no natural interpretation, but is merely a mathematical construct. The flow conservation constraints (5.1b) guarantee that all trips are assigned onto the road network. The nonnegativity constraints (5.1c) are due to physical requirements (there is no negative flow). The graph structure enters the mathematical program through constraints (5.1d).

Solution Algorithm. As argued in Section 5.1, this chapter builds upon the CFW algorithm, a variant of the link-based FW algorithm. Starting at an initial solution, the FW algorithm repeatedly generates a feasible direction of descent, and advances by an optimal step size along the descent direction. See Figure 5.1 for an illustration. An important subroutine of the FW algorithm is the *all-or-nothing (AON) assignment* procedure, which processes each OD pair in succession and assigns one flow unit to each link on the shortest path from the origin to the destination. In the simplest case, shortest paths are computed with Dijkstra's algorithm.

Recall that FW represents the solution by the link flows f_e. Let $f^i = (f_1^i, \ldots, f_{|E|}^i)$ be the link flows at the beginning of iteration i. The initial solution f^1 is generated by an AON assignment based on free-flow link costs. In each iteration i, the FW algorithm performs the following steps: (1) Update link costs based on the current link flows, i.e., set the cost of each link e to $c_e(f_e^i)$. (2) Perform an AON assignment

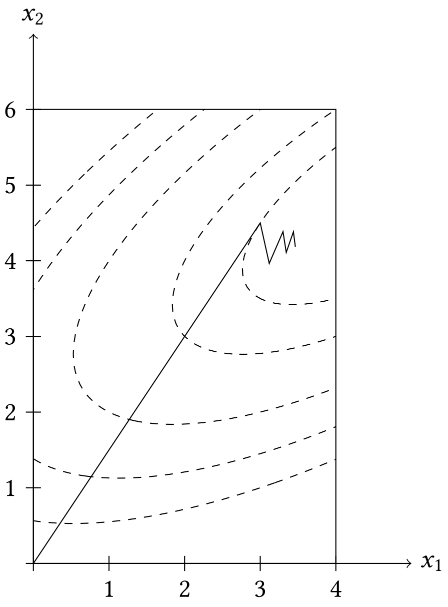


Figure 5.1: Execution of the FW algorithm in two-dimensional space. The dashed lines indicate contour lines of the convex objective function that is to be minimized, and the rectangle indicates the feasible region. The solution is represented by the two variables x_1 and x_2. Starting at the initial feasible solution $(0, 0)$, the algorithm moves in each iteration closer to the solution, and starts zigzagging in its vicinity.

based on the current link costs, yielding a link flow vector y^i, and let the descent direction d^i be $y^i - f^i$. (3) Perform a *line search* in the descent direction, i.e., along the line segment between f^i and y^i, which determines how far the current solution must be moved along the direction of descent. (4) Move the current solution along the descent direction, i.e., set $f^{i+1} = f^i + \lambda^i d^i$, where λ^i is the step size found by the line search. (5) Check if the convergence criterion is met, and stop or go to step (1).

Note that the line search is nothing more than a one-dimensional minimization of a convex function, which we implement using recursive bisection [She85]. The CFW algorithm, which we build upon, improves the second step by choosing as descent direction a certain convex combination of the FW direction (generated by an AON assignment based on the current link costs as described above) and the descent direction from the previous iteration. For further details, we refer the interested reader to Mitradjieva and Lindberg [ML13].

Link Cost Function. Due to congestion, the time or cost to travel along a link is not constant, but increases with the traffic flow on it. The relation between cost and flow is expressed by link cost functions, which are required to be monotonically increasing so that the objective function Equation (5.1a) becomes convex [She85]. There is a variety of different link cost functions, such as the BPR function [Bur64], the Davidson function [Dav66], and the Lohse function [PTV14]. Our benchmark instances, taken from production systems, use the BPR function, which is defined as

$$c_e(f_e) = c_e^0 \left(1 + \alpha_e \left(\frac{f_e}{f_e^{\max}}\right)^{\beta_e}\right), \tag{5.2}$$

where c_e^0 is the free-flow cost of link e, $f_e^{\max}$ is the capacity of link e, and α_e and β_e are model parameters (set to 1 and 2, respectively, for each link e).

Convergence Criterion. There is a large number of convergence criteria, such as the change in link flows [She85] and simply the number of iterations [FH95]. However, the most common criterion in both research papers and practice [PERW15, ML13, SBR06, Dia06, FCF09, Gen14, ZYC11] is the *relative gap* [BRB04]. Recall that at equilibrium, each driver takes a shortest path between its endpoints. Before reaching an equilibrium, the total cost of currently used paths is therefore larger than the total cost of current shortest paths. Intuitively, the relative gap is the difference between the total cost of currently used paths and the total cost of current shortest paths. More precisely, the relative gap after iteration i is defined as

$$\frac{\sum_{e \in E} f_e^i \cdot c_e(f_e^i) - \sum_{p \in P} t_p \cdot \mu_p(f^i)}{\sum_{e \in E} f_e^i \cdot c_e(f_e^i)}, \tag{5.3}$$

where $\mu_p(f^i)$ is the shortest-path cost between OD pair p based on the link flows f^i. Obviously, the relative gap ranges between 0 and 1. At the user equilibrium, we have $\sum_{e \in E} f_e^i \cdot c_e(f_e^i) = \sum_{p \in P} t_p \cdot \mu_p(f^i)$, and thus the relative gap becomes zero. In our experiments, we stop the iterative procedure when the relative gap drops below the threshold of 10^{-4}, as recommended by Boyce et al. [BRB04].

Acceleration by Contraction Hierarchies. The shortest-path computations in the direction-finding step are by far the most time-consuming part of the FW algorithm. Carrying them out with CHs (instead of Dijkstra's algorithm) accelerates traffic assignments by more than an order of magnitude [LS11]. Since the cost of each edge changes between two iterations, the CH is rebuilt from scratch in each iteration. Queries do not unpack shortcuts but assign one flow unit to each (shortcut) edge on the packed path. After computing all paths, the shortcuts are unpacked in top-down fashion, with cumulated flow units propagated from shortcut to original edges.

5.3 Traffic Assignment Using Customization

Previous work [LS11] applying speedup techniques to the traffic assignment problem observed that the performance bottleneck depends on the traffic scenario under study. For short or off-peak periods, where there are few OD pairs, preprocessing dominates the total running time. When there are many OD pairs, as for long or peak periods, queries become the main performance bottleneck. Therefore, it is necessary to accelerate both preprocessing and query times.

To decrease the preprocessing time, we apply the concept of customization to the traffic assignment problem. Customizable speedup techniques [DGPW17, DSW16, EP13] split preprocessing into a metric-independent part, taking only the graph structure into account, and a metric-dependent part (the *customization*), incorporating new edge costs (the *metric*). Since the graph topology does not change in all iterations of the traffic assignment procedure and only edge costs change, it suffices to run a fast customization in each iteration instead of an entire preprocessing.

The two most prominent and versatile customizable speedup techniques are CRP and CCHs, which yield different tradeoffs between customization and query time. While CRP achieves slightly smaller customization times, CCHs have somewhat better query times [DSW16]. We choose to build our traffic assignment procedure upon CCHs mainly for two reasons. First, as our experiments will show, even on our benchmark instance having the smallest number of OD pairs the total running time is dominated by queries. Therefore, it makes sense to trade customization time for query time. Second, after each query, we have to traverse the computed path to assign one flow unit to each edge on the path. Both CRP and CCHs would allow to collect flow units on the packed paths containing shortcuts, with flow values propagated from shortcut to original edges after computing all paths (in the spirit of [LS11]). However, the packed paths computed by CCHs usually contain less edges than the ones computed by CRP, and thus the overhead per query is less when using CCHs. Hence, we prefer to use CCHs within our traffic assignment procedure.

Again note that CRP and CCHs do not have the fastest known queries for road networks. However, the preprocessing of techniques such as TNR and HL, which is much heavier than the lightweight customization of CRP and CCHs, make these techniques less suited for a dynamic scenario such as traffic assignment (where the edge costs change quite frequently due to flow shifts). For example, on our main benchmark instance (a typical morning peak), the CH-based traffic assignment algorithm by Luxen and Sanders [LS11] takes a significant fraction of the total time (36.1 out of 90.9 seconds) for preprocessing; see also Section 5.7. TNR and HL preprocessing take multiple times longer than CH preprocessing [Bas+16], and therefore TNR- and HL-based traffic assignments would even be slower than the existing state-of-the-art CH-based traffic assignment.

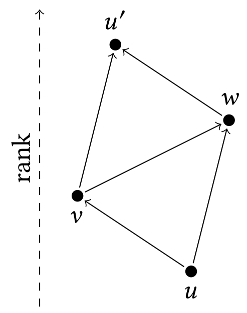


Figure 5.2: Triangles in the customizable contraction hierarchy. We say that $\langle u, v, w\rangle$ is a lower triangle of (v, w), and $\langle v, w, u'\rangle$ is an upper triangle of (v, w).

Switching from weighted CHs to customizable CHs decreases the total time to compute an equilibrium flow pattern on our main benchmark instance from 90.9 to 41.5 seconds. However, to achieve running times enabling interactive applications, we must engineer all aspects of CCHs. In particular, we must accelerate and fully parallelize customization. Moreover, we introduce a unified query algorithm, and a centralized search algorithm for batched point-to-point shortest paths. The remainder of this chapter discusses each optimization in turn.

5.4 Accelerating Customization

CCH customization can be divided into three subphases. First, *basic customization* computes costs for all edges in the hierarchy. It is the only subphase required and enough to ensure that queries are correct. However, after basic customization some edges in the hierarchy can have suboptimal costs that are not the same as the shortest-path distances between their endpoints. The second (optional) subphase, *perfect customization*, sets the cost of each edge to the distance between its endpoints. The third (again optional) subphase constructs a standard CH having the smallest possible number of edges for the given contraction order, by removing each edge whose cost was improved by the perfect customization algorithm.

The fundamental operation of both basic and perfect customization is enumerating triangles. A triangle is a set of three pairwise adjacent vertices. Consider a triangle $\langle u, v, w\rangle$, where u is the lowest-ranked vertex and w the highest-ranked vertex (see Figure 5.2). We call $\langle u, v, w\rangle$ a *lower triangle* of (v, w), an *intermediate triangle* of (u, w), and an *upper triangle* of (u, v). To efficiently support enumerating triangles, we store an upward adjacency array $G^{\uparrow} = (V, E^{\uparrow})$ and a downward adjacency array $G^{\downarrow} = (V, E^{\downarrow})$ of the hierarchy. In the former, each vertex stores its incident edges leading to neighbors with higher rank. This is the standard CH representation in the literature [GSSV12, DGNW13, DGW11]. In the latter, each vertex stores its incident edges leading to neighbors with lower rank.

If the incident edges of each vertex are sorted by neighbor ID, enumerating the upper triangles of (u, v) can be done by a coordinated sweep over all edges $(u, u') \in E^{\uparrow}$ and $(v, v') \in E^{\uparrow}$. More precisely, we maintain indices i_u and i_v, initialized to the indices of the first edges in $E^{\uparrow}$ out of u and v, respectively. Let (u, u') be the edge with index i_u, and let (v, v') be the edge with index i_v. In each iteration, we compare the vertices u' and v'. If these IDs are equal, then we found a new upper triangle, and increment both i_u and i_v. If the IDs differ, then we increment either i_u (if $u' < v'$) or i_v (if $u' > v'$). We stop when either i_u or i_v points one past the last edge out of u or v, respectively. Enumerating intermediate and lower triangles works analogously.

During basic customization, we process the edges in bottom-up fashion, ordered increasingly by the rank of the lower-ranked endpoint. For each edge (v, w), we enumerate all lower triangles $\langle u, v, w\rangle$ and set $\ell(v, w) = \min\{\ell(v, w), \ell(u, v) + \ell(u, w)\}$, where $\ell(v, w)$ denotes the cost of (v, w).[14] Analogously, during perfect customization, we process the edges in top-down fashion, ordered decreasingly by the rank of the lower-ranked endpoint. For each edge (v, w), we enumerate all intermediate triangles and all upper triangles, and again try to update the cost of (v, w). If multiple CPU cores are available, then all edges departing on the same level can be processed in parallel (during both basic and perfect customization).

In the following, we propose several optimization techniques that accelerate the customization subphases described above. We start by carefully reengineering the single-threaded versions and discuss the multithreaded versions in Section 5.4.2.

5.4.1 Sequential Execution

The crucial observation is that enumerating upper triangles is cheaper than enumerating lower or intermediate triangles. The main reason for this is that the coordinated sweeps perform fewer iterations, due to the distribution of the vertex degrees in $G^{\uparrow}$ and $G^{\downarrow}$. Note that the number of iterations performed by the coordinated sweeps (without stopping) when enumerating lower triangles during customization is

$$\sum_{(u,v) \in E} (\deg_{G^{\downarrow}}(u) + \deg_{G^{\downarrow}}(v)) = \sum_{u \in V} (\deg_{G^{\uparrow}}(u) \deg_{G^{\downarrow}}(u) + \deg_{G^{\downarrow}}(u)^2),$$

where $\deg_G(v)$ is the degree of v in G. Similarly, the number of iterations performed by the coordinated sweeps when enumerating upper triangles is

$$\sum_{(u,v) \in E} (\deg_{G^{\uparrow}}(u) + \deg_{G^{\uparrow}}(v)) = \sum_{u \in V} (\deg_{G^{\uparrow}}(u) \deg_{G^{\downarrow}}(u) + \deg_{G^{\uparrow}}(u)^2).$$

[14] For simplicity, we assume that the edges have the same cost in both directions. To support different costs in each direction, our implementation maintains an upward cost $\ell^{\uparrow}(v, w)$ and a downward cost $\ell^{\downarrow}(v, w)$ for all edges (v, w) in the hierarchy, as described in [DSW16].

Note that $\sum_{v \in V} \deg_{G^\uparrow}(v)$ must be equal to $\sum_{v \in V} \deg_{G^\downarrow}(v)$, since $G^\uparrow$ and $G^\downarrow$ represent the same hierarchy. However, the values of $\sum_{v \in V} \deg_{G^\uparrow}(v)^2$ and $\sum_{v \in V} \deg_{G^\downarrow}(v)^2$ also depend on the distribution of the degrees in $G^\uparrow$ and $G^\downarrow$, respectively. A uniform distribution minimizes the sums. We observe that the degrees in $G^\downarrow$ are more widely dispersed than the ones in $G^\uparrow$, and therefore $\sum_{v \in V} \deg_{G^\uparrow}(v)^2 < \sum_{v \in V} \deg_{G^\downarrow}(v)^2$. Indeed, the coordinated sweeps perform half as many iterations when enumerating upper triangles instead of lower triangles.

For an intuition of why vertex degrees are more widely dispersed in $G^\downarrow$, consider the vertices of degree zero. In $G^\uparrow$, a vertex has degree zero if and only if it is in the highest level. Similarly, in $G^\downarrow$, a vertex has degree zero if and only if it is in the lowest level. While roughly 40 % of all vertices in the hierarchy are in the lowest level, there can only be a single vertex in the highest level. Since the total degree of all vertices is fixed, the degree of vertices higher up in the hierarchy must be larger in $G^\downarrow$, explaining the wider dispersion of vertex degrees.

We can reduce the number of iterations even further. When enumerating the upper triangles of (u, v), we initialize the index i_u to point to the edge in $E^\uparrow$ immediately following (u, v) (and not to the first edge in $E^\uparrow$ out of u). Because the incident edges of u are sorted by neighbor ID, and lower-ranked vertices have lower IDs, the edges $(u, u') \in E^\uparrow$ preceding (u, v) cannot induce upper triangles of (u, v). Therefore, we can skip them during the coordinated sweeps. Note that this optimization does not carry over to enumerating lower triangles, but could also be applied when enumerating intermediate triangles during customization.

Based on this observation, we propose the following basic customization algorithm. We use the same bottom-up processing order as before. When processing an edge (v, w), however, we do not enumerate all *lower* triangles $\langle u, v, w\rangle$ and set $\ell(v, w) = \min\{\ell(v, w), \ell(u, v) + \ell(u, w)\}$, but enumerate all *upper* triangles $\langle v, w, u'\rangle$ and set $\ell(w, u') = \min\{\ell(w, u'), \ell(v, w) + \ell(v, u')\}$. Since each triangle is at the same time a lower, an intermediate and an upper triangle, we ultimately enumerate the same triangles, even though in a slightly different order. To preserve correctness, the costs of both (v, w) and (v, u') must be final at the time we try to update the cost of (w, u'). This is the case since the cost of (v, w) can only be updated by an upper triangle of an edge (u, v), where u has lower rank than v. The bottom-up processing order ensures that each such edge is processed before (v, w), and therefore the cost of (v, w) is final. The same argument applies to the cost of (v, u'), which is enough to preserve correctness of our basic customization algorithm.

Moreover, we propose the following perfect customization algorithm. We again keep the same top-down order as before. For each edge (v, w), we enumerate all upper triangles $\langle v, w, u'\rangle$ and set $\ell(v, w) = \min\{\ell(v, w), \ell(v, u') + \ell(w, u')\}$, as in the original algorithm. In addition, we also set $\ell(v, u') = \min\{\ell(v, u'), \ell(v, w) + \ell(w, u')\}$, which avoids enumerating intermediate triangles explicitly. Note that our variant differs from the original variant only in the order in which we consider the upper and intermediate triangles of the incident edges of each vertex. Correctness follows immediately from the original proof in [DSW16], since the proof works for any order.

To summarize, both the basic and perfect customization algorithm enumerate only upper triangles. Therefore, another advantage is that both algorithms do not access the downward adjacency array $G^{\downarrow}$, improving cache efficiency. On the European road network using travel times as the length function, we decrease the sequential time for basic customization from 10.9 to 5.6 seconds, and for perfect customization even from 22.1 to 6.5 seconds (see Section 5.7.2).

5.4.2 Parallel Execution

The optimizations discussed above can also be used in the multithreaded versions. Depending on the approach for parallelization, however, the edge costs have to be updated atomically, which can be implemented lock-free on the x86 microarchitecture. Building upon our engineered single-threaded customization algorithms, we propose two optimizations for the multithreaded versions in this section, starting with an alternative, more scalable approach for parallelization.

Task-Based Parallelism. The original CCH publication [DSW16] suggests processing all edges departing on the same level in parallel. This simple loop-based parallelism suffers from three drawbacks. First, the edges processed by each thread are not consecutive in memory, which leads to worse locality and more cache misses. Second, we require a synchronization step after each level, which has some overhead. Third, when building upon our engineered customization algorithms, we have to use atomic operations. We compensate for these drawbacks by proposing an alternative approach for parallelization based on the separator decomposition of the hierarchy.

The idea is as follows. Since the removal of the top-level separator decomposes the hierarchy into several subgraphs, each of these components can be handled in parallel. We propose using task-based parallelism provided by OpenMP [PST17] to implement this idea. A task is responsible for processing all edges in a subgraph of the hierarchy. Initially, we generate a task that is responsible for the entire hierarchy. The execution of a task differs slightly between basic and perfect customization.

We first address basic customization. Consider a task that is responsible for a subgraph G. The removal of the top-level separator of G decomposes G into several components G_i. If the size of G is below a given threshold, then we process all edges in G in the order in which they are stored in memory. Since the edges are sorted by tail ID, and lower-ranked vertices have lower IDs, this is a valid processing order. If the size of G exceeds the threshold, then we generate a child task for each G_i, wait on the completion of all child tasks, and then process the incident edges of the separator vertices. We use $n/(\tau \cdot c)$ as our threshold, where n is the size of the entire customizable contraction hierarchy, c is the number of cores available, and τ is a tuning parameter. Increasing τ leads to better load balancing but also to larger tasking and synchronization overhead. We set $\tau = 32$ in our experiments.

Task-based parallelism is even better suited to parallelizing perfect customization, since the steps of a task are reversed. First, we process the incident edges of the separator vertices (in reverse order of the layout in memory), and then we generate a child task for each G_i. Therefore, we do not need a single synchronization step. Contrary to basic customization, perfect customization also does not need atomic operations, since there are no concurrent modifications to a single cost.

Note that our approach for parallelization is not only a natural fit for the customization phase, but can also be used to parallelize other algorithms working on a CCH, such as the one-to-all PHAST algorithm [DGNW13]. Since PHAST requires only a top-down sweep (and no bottom-up sweep), it can be implemented synchronization-free, improving scaling on multi-core machines.

Building the Minimum Weighted CH. The original CCH publication [DSW16] gives no details (and no running times) of the implementation of the third customization subphase, the construction of the minimum weighted CH. This is surprising, since the third subphase actually takes 47–56 % of the sequential customization time in our experiments. Therefore, when parallelizing only basic and perfect customization, the speedup for the entire customization phase is less than a factor of two. In the following, we present a parallelization technique for the third subphase.

During perfect customization, we maintain one bit per edge in the hierarchy. All bits are initially set. Whenever we improve the cost of an edge, we clear its bit. Maintaining these bits has no measurable performance penalty. The third subphase then forms the prefix sums of the bits, which gives us the location of each necessary edge in the minimum weighted CH. This allows us to fill the new adjacency array in parallel, taking advantage of multiple cores.

Since computing prefix sums in parallel can overload the memory system [Sin10], we use an approach inspired by [BPZ07]. We precompute and store the location of every k'-th edge in the customizable contraction hierarchy. To compute the location of the edge with index i, we look up the location of the edge with the largest precomputed index $j \leq i$, and count the number of necessary edges with an index between j and $i - 1$. Increasing k' reduces precomputation time, but also leads to slower location lookups. We use $k' = 4$ in our experiments.

For setting up the path-unpacking data structure [GSSV12] of the minimum weighted CH, we must know the lower triangle that created each necessary shortcut. At first glance, it is tempting to record this information during basic customization, to have it available in the third subphase. However, remember that concurrent modifications to a single edge can arise in the multi-threaded version of our engineered basic customization. While we can use atomic operations to modify edge costs, maintaining unpacking information would require locks, rendering this approach impractical. One possibility is to switch back to the original basic customization algorithm, where there are no concurrent modifications. However, this would double the running time.

The fastest approach if multiple cores are available is to stick to our engineered basic customization algorithm (without recording unpacking information), and explicitly enumerate lower triangles of each necessary edge (v, w) during the third subphase. We do not need to enumerate all lower triangles, though, but can stop at the first triangle $\langle u, v, w\rangle$ such that $\ell(v, w) = \ell(u, v) + \ell(u, w)$. Note that we can handle all edges in parallel, and do not need to obey a certain (bottom-up or top-down) order, as in the first two subphases of the customization.

5.5 Accelerating Elimination Tree Searches

While Dibbelt et al. [DSW16] observe that the CCH query algorithm based on elimination trees achieves fastest query times for random queries (which tend to be long-range), it is slower by more than an order of magnitude than the Dijkstra-based query algorithm for local queries. However, the input of the traffic assignment problem consists of both local and long-range OD pairs, requiring a query algorithm that can handle both types of queries well. Therefore, we review and carefully engineer the elimination tree search in this section. The result is a fast, unified CCH query algorithm, combining good performance for both local and long-range queries.

Given a source s and a target t, the original elimination tree search [DSW16] works in five phases. First, we compute the lowest common ancestor (LCA) x of s and t in the elimination tree T rooted at the highest-ranked vertex r. This is done by enumerating the ancestors of s and t in increasing rank order until a common ancestor is found. Second, we revisit each vertex v on the s–x path in T, relaxing all *outgoing upward* edges of v. Third, we do the same for each vertex v on the t–x path in T, relaxing all *incoming downward* edges of v. Fourth, we visit each vertex v on the x–r path in T, relaxing all outgoing upward *and* incoming downward edges of v. Moreover, we check at each such vertex v whether the s–t path via v improves the tentative s–t distance. Fifth, we again revisit each vertex on the s–r and t–r path to reset its distance labels for the next computation.

Phase Reduction. Our first optimization reduces the number of phases of the elimination tree search. We refrain from computing the LCA first, and then visiting each vertex from the source (target) to the LCA again. Instead, while we enumerate the ancestors of s and t in the same fashion as before, we immediately relax their edges. Moreover, we observe that the resetting phase is unnecessary. After relaxing the edges of a vertex, its distance labels are never read again. Therefore, we can safely reset them to ∞ right after relaxing the edges, avoiding the fifth phase completely. Note that we cannot reset parent pointers, since they may be needed afterwards. However, this is not an issue, because resetting the distance labels suffices to decide whether a vertex has been visited before during the next query. With this optimization, each vertex is visited at most once, instead of up to three times as before.

Pruning Rule. The basic elimination tree search does not make use of pruning. Only when combined with the full (three-subphase) customization, Dibbelt et al. [DSW16] employ the following basic pruning rule. Due to the removal of superfluous edges, a vertex may have an ancestor in the elimination tree that is not in its perfect search space. Such an ancestor will have a distance label of ∞ when visited during the search. To accelerate queries, Dibbelt et al. do not relax the edges of a vertex with a distance label of ∞. We observe that a stricter pruning rule is possible. We do not relax edges of a vertex whose distance label exceeds the current tentative shortest-path distance, since those edges cannot possibly contribute to a shorter path. Despite its simplicity, this optimization accelerates the search quite drastically, by a factor of 13 for short-range queries. Moreover, our pruning rule does not require the full customization, but can also be combined with the basic customization, without building a weighted CH having the smallest possible number of edges for the given contraction order.

5.6 Accelerating Batched Shortest Paths

The Frank-Wolfe method requires computing multiple point-to-point shortest paths in each iteration. The obvious approach is to perform an independent elimination tree search for each OD-pair. However, we can do better by explicitly adapting CCHs to compute batched point-to-point shortest paths.

5.6.1 Reordering OD Pairs to Exploit Locality

Previous work processed the OD pairs in no particular order. However, reordering the OD pairs so that pairs with similar forward and reverse search spaces tend to be processed in succession improves memory locality and cache efficiency. We call two search spaces similar if their symmetric difference is small. Note that the size of the symmetric difference between the search spaces of u and v is equal to the distance between u and v in the elimination tree. Hence, we partition the elimination tree into as few cells with bounded diameter as possible, assign IDs to cells according to the order in which they are reached during a DFS [SMDD19] on the elimination tree, and reorder OD pairs lexicographically by the origin and destination cells.

We use a simple yet optimal greedy algorithm to partition the elimination tree into as few cells with diameter at most U as possible. Our algorithm repeatedly cuts out a subtree (with diameter at most U) and makes it a cell of its own. To do so, it maintains for each vertex v the height $h(v)$ of the remaining subtree T_v rooted at v, initialized to zero, and processes vertices in ascending rank order. To process v, we examine its children w_i in order of increasing height of T_{w_i}. If $h(v) + 1 + h(w_i) \leq U$, then we set $h(v) = 1 + h(w_i)$. Otherwise, we cut out T_{w_i}, making it a cell of its own. We use $U = 40$ in our experiments in Section 5.7.

Theorem 5.1. *Our algorithm produces a partition into cells with diameter at most U.*

Proof. Since our algorithm only makes subtrees of the elimination tree cells of their own, all cells are connected. We need to show that a longest path within a cell is of length at most U. Let $(s, \dots, s', v, t', \dots, t)$ be such a path, where v is the maximum-rank vertex on the path. The s–s' subpath is not longer than $h(s')$, and the t'–t subpath is not longer than $h(t')$. Hence, the s–t path is of length at most $h(s') + 2 + h(t')$. Assume, without loss of generality, that when our algorithm processed v, it examined child s' before child t'. Since v and s' are in the same cell, $h(v)$ was set to at least $1 + h(s')$ before child t' was examined. When examining child t', since v and t' are in the same cell, we had $h(s') + 2 + h(t') \leq h(v) + 1 + h(t') \leq U$. Hence, the s–t path is not longer than U, which completes the proof. □

Theorem 5.2. *Our greedy algorithm produces an optimal solution, i.e., it finds a partition of the elimination tree with a minimum number of cells.*

Proof. We use induction on the number k of cells in an optimal solution. For $k = 1$ the statement is certainly true: if there is no path of length larger than U, then our greedy algorithm does not cut out any subtrees.

Now let $k > 1$. We assume as our induction hypothesis that the statement is true for $k - 1$, and prove it for k. Since an optimal solution has at least two cells, the elimination tree T contains a vertex v with d children w_i such that the subtree T_v rooted at v has diameter larger than U, and all T_{w_i} have diameter at most U. Assume, without loss of generality, that the children w_i are ordered by increasing height of T_{w_i}; see also Figure 5.3. We claim that there is an optimal solution in which T_{w_d} is a cell of its own. Since an optimal solution for T has k cells, an optimal solution for $T \setminus T_{w_d}$ has $k - 1$ cells. When our greedy algorithm processes v, it cuts out T_{w_d}, making it a cell of its own. By the induction hypothesis, it produces an optimal solution for the subproblem $T \setminus T_{w_d}$. This completes the induction step.

It remains to prove that there is an optimal solution in which T_{w_d} is a cell of its own. Since T_v has diameter larger than U, there is at least one cut edge (an edge with

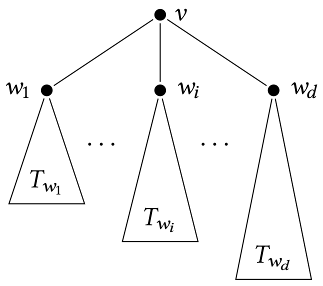


Figure 5.3: The subtree T_v.

endpoints in different cells) in T_v. Note that k' cut edges partition a tree into $k' + 1$ cells. Hence, exchanging a cut edge for another edge does not increase the number of cells. Now assume that all cut edges in T_v are incident on v. (In the case where there is one in T_{w_i}, then we exchange it for $\{v, w_i\}$ without creating a cell with diameter larger than U, since T_{w_i} has diameter at most U by assumption.) If $\{v, w_d\}$ is a cut edge, we are done. If not, then we exchange any cut edge for $\{v, w_d\}$, again without creating a cell with diameter larger than U, completing the proof. □

5.6.2 Centralized Searches

Instead of processing similar OD pairs *in succession*, processing them *at once* in a single search achieves additional speedup. The idea of bundling together multiple shortest-path computations was introduced in [HKMS09] and later used in [DGPW17, DGNW13, DGW11, BD09, Yan10]. However, in each case, centralized searches were only used for one-to-all and one-to-many queries, and only combined with plain Dijkstra (and Bellman-Ford in [DGPW17]). Even (R)PHAST [DGNW13, DGW11] performs the CH searches sequentially, and bundles only the linear sweeps. We extend the idea to point-to-point queries, and combine it with CH searches, including appropriate stopping and pruning criteria.

The basic idea of centralized searches is to maintain k distance labels for each vertex u, laid out consecutively in memory. The i-th distance label represents the tentative distance from the i-th source to u. Initially, the i-th distance label of the i-th source is set to zero, and all remaining $kn - k$ distance labels to ∞. When relaxing an edge (u, v), we try to improve all k distance labels of v at once. The number of simultaneous shortest-path computations k is a tuning parameter. Increasing k allows us to compute more shortest paths at once, however, it also evicts useful data from caches. Setting $k = 32$ works well for all scenarios we study.

Dijkstra-based Search. Initially, we insert all k sources (targets) into the queue of the forward (reverse) search. As keys, we can use many different values, for example the minimum over all k entries in a distance label or the minimum over the entries that were improved by the last edge relaxation. However, a preliminary experiment showed that using the minimum over *all* k entries clearly dominates the others, which is consistent with previous observations on related techniques [HKMS09]. We can stop the forward (reverse) search as soon as its queue is empty or the smallest queue entry exceeds the maximum over all k tentative shortest-path distances. When using stall-on-demand [GSSV12], we prune the forward (reverse) search at a vertex v when each of the k distance labels of v is suboptimal.

Elimination Tree Search. Computing multiple shortest paths in a single elimination tree search is more involved, since it uses no queues that can easily be initialized with multiple sources and targets. Instead, we equip the forward and reverse search each with a tournament tree (often also called selection tree or loser tree) [Knu98]. Suppose we have k sorted sequences that are to be merged into a single output sequence, as in k-way mergesort. In order to do so, we have to repeatedly find the smallest from the leading elements in the k sequences. This can be done very efficiently with the help of a tournament tree.

In our case, the k sorted sequences are the paths in the elimination tree T from each source (target) to the root, and the single output sequence is the order in which we want to process the vertices during the search. More precisely, we initialize the tournament tree with all k sources (targets). Then, we extract a lowest-ranked vertex from the tournament tree, process it, and insert its parent in T into the tournament tree. We continue with a next-lowest-ranked vertex, until we reach the root of T. Note that in our case, the sequences are implicit, and never stored explicitly.

As soon as two (or more) of the k paths in T converge at a common vertex, there are duplicates in the single output sequence. However, we want to process each vertex at most once. Therefore, whenever two or more paths converge, we block all but one of them, so that only one continues to move through the tournament tree. To do so, we insert for each path to be blocked a vertex with infinite rank into the tournament tree (instead of the next vertex on the path). We know that some paths converged, when we extract the very same vertex several times in succession.

A clear advantage of the centralized elimination tree search is that it retains the *label-setting* property, i.e., each vertex and each edge is processed at most once. In contrast, the centralized Dijkstra-based search is a *label-correcting* algorithm. Note that one centralized elimination tree search is slower than k elimination tree searches by a factor of $\log k$ in O-notation (due to k-way merging), but outperforms them in practice as our experiments will show (see Section 5.7).

5.6.3 Instruction-Level Parallelism

Modern CPUs have special registers and instructions that enable single-instruction multiple-data (SIMD) computations performing basic operations (e.g., additions, subtractions, shifts, compares, and data conversions) on multiple data items simultaneously [Kus14]. We implemented versions of the centralized searches using SSE instructions (working with 128-bit registers), and additionally versions using AVX(2) instructions (manipulating 256-bit registers).

As an example, we describe how an AVX-accelerated edge relaxation (used in Dijkstra-based and elimination tree searches) works, assuming $k = 8$. Since we use 32-bit distance labels, all k labels of a vertex fit in a single 256-bit register. To relax an edge (u, v), we copy all k distance labels of u to an AVX register, and broadcast the edge cost to all elements of another register. Then, we add both registers with a single instruction, and check with an AVX comparison whether any tentative distance improves the corresponding distance label of v. If so, then we compute the packed minimum of the tentative distances and v's distance labels. In the same fashion, we implement the other aspects (stopping and pruning criteria).

5.6.4 Core-Level Parallelism

We now consider how to use core-level parallelism to speed up batched shortest paths. Since the centralized computations are independent from one another, we assign contiguous subsets of OD pairs to distinct cores. We distribute the OD pairs to cores in chunks of size 64. This maintains some locality even between centralized computations. Each core executes a chunk, then requesting another chunk until no chunk remains. Flow units on the (shortcut) edges are cumulated locally and aggregated after computing all shortest paths.

As our experiments in Section 5.7.4 will show, we observe almost perfect speedups for the time spent on shortest-path queries when computing equilibrium flow patterns. Running on 4 cores, the speedup on our largest benchmark instance is 3.8. With all 16 cores available, we see a speedup of 14.3.

5.7 Experiments

Our publicly available code[15] is written in C++14 (using OpenMP for parallelization) and compiled with the GNU compiler 8.2 using optimization level 3. We use 4-heaps [Joh75] as priority queues. To ensure a correct implementation, we make extensive use of assertions (disabled during measurements), and check results against reference implementations such as Dijkstra's algorithm. Our primary benchmark machine, denoted by M3500, runs openSUSE Leap 15.1 (kernel 4.12.14), and has

[15] `https://github.com/vbuchhold/routing-framework`

Table 5.1: Traffic scenarios used for the evaluation of our traffic assignment procedures. We report for each scenario the planning area, the period of analysis, and the number of OD pairs departing within that period.

scenario	planning area	analysis period	OD pairs
S-morn	Stuttgart Region	Tue., 7.30–8.30	248 431
S-even	Stuttgart Region	Tue., 16.30–17.30	280 364
S-day	Stuttgart Region	a whole Tuesday	3 355 442
S-week	Stuttgart Region	a whole week	21 248 278
L-peak	Greater London	a peak hour	468 602

192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.50 Ghz and 8×64 KiB of L1, 8×1 MiB of L2, and 24.75 MiB of shared L3 cache. To ensure comparability of results, we perform some experiments on a secondary machine, denoted by M2600. It also runs openSUSE Leap 15.1 (kernel 4.12.14), and has 64 GiB of DDR3-1600 RAM and two Intel Xeon E5-2670 CPUs, each of which has eight cores clocked at 2.60 Ghz and 8×64 KiB of L1, 8×256 KiB of L2, and 20 MiB of shared L3 cache. Unless otherwise mentioned, we conduct the experiments in this section on our primary machine M3500.

5.7.1 Inputs and Methodology

Our main benchmark instance is the Stuttgart Region [SHP11] in Germany, encompassing more than 2.7 million inhabitants. The experiments are run on the largest strongly connected component consisting of 134 663 vertices and 307 759 edges. While this instance is significantly smaller than road networks studied before for evaluating point-to-point queries [Bas+16], it is the largest available to us that provides real-world capacities and OD pairs, and is still an order of magnitude larger than the instances collected in [BV08], and the instances considered in a recent overview of the state-of-the-art in the area of traffic assignment [PERW15]. Moreover, urban planners are usually interested in assignments on metropolitan areas, not continents.

The instance provides demand data for a whole week. The demand was originally forecasted using mobiTopp [MKV13, MV15], which was calibrated from a household travel survey [VRS11] conducted in 2009/2010. The raw data contains about 51.8 million trips between 1174 traffic zones, encompassing various modes of transportation such as pedestrian, bicycle, public transit, and car. For our experiments, we only consider car trips, and extract four different traffic scenarios, as shown in Table 5.1. We choose a typical morning peak on a working day (Tuesday), the evening peak on a Tuesday, a whole Tuesday, and a whole week. While it may be unrealistic to compute a traffic assignment for a whole week (as the period of analysis would be too inhomogeneous), it shows the scalability of our procedures for tens of millions of OD pairs. We assume the trip endpoints to be uniformly distributed in the traffic zones, and pick for each trip the origin vertex and the destination vertex uniformly at random from its origin and destination zone, respectively.

Besides the Stuttgart Region, we also consider an instance representing the Greater London area, with about 8.2 million inhabitants. We again take the largest strongly connected component, consisting of 45 158 vertices and 101 897 edges. The region is divided into 5692 traffic zones Z, and we are given a fractional demand $t_{ij} \in \mathbb{R}_{\geq 0}$ between each pair (i, j) of traffic zones $i, j \in Z$. We generate $\lfloor \sum_{k,\ell \in Z} t_{k\ell} \rceil$ OD pairs, each as follows. First, we draw the origin zone O from a discrete distribution determined by the probability function $\Pr[O = i] = \sum_{\ell \in Z} t_{i\ell} / \sum_{k,\ell \in Z} t_{k\ell}$. Let i be the value of O. Second, we draw the destination zone D_i from a discrete distribution determined by the probability function $\Pr[D_i = j] = t_{ij} / \sum_{\ell \in Z} t_{i\ell}$. Finally, we pick the origin vertex and the destination vertex uniformly at random from the set of vertices in the origin and destination zone, respectively.

To ensure comparability of our experimental results, we evaluate our engineered customization and elimination tree search on the European road network, which is the standard benchmark instance for point-to-point queries [Bas+16]. It has 18 017 748 vertices and 42 560 275 edges, and was made available by PTV AG for the 9th DIMACS Implementation Challenge [DGJ09].

The CH preprocessing is borrowed from the open-source library RoutingKit[16]. We compute nested dissection orders for CCHs using Inertial Flow [SS15], setting the balance parameter $b = 0.3$. The reported running times do not include partitioning time, as it suffices to partition a network only once, and reuse the resulting order for all traffic assignments on the same network. Partitioning the Stuttgart Region takes less than two seconds (even on a single core). We always use the full (three-subphase) customization approach in combination with CCHs.

5.7.2 Customization

Our first experiment compares our customization algorithms to the original ones. For both implementations, Table 5.2 reports the running time for the different subphases, using varying number of CPU cores. For the original variant, we give the numbers reported in [DSW16], which were obtained on the machine M2600. To ensure comparability, we run our algorithms also on M2600. All running times are averages over 1000 executions. Note that while the benchmark machine and instance are exactly the same, the nested dissection orders differ. We use Inertial Flow [SS15] to compute the contraction order, whereas the original publication uses KaHIP [SS13]. However, both partitioners generate comparable orders of the same quality [HS18].

[16] `https://github.com/RoutingKit/RoutingKit`

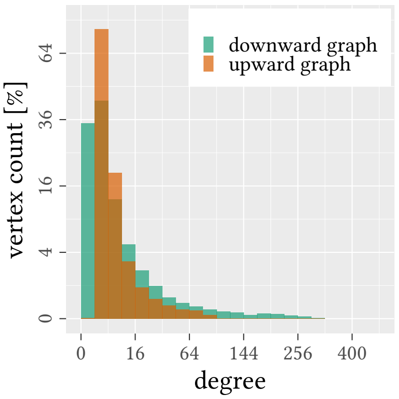
Table 5.2: Customization times on the machine M2600, using varying number of CPU cores. We report the time to perform the basic customization, run the perfect customization algorithm, and construct the minimum weighted CH for the given contraction order. For comparison, we also give the numbers reported in the original CCH publication [DSW16] (obtained on the same machine). Note that [DSW16] does not report the time to construct the minimum weighted CH, and only a combined parallel time for basic and perfect customization.

		Stuttgart [ms]				Europe [s]			
impl	cores	basic	perf	cons	total	basic	perf	cons	total
[DSW16]	1					10.88	22.08	≈ 9.39	≈ 42.35
	16					5.47		≈ 9.39	≈ 14.86
[ours]	1	20.51	20.77	48.64	89.93	5.60	6.48	9.39	21.47
	2	23.47	10.68	23.79	57.93	6.13	3.31	4.62	14.07
	4	11.39	5.79	11.41	28.59	3.42	1.71	2.50	7.63
	8	6.86	3.55	8.01	18.42	1.80	0.92	1.40	4.13
	12	5.25	3.05	5.41	13.71	1.31	0.68	1.00	2.99
	16	4.91	4.41	4.35	13.66	1.11	0.63	0.80	2.54

The table confirms that enumerating upper triangles is much faster than enumerating lower triangles. Our basic customization algorithm based on upper triangles is about twice as fast as the original variant based on lower ones, decreasing the running time from 10.9 to 5.6 seconds. Switching from lower to upper triangles reduces the number of iterations performed by the coordinated sweeps from 3.9 to 2.0 billion. Our improved initialization decreases the number of iterations further to 1.3 billion. The speedup for our perfect customization over the original variant is even higher. The running time is reduced from 22.1 to 6.5 seconds.

As discussed in Section 5.4.1, the decrease in the number of iterations performed by the coordinated sweeps is due to the distribution of the vertex degrees in $G^{\uparrow}$ and $G^{\downarrow}$, which is shown in Figure 5.4. For Europe, we observe that the fraction of the vertices with a degree between 1 and 8 is 93 % in $G^{\uparrow}$, but only 53 % in $G^{\downarrow}$. Simultaneously, the maximum degree in $G^{\uparrow}$ is 572, whereas it is 2075 in $G^{\downarrow}$. The degrees in $G^{\downarrow}$ thus are more widely dispersed, explaining the good performance of our engineered customization algorithms based on enumerating upper triangles.

As expected, our approach for parallelization based on the separator decomposition of the customizable contraction hierarchy works better for perfect customization, which requires neither synchronization steps nor atomic operations. When using 16 cores, perfect customization is faster by a factor of 10.3, while basic customization



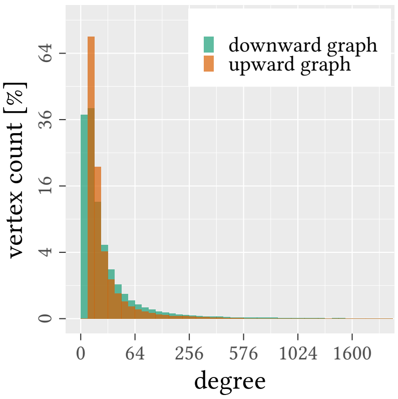


Figure 5.4: Distribution of the vertex degrees in the customizable contraction hierarchy representing Stuttgart (left) and Europe (right).

achieves a factor of only 5.0. This makes the approach a promising candidate for parallelizing the one-to-all PHAST algorithm [DGNW13].

The original CCH publication [DSW16] reports no running times for the third customization subphase, the construction of the minimum weighted CH. As our experiments show, this subphase is by far the most expensive one (taking 44 % of the sequential customization time), which justifies our engineering effort. We conjectured that the third subphase would be limited by the memory bandwidth, since there is more I/O than computation. To our own surprise, we achieve a substantial speedup of 3.8 (11.7) when using 4 (16) cores instead of one.

Due to the lack of data, we need to estimate the total customization time of the original CCH implementation. Adding the sequential running time of our third subphase to the basic and perfect customization times reported in [DSW16] yields a sequential and parallel customization time of 42.27 and 14.78 seconds, respectively. This represents a limited speedup of 2.9 when using 16 cores. In contrast, using tasking to parallelize triangle enumeration and prefix sums to parallelize CH construction, we achieve a speedup of 8.4 when using 16 cores instead of 1. Our single-threaded version is faster by a factor of 1.9, and our multithreaded version by a factor of 5.9.

As expected, we obtain smaller speedups on the much smaller Stuttgart instance, since if the amount of work is small, the parallel overhead offsets performance gains.

Compared to CRP, our sequential customization time for Europe is twice as fast, due to our optimizations. CRP customization takes 11.10 (1.09) seconds [DGPW17] on 1 core (12 cores), and enables query times of 1670 microseconds [DGPW17].

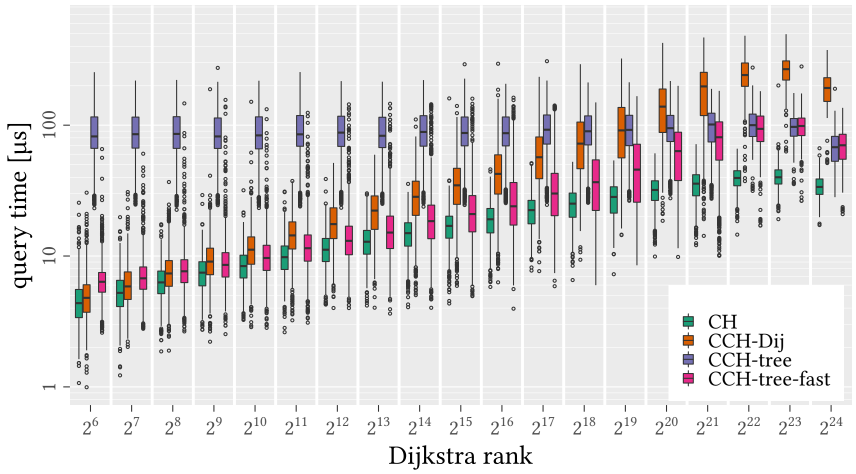


Figure 5.5: Performance of our engineered elimination tree search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and a CH. The input is the European road network with travel times.

However, our engineered basic CCH customization takes 5.60 (1.31) seconds on 1 core (12 cores), and enables query times of 218.85 microseconds. To reduce query times even further (to 88.11 microseconds), we can run the entire CCH customization, which takes 21.47 (2.99) seconds on one core (twelve cores). Note that both the CRP and CCH customization times can be further decreased by two related techniques known as *microcode* [DW13] (for CRP) and *triangle preprocessing* [DSW16] (for CCHs). However, both techniques require significantly more space, and we choose not to use them to keep the space requirement low.

5.7.3 Elimination Tree Search

Next, we evaluate our engineered elimination tree search. As most queries in the real world tend to be local, we use the established Dijkstra rank methodology [SS05], which considers local and long-range queries equally. In contrast, random queries (with the source vertex and the target vertex picked uniformly at random) tend to be long-range. The Dijkstra rank (with respect to a source vertex s) of a vertex v is r if v is the r-th vertex settled by a Dijkstra search from s. We run 1000 point-to-point queries (without path unpacking) for each Dijkstra rank tested, with s picked uniformly at random. Figure 5.5 compares the performance of our accelerated elimination tree

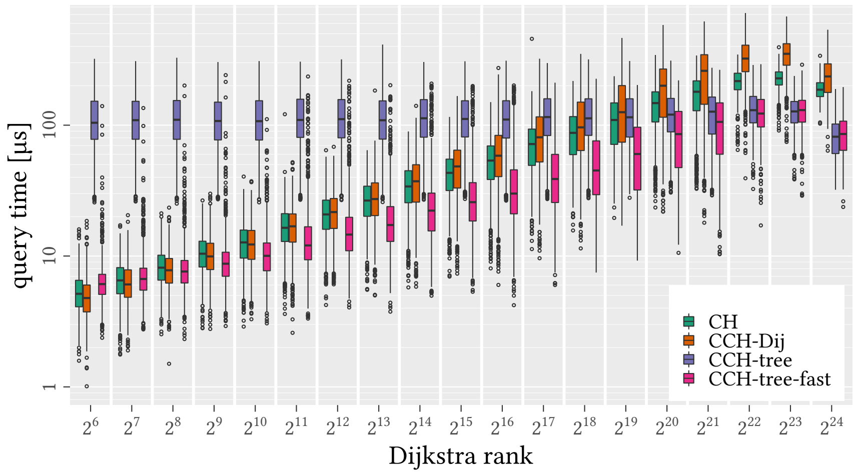


Figure 5.6: Performance of our engineered elimination tree search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and a CH. The input is the European road network with travel distances.

search (CCH-tree-fast), the original CCH query algorithms (CCH-Dij and CCH-tree), and the plain CH search on Europe with travel times. Note that CCH-tree is not really the original search, but already uses our phase reduction optimization. CCH-tree-fast additionally uses our stricter pruning rule. Moreover, CH applies stall-on-demand, whereas CCH-Dij does not, since it would slow down queries.

We observe that CCH-tree, while outperforming CCH-Dij on uniform random queries [DSW16], is actually much slower for most Dijkstra ranks, especially for the realistic ones. The reason is that the performance of CCH-tree is independent of the Dijkstra rank, since it always processes each vertex in the search space. However, our stricter pruning rule makes the algorithm sensitive to the Dijkstra rank, drastically speeding up short- and mid-range queries (by up to a factor of 13). This speedup is due to a reduction in the average number of edge relaxations, which decreases from 90 871 to 381 for rank 2^6, and from 94 721 to 12 387 for rank 2^{15}. As a result, CCH-tree-fast combines the good local-query performance of CCH-Dij with the good global-query performance of CCH-tree, and is faster than both on mid-range queries. It can be seen as a unified CCH query algorithm, replacing both original ones. Moreover, for many (realistic) Dijkstra ranks, it is about as fast as the non-customizable CH search. When optimizing travel *distances* (see Figure 5.6), CCH-tree-fast even outperforms the CH search, which was observed before [GSSV12, DGNW13, DGPW17].

Note that the decrease in query time for Dijkstra rank 2^{24} is due to boundary effects. Since 2^{24} is close to the number of vertices in the graph, the targets for rank 2^{24} are close to the boundary of the graph, where search spaces tend to be smaller. For example, the average size of the CCH search space of the targets tested is 532 for rank 2^{24}, whereas it is between 750 and 871 for all other ranks.

5.7.4 Traffic Assignment

We now evaluate the impact of several optimizations on the performance of the traffic assignment procedure. As already mentioned, we stop the iterative procedure when the relative gap drops below the threshold of 10^{-4} (as recommended by Boyce et al. [BRB04]), resulting in 29 iterations for S-morn, 34 iterations for S-even, 39 iterations for S-day, 29 iterations for S-week, and 20 iterations for L-peak.

Customization and Centralized Searches. Table 5.3 considers the influence of customization and of the centralized searches on the performance of the traffic assignment procedure. For now, we use only a single core. The CCH-based procedures use the engineered elimination tree search.

Switching from weighted to customizable CHs reduces the running time for all traffic scenarios. As expected, we obtain larger speedups for smaller scenarios (a factor of 2.2 on S-morn), since preprocessing time dominates more in such scenarios. In contrast, reordering the OD pairs so that similar OD pairs are processed successively works better for larger scenarios, improving the running time on S-week by more than 50 %. Again, this is expected, as larger scenarios have more OD pairs between each origin and destination cell. Moreover, we observe that the CH-based procedure benefits less from better locality (its running time for S-week improves by only 23 %), since the Dijkstra-based CH search performs more computational work than the elimination tree search. (Although the clustering approach described in Section 5.6 is tailored to the elimination tree search, experiments with unbiased clustering approaches not building upon the elimination tree showed a quite similar difference.)

The impact of computing multiple shortest paths at once without exploiting instruction-level parallelism is limited. However, when using SIMD instructions, the centralized searches decrease the running time by up to another factor of 3.2. Increasing k allows us to compute more shortest paths at once, but it also evicts useful data from caches. Setting $k = 32$ seems to be a good choice. Moreover, we observe that the centralized elimination searches achieve greater speedups than the Dijkstra-based CH searches, since they are label-setting.

Combining the optimizations, the traffic assignment procedure based on AVX-accelerated centralized elimination tree searches with $k = 32$ gives the best overall performance. It speeds up the state of the art by a factor of about 8 on all of our

Table 5.3: Impact of the centralized searches on the running time (in seconds) of the traffic assignment procedure for our scenarios. We evaluate the influence of using customizable CHs, reordering the OD pairs (sort), computing *k* shortest paths simultaneously, and using SSE and AVX instructions. The prior state of the art and our default configuration are highlighted in gray.

algo	sort	*k*	SIMD	S-morn	S-even	S-day	S-week	L-peak
Dij	○	1	–	5 753.22	8 239.57	106 687.46	508 186.68	1 648.98
Bi-Dij	○	1	–	2 459.27	3 265.95	44 078.13	202 515.48	907.85
CH	○	1	–	90.89	120.83	1 048.10	4 613.35	86.58
CH	●	1	–	84.19	111.22	876.70	3 573.20	81.98
CH	●	4	–	84.88	107.30	742.18	2 738.59	82.70
CH	●	4	SSE	71.72	90.49	537.50	1 944.78	63.80
CH	●	8	–	94.27	118.20	781.19	2 823.55	92.11
CH	●	8	SSE	71.12	88.06	469.65	1 662.97	60.02
CH	●	8	AVX	68.95	85.18	439.50	1 557.99	56.83
CH	●	16	AVX	70.31	87.04	424.94	1 440.46	55.48
CH	●	32	AVX	73.87	92.16	412.80	1 292.16	54.36
CH	●	64	AVX	91.09	113.20	502.56	1 441.68	62.48
CCH	○	1	–	41.50	55.02	698.16	3 203.09	49.01
CCH	●	1	–	26.98	35.45	372.34	1 552.25	32.23
CCH	●	4	–	31.73	42.10	452.73	1 879.35	40.03
CCH	●	4	SSE	18.29	23.95	230.18	930.83	20.47
CCH	●	8	–	34.39	45.32	472.77	1 954.82	42.69
CCH	●	8	SSE	17.45	22.74	211.26	856.81	18.65
CCH	●	8	AVX	15.30	19.94	175.72	690.34	15.89
CCH	●	16	AVX	14.46	18.68	153.06	585.87	13.52
CCH	●	32	AVX	14.12	18.20	132.54	490.67	11.44
CCH	●	64	AVX	18.83	24.27	160.51	553.09	13.07

benchmark scenarios. Compared to the Dijkstra-based baseline, this configuration is between two and three orders of magnitude faster.

Core-Level Parallelism. Table 5.4 shows how the traffic assignment procedure scales as the number of cores increases. We observe that the time spent on queries scales very well. With 4 (16) cores, we gain a speedup of 3.8 (14.3) for S-week, and even our smallest scenario is accelerated by a factor of 3.1 (5.9). In total, our multi-threaded AVX-accelerated centralized traffic assignment procedure decreases the running time on our main benchmark instance S-morn from 90.9 to 2.4 seconds.

Table 5.4: Impact of core-level parallelization on the performance of the traffic assignment procedure. We report for each traffic scenario the time spent on customization and queries, as well as the total running time of the traffic assignment (all in seconds).

		S-morn			S-day			S-week		
algo	cores	cust	qy	total	cust	qy	total	cust	qy	total
CH	1	36.1	54.1	90.9	49.5	997.6	1 048.1	36.2	4 576.4	4 613.3
	16	36.5	4.0	40.5	50.2	67.7	118.0	36.2	306.0	342.3
CCH	1	1.8	11.8	14.1	2.4	129.3	132.5	1.8	488.3	490.7
	2	1.1	6.6	8.0	1.5	69.0	70.9	1.1	253.3	254.7
	4	0.6	3.8	4.6	0.8	36.4	37.5	0.6	130.0	130.8
	8	0.3	2.5	2.9	0.4	19.3	19.8	0.3	66.2	66.6
	12	0.3	2.1	2.4	0.4	13.4	13.9	0.3	44.9	45.2
	16	0.4	2.0	2.4	0.4	10.6	11.1	0.3	34.2	34.5

For comparison, we also run the state of the art on multiple cores, parallelizing the shortest-path computations as described in Section 5.6. We observe that even on a single core, our AVX-accelerated traffic assignment procedure is more than three times faster for S-morn than parallelized state of the art. The difference between both parallelized versions is about an order of magnitude.

Convergence. Next, we evaluate how long our traffic assignment algorithm takes to achieve convergence. Figure 5.7 shows the relative gap (our convergence criterion) after each iteration for each scenario. The convergence rate is quite similar for all scenarios. We observe that our traffic assignment algorithm enjoys fast convergence in early iterations, but exhibits slower convergence in later iterations, when the current solution is in the vicinity of the optimal solution. This is not surprising [PERW15], since there is a CFW algorithm at the heart of our traffic assignment.

Time per Iteration. Figure 5.8 plots the running time (per phase) that our multi-threaded traffic assignment spends in each iteration. First, we observe that the procedure spends the same amount of time in each iteration. Although the inherent hierarchy of the network is weakened while computing an equilibrium flow pattern [LS11], this is expected, since the performance of both CCH customization and queries is mostly metric-independent [DSW16]. For our smallest scenario, customization takes 15.4 % of the total time. This decreases to 3.8 % for S-day, and to 0.9 % for our largest scenario. All other work, such as the line search, the edge updates, and the convergence checks, is negligible. For S-morn, it takes only 2.6 % of the total time.

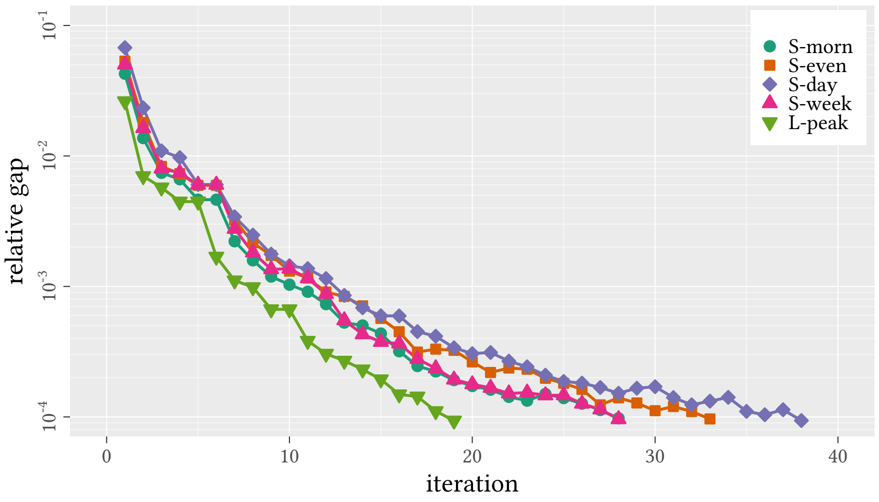
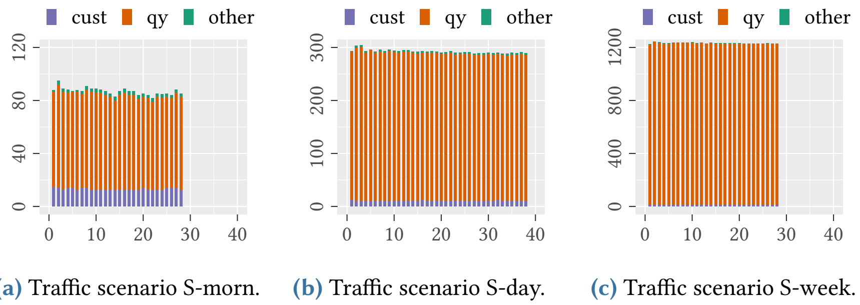


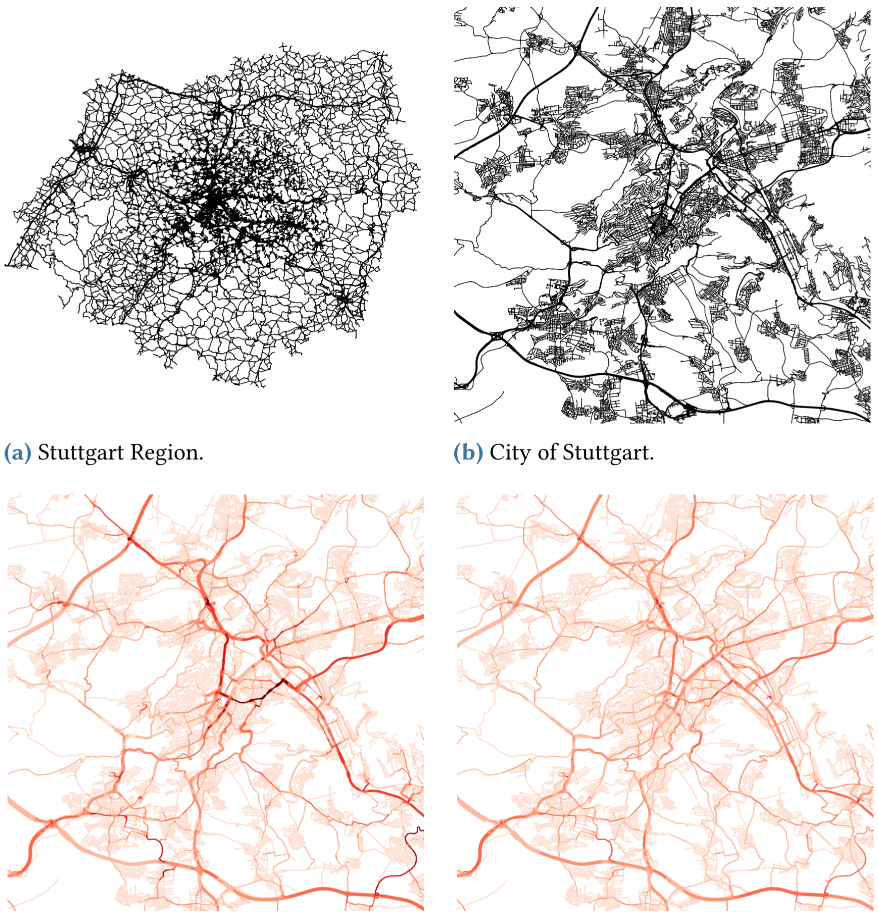
Figure 5.7: Convergence of the traffic assignment procedure. The plot shows the relative gap after each iteration. We stop as soon as the gap drops below 10^{-4}.



(a) Traffic scenario S-morn. (b) Traffic scenario S-day. (c) Traffic scenario S-week.

Figure 5.8: Time in milliseconds (vertical) spent in each iteration (horizontal) for the multi-threaded traffic assignment procedure (using all 16 cores). For S-week, customization and other work are hardly visible, since they take only 0.95 % and 0.17 % of the total running time, respectively.

Visualization. Our benchmark instance representing the Stuttgart Region and the S-morn flow patterns after one and 28 iterations are shown in Figure 5.9. We see that the traffic is distributed among more road segments after 28 iterations, since some motorists use alternative paths to improve their travel time.


(a) Stuttgart Region. (b) City of Stuttgart.

(c) Flow pattern after one iteration. (d) Flow pattern after 28 iterations.

Figure 5.9: Convergence of the equilibrium flow pattern associated with the traffic scenario S-morn. The darker the red, the more congested the road segments are.

5.8 Conclusion

We accelerated the computation of equilibrium flow patterns significantly. This was achieved by revisiting and carefully engineering several algorithms working on customizable contraction hierarchies. We proposed an improved and fully parallelized CCH customization phase, a unified CCH query algorithm (replacing both original query algorithms), and a centralized elimination tree search for batched point-to-point queries. All optimizations were extensively evaluated on real-world data used in production systems. On a metropolitan area encompassing about 2.7 million inhabitants, we compute the flow pattern for a typical one-hour morning peak (a quarter million trips) in merely 2.4 seconds, 37 times faster than the state of the art, and more than three orders of magnitude faster than the Dijkstra-based baseline.

For traffic scenarios where the shortest-path computations are still the performance bottleneck of the traffic assignment procedure, it would be interesting to process only a sample of the demand in early iterations and add more and more OD pairs in subsequent iterations. Since the result of early iterations is only a loose approximation of the equilibrium flow pattern (because the edge costs still change), it probably suffices to examine only an approximation (sample) of the demand. We hope that this speeds up early iterations significantly, without negatively affecting the convergence of the procedure. In a preliminary experiment, we were not able to achieve a significant speedup on our benchmark instance S-morn. However, we are interested to test sampling on benchmark instances that are even an order of magnitude larger than the one used in this chapter. Moreover, it would be interesting to study the efficient computation of time-dependent traffic flow profiles, which relate the traffic flow on an edge to the time of day.

6 Dynamic Ridesharing

This chapter studies the problem of servicing a set of ride requests by dispatching a set of shared vehicles, faced by ridesharing companies such as Uber and Lyft. Solving this problem at a large scale might be crucial in the future for effectively using large fleets of autonomous vehicles. Since finding a solution for the entire set of requests that minimizes the total driving time is NP-complete, most practical approaches process the requests one by one. Each request is inserted into any vehicle's route such that the increase in driving time is minimized. Although this variant is solvable in polynomial time, it still takes considerable time in current implementations, even when inexact filtering heuristics are used. In this chapter, we present a novel algorithm for finding best insertions, based on (customizable) contraction hierarchies with local buckets. Our algorithm finds provably exact solutions, is still 30 times faster than a state-of-the-art algorithm currently used in industry and academia, and scales much better. When used within iterative transport simulations, our algorithm decreases the simulation time for largescale scenarios with many requests from days to hours.

This chapter is based on joint work with Peter Sanders and Dorothea Wagner [BSW21].

6.1 Introduction

Taxi-like transport options such as cabs, minibuses, rickshaws and ridesharing services already play a vital role in meeting the transport demand in metropolitan areas. They may become even more important in the presence of intelligent ridesharing software, autonomous vehicles, and the desire to combat traffic jams, accidents, air pollution, and lack of sufficient parking. With many thousands and eventually millions of vehicles and riders, this yields fairly complex combinatorial optimization problems that have to be solved in real time. In order to evaluate the impact of ridesharing on people, the environment and the economy, we also have to simulate large realistic scenarios *now*. This requires processing millions of ride requests again and again. For example, one of the leading transport simulators [HNA16] performs hundreds of runs in order to compute realistic activity-travel patterns that describe how travelers behave under certain assumptions.

Current approaches to solve the ridesharing problem require a huge number of calls to Dijkstra's shortest-path algorithm. These are prohibitively expensive for large-scale transport simulations and they are a limiting factor for real-time dispatching of large fleets in metropolitan areas. The goal of this chapter is to show how to replace Dijkstra's classic algorithm with much faster route planning algorithms.

Ridesharing problems come in a wide variety with different assumptions, objectives, and constraints. To make our work tractable and concrete, we focus on one particular scenario adopted by a leading group in transport simulation [BMN17, HNA16]. This scenario mimics a ridesharing service that answers real-time requests for immediate rides from a given source to a given target. The dispatching algorithm knows the current routes of a fleet of vehicles, each of which has a certain number of seats. The algorithm tries all possible ways to insert a ride request into each vehicle's route. The objective is to minimize the total operation time of the fleet. There are also constraints on the maximum wait time and the maximum time when a rider should reach their target. The best insertion that satisfies all constraints is selected. We use a network with scalar (time-independent) travel times. However, by building on customizable contraction hierarchies [DSW16], we can quickly update these costs according to the current traffic situation every few minutes.

Our novel dispatching algorithm LOUD (for local buckets dispatching) adapts bucket-based contraction hierarchies [Kno+07] developed for many-to-many shortest paths to the ridesharing problem. We now briefly outline the main ideas of LOUD.

Contraction hierarchies (CHs) [GSSV12] are a point-to-point route planning technique that is much faster than Dijkstra's algorithm (four orders of magnitude on continental networks). CHs replace systematic exploration of *all* vertices in the network with two much smaller search spaces (forward and reverse) in directed acyclic graphs, in which each edge leads to a "more important" vertex. Customizable contraction hierarchies (CCHs) [DSW16] are a variant of CHs that can handle updates to the edge costs quickly (e.g., to support real-time traffic updates).

CHs with buckets (BCHs) [Kno+07] extend standard and customizable CHs to the many-to-many shortest-path problem by storing CH search spaces in buckets. More precisely, if v appears in a search space from s with distance x, then (s, x) is stored in a *bucket* $B(v)$ associated with v. For example, assume that we have stored the forward search spaces of a set S of vertices in buckets. Now, we can perform a many-to-one query (from S to a vertex t) by computing the reverse CH search space from t. For each vertex v in the search space with distance y to t, we scan the bucket $B(v)$. For each entry $(s, x) \in B(v)$, we obtain $x + y$ as a candidate for the shortest-path distance from source s to target t.

Geisberger et al. [Gei+10] adapt BCH to a simple carpooling problem, where drivers with a fixed source and target can pick up and drop off passengers heading the same way, as a means of sharing the costs of travel. Their problem, however, is very simplistic. The authors neglect departure times, vehicles shared with more than one passenger, and vehicles already on their way.

Our Contribution. We present LOUD, a novel algorithm for the ridesharing problem outlined above. LOUD maintains the forward and reverse CH search spaces of all scheduled (but not completed) pickups and dropoffs in buckets. From these buckets, LOUD can quickly obtain the cost of each possible insertion (i.e., the increase in operation time that is caused by the insertion).

One of our main contributions is a technique to aggressively prune the buckets, so that only those entries remain that can possibly contribute to feasible insertions. This technique decreases the search-space size by a factor of more than 20. Another major contribution is a filtering technique that restricts the search for the best insertion to a small set of promising vehicles. We stress that both techniques do not sacrifice optimality. A contribution that is also applicable to other dispatching algorithms is a data structure for checking whether an insertion into a vehicle's route satisfies the constraints of each rider assigned to the same vehicle. We can do this in constant time, independent of the number of riders assigned to the vehicle.

We extensively evaluate our LOUD implementation on the state-of-the-art Open Berlin Scenario [ZKN19] and a second, even larger benchmark instance. The experimental results show that LOUD is 30 times faster than algorithms currently used in industry and academia. When used in a transport simulator that performs hundreds of runs, the simulation time decreases from days to hours.

Related Work. Dynamic ridesharing is related to the classic *dial-a-ride problem* (DARP) in operations research; see [CL07, Ho+18] for recent overviews. The DARP literature, however, primarily considers the *static* variant (where all ride requests are known in advance), often defines the problem on a complete graph, and frequently solves only small problem instances (using integer linear programming methods in many cases). For these reasons, most DARP approaches are unsuitable for modern largescale ridesharing services in practice.

Finding a solution for an entire set of ride requests that minimizes the total driving time is NP-complete by reduction from the traveling salesman problem with time windows [MZW13, Sav85]. Jung et al. [JJP16] propose a simulated-annealing algorithm for this problem. More scalable approaches insert the requests one by one into any vehicle's route while leaving all other vehicle routes unchanged (often using inexact filtering heuristics to make them practical).

The vehicle dispatching algorithm [BMN17] used by the transport simulation *MATSim* [HNA16] works in three phases. Given a ride request, the first phase tries *all* possible insertions into *each* vehicle's route. For efficiency, all needed detour times are estimated using geometric distances. The second phase uses Dijkstra's algorithm [Dij59] to compute exact detour times for each insertion that is feasible based on the detour estimates. The last phase evaluates all filtered insertions again (now using exact detour times) and picks the best insertion among those.

The *T-Share algorithm* [MZW13] partitions the network into cells using a grid and precomputes the shortest-path distance between all cell centers. To quickly find a heuristic set of candidate vehicles, T-Share searches cells close to the request's source and target cell. For each candidate vehicle, T-Share tries all possible insertions. Each insertion is first evaluated using detour estimates based on precomputed distances, and if it looks feasible, T-Share computes exact (shortest-path) detour times.

Huang et al. [HBJW14] also use grid partitions to find a heuristic set of candidate vehicles. They allow to reorder requests that are already assigned to a vehicle. Shortest-path distances are computed using a very fast point-to-point routing algorithm (hub labeling [ADGW11]) and caching.

A special case of dynamic ridesharing is *dynamic carpooling*, a problem faced by carpooling services such as BlaBlaCar. In this case, the vehicle routes are not determined solely by the passengers. Instead, each driver has a fixed source and target and can pick up and drop off passengers heading the same way, as a means of sharing the costs of travel. Moreover, all constraints (such as an upper bound on the detour time) apply not only to passengers but also to drivers.

Pelzer et al. [Pel+15] partition the network along main roads into cells. For each vehicle, they maintain the sequence of cells through which the vehicle will pass (its *corridor*). A vehicle is a candidate for servicing a given ride request if the pickup is in the same cell as the vehicle and the dropoff is in the corridor of the vehicle. For each candidate vehicle, the authors compute exact detour times using Dijkstra's algorithm.

The carpooling algorithm by Geisberger et al. [Gei+10] is based on the route planning technique contraction hierarchies (CHs) [GSSV12]. It stores the forward and reverse CH search space of each vehicle's source and target, respectively, in buckets [Kno+07]. Given a ride request, the buckets are used to compute exact detour times for *all* vehicles. The studied problem, however, is very simplistic. The authors neglect departure times and can match neither more than one request with the same vehicle nor vehicles that are already on their way. Abraham et al. [Abr+12] solve the same simplistic problem in a database, with CH search spaces stored in tables.

Herbawi and Weber [HW12] combine an insertion-based algorithm with periodic reoptimizations using a relatively slow evolutionary algorithm.

There has also been previous work on *multi-hop* carpooling [DL13, MJ17], where passengers can transfer from one vehicle to another as part of a single journey. These algorithms model the problem as a time-expanded graph [PS98], similar to graph-based techniques for journey planning in public transit networks [SWW00, Bas+10, DDPW15]. To avoid combinatorial explosion, however, they need to discretize both space and time. That is, they do not support door-to-door transport and departures, arrivals and transfers can only happen at interval endpoints. Despite these limitations, the matching algorithms are relatively slow, even on medium-sized instances.

Outline. This chapter is organized as follows. Section 6.2 provides a precise definition of the basic problem we solve. Section 6.3 describes LOUD in detail, including extensions to meet additional requirements of real-world production systems. Section 6.4 presents an extensive experimental evaluation on various benchmark instances, which includes a comparison to related work. Section 6.5 concludes with final remarks. Crucial building blocks LOUD builds on were described in Chapter 2.

6.2 Problem Statement

This section defines the basic problem we consider in this chapter. Potential extensions of the basic problem will be discussed in Section 6.3.5.

We treat a road network as a directed graph $G = (V, E)$ where vertices represent intersections and edges represent road segments. Each edge $(v, w) \in E$ has a nonnegative length $\ell(v, w)$ representing the travel time between v and w. Note that we denote by $dist(v, w)$ the shortest-path distance (i.e., travel time) from v to w.

We are given a set of vehicles. Each vehicle $\nu = (l_\mathrm{i}, c, t^{\mathrm{min}}_{\mathrm{serv}}, t^{\mathrm{max}}_{\mathrm{serv}})$ has an initial location l_i, a seating capacity c, and a service interval $[t^{\mathrm{min}}_{\mathrm{serv}}, t^{\mathrm{max}}_{\mathrm{serv}})$. For each vehicle ν, we maintain its route $R(\nu) = \langle s_0, \dots, s_k \rangle$, which is a sequence of stops s at locations $l(s) \in V$ that are already scheduled for the vehicle. At each stop, the vehicle picks up and/or drops off one or more riders. Independent of the number of riders boarding and alighting, each stop takes time t_{stop}. Each vehicle's route is continuously updated according to the current situation. More precisely, if a vehicle ν is currently making a stop, then s_0 is the current stop. If a vehicle ν is currently driving, then s_0 is the previous stop (i.e., the vehicle's current location $l_\mathrm{c}(\nu)$ is somewhere between s_0 and s_1). Idle vehicles prolong their last stop. Abusing notation, we sometimes use stops as vertices. For example, $dist(s, s')$ is a shorthand for $dist(l(s), l(s'))$.

We consider a scenario in which a dispatching server receives ride requests and immediately matches them to vehicles. Each request $r = (p, d, t_{\text{dep}}^{\text{min}})$ has a pickup spot $p \in V$, a dropoff spot $d \in V$, and an earliest departure time $t_{\text{dep}}^{\text{min}}$. We do not allow pre-booking, i.e., each ride request is submitted, received and matched at $t_{\text{dep}}^{\text{min}}$. This is by far the most common scenario, adopted by the leading ridehailing services Uber and Lyft and also by related work [BMN17, MZW13, HBJW14, JJP16]. The goal is to insert each request into any vehicle's route such that the vehicle's detour δ (i.e., the increase in operation time) is minimized. Formally, an insertion can be described by a quadruple (ν, r, i, j) indicating that vehicle ν picks up request r immediately after stop $s_i(\nu)$ and drops off r immediately after stop $s_j(\nu)$. Besides capacity and service time constraints, the insertion is subject to two additional constraints.

(1) The *wait time* for each request r' already matched to the vehicle must not exceed a certain threshold, i.e., after the insertion the vehicle must still pick up request r' no later than $t_{\text{dep}}^{\text{max}}(r') = t_{\text{dep}}^{\text{min}}(r') + t_{\text{wait}}^{\text{max}}$, where $t_{\text{wait}}^{\text{max}}$ is a parameter.

(2) The *trip time* for each request r' already matched to the vehicle must not exceed a certain threshold, i.e., after the insertion the vehicle must still drop off r' no later than $t_{\text{arr}}^{\text{max}}(r') = t_{\text{dep}}^{\text{min}}(r') + t_{\text{trip}}^{\text{max}}(r') = t_{\text{dep}}^{\text{min}}(r') + \alpha \cdot \mathit{dist}(p(r'), d(r')) + \beta$, where α and β are model parameters as well.

For each request already matched to the vehicle, (1) and (2) are *hard* constraints, i.e., they must always be satisfied. If any wait or trip time constraint is violated, the insertion is feasible only if it leads to no additional delay for any already matched request. For the request r to be inserted, (1) and (2) are *soft* constraints, i.e., they may be violated. However, the violation of the wait time constraint and the violation of the trip time constraint are added to the objective value. More precisely, the objective value $f(\iota)$ of an insertion ι is formally given by

$$\begin{aligned} f(\iota) = \delta + \gamma_{\text{wait}} \cdot \max\{t_{\text{dep}}(p(r)) - t_{\text{dep}}^{\text{max}}(r), 0\} \\ + \gamma_{\text{trip}} \cdot \max\{t_{\text{arr}}(d(r)) - t_{\text{arr}}^{\text{max}}(r), 0\}, \end{aligned} \tag{6.1}$$

where $t_{\text{dep}}(p(r))$ is the scheduled departure time at the pickup spot, $t_{\text{arr}}(d(r))$ is the scheduled arrival time at the dropoff spot, and γ_{wait} and γ_{trip} are parameters.

Whenever a request is received, the goal is to find the insertion ι into any route that minimizes $f(\iota)$. If there is no feasible insertion, the request is rejected. However, since the wait and trip time constraint are soft for the request to be inserted, a request is rejected only if all vehicles go out of service before the request can be served. With unbounded service intervals (which are particularly feasible for driverless vehicles), no requests are rejected, and each rider is serviced.

6.3 Our Approach

We begin with a high-level description of LOUD, our new algorithm for dispatching a fleet of shared (potentially autonomous) vehicles. Let $r = (p, d, t_{\mathrm{dep}}^{\mathrm{min}})$ be the ride request to be inserted and let ν be a vehicle with route $R(\nu) = \langle s_0, \dots, s_k \rangle$. We will ignore some special cases for now but will discuss them later. In particular, we defer insertions (ν, r, i, j) with $i = 0$ or $j = k$ to Section 6.3.3.

To find the best insertion for request r, we consider a superset C of the vehicles ν that allow at least one feasible insertion (ν, r, i, j) with $i \neq k$. For each vehicle $\nu \in C$, we look at all insertions (ν, r, i, j) with $0 < i \leq j < k$. For each such insertion, we check whether the hard constraints are satisfied and compute the insertion cost according to Equation (6.1), i.e., the vehicle's detour plus the violations of the soft constraints. When the algorithm stops, we return the best feasible insertion seen.

To obtain the cost of an insertion (ν, r, i, j) we generally need the distance $\mathit{dist}(s_i, p)$ from stop s_i to the pickup spot p, the distance $\mathit{dist}(p, s_{i+1})$ from p to stop s_{i+1}, the distance $\mathit{dist}(s_j, d)$ from stop s_j to the dropoff spot d, and finally the distance $\mathit{dist}(d, s_{j+1})$ from d to stop s_{j+1}. We propose using BCHs to compute these distances. For each vertex h, we maintain a *source bucket* $B_{\mathrm{s}}(h)$ and a *target bucket* $B_{\mathrm{t}}(h)$, both initially empty. Whenever we insert a stop s into a vehicle's route, we run a forward (reverse) CH search from s and insert, for each vertex h settled by the search, an entry $(s, d_s(h))$ into $B_{\mathrm{s}}(h)$ ($B_{\mathrm{t}}(h)$). When we receive the ride request r, we run two forward BCH searches (from p and from d) that scan the target buckets, and two reverse BCH searches (from p and from d) that scan the source buckets. This gives us the distances we need to compute the costs of all candidate insertions.

We are now ready to introduce one of the main ideas of LOUD. We observe that the leeway λ between each pair of consecutive stops we have to insert new stops is bounded, due to the hard constraints for the requests already matched to a vehicle. That is, we are not allowed to take arbitrarily long detours between two consecutive stops on a vehicle's route. See Figure 6.1 for an illustration. Each additional stop s we may insert between stops s_i and s_{i+1} has to lie inside a *shortest-path ellipse*, defined as the set of vertices v with $\mathit{dist}(s_i, v) + \mathit{dist}(v, s_{i+1}) \leq \lambda$ (i.e., s_i and s_{i+1} are the foci of the ellipse). Naturally, the entire shortest path from s_i via s to s_{i+1} has to lie inside the ellipse. Hence, when computing source bucket entries from s_i, we need to insert an entry $(s_i, d_{s_i}(h))$ into $B_{\mathrm{s}}(h)$ only if h lies inside the ellipse around s_i and s_{i+1}. Target bucket entries can be pruned analogously. We call this *elliptic pruning* and it is surprisingly effective, as our experiments in Section 6.4 will show.

Elliptic pruning has multiple advantages. First, it accelerates the BCH searches, since these searches now scan smaller buckets. Second, it speeds up the removal of bucket entries that refer to completed stops. Note that whenever a vehicle completes a stop, the buckets are updated accordingly. The biggest advantage, however, is that

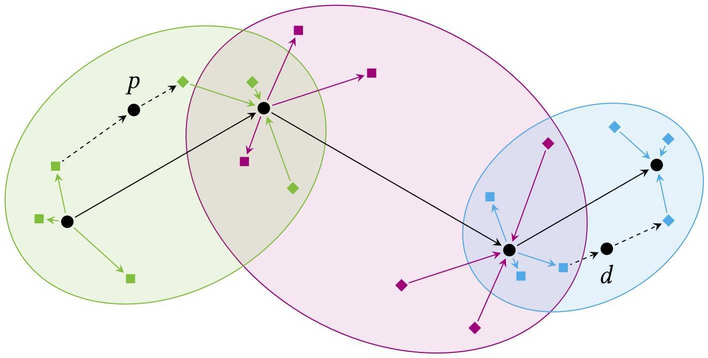


Figure 6.1: A vehicle's route consisting of four stops and the bucket entries induced by them. The stops are shown as circles and the leeway between two consecutive stops is shown as an ellipse. Source bucket entries are shown as edges with square-shaped heads and target bucket entries are shown as edges with diamond-shaped tails. Green, lilac and blue bucket entries are pruned by the respective ellipse. Consider a request $r = (p, d, t_{\text{dep}}^{\text{min}})$ where p is to be inserted immediately after the first stop s_0 and d immediately before the last stop s_3. Note that the shortest paths from s_0 to s_1 via p and from s_2 to s_3 via d lie entirely inside the respective ellipse.

elliptic pruning enables us to obtain a small superset C of the vehicles ν that allow at least one feasible insertion (ν, r, i, j) with $i \neq k$. Besides a stop identifier and a distance label, we store in each bucket entry the identifier of the vehicle to which the stop belongs. During the BCH searches, we insert all vehicle identifiers seen into C. Without elliptic pruning, the source and target bucket of the highest-ranked vertex in the hierarchy would contain an entry for each stop on each vehicle's route, and thus C would contain each vehicle, allowing no filtering at all.

The following sections work out the details of LOUD. Section 6.3.1 discusses how to check whether an insertion is feasible (i.e., satisfies the hard constraints) in constant time. Section 6.3.2 shows which bucket entries are necessary and sufficient to find the needed distances, and presents an algorithm that can efficiently check this elliptic pruning criterion. Section 6.3.3 discusses the special case of insertions (ν, r, i, j) with $i = 0$ or $j = k$. Section 6.3.4 assembles the basic LOUD algorithm from the building blocks introduced in the preceding sections. Section 6.3.5 discusses additional requirements of real-world production systems such as retrieving complete path descriptions for an insertion, incorporating real-time traffic information into the dispatching server and other potential objective functions.

6.3.1 Maintaining Feasibility

Consider a vehicle's route $\langle s_0, \dots, s_k \rangle$ and a request $r = (p, d, t^{\min}_{\text{dep}})$. We need a subroutine that checks whether the service time constraint and the wait and trip time constraints for each request assigned to the vehicle are still satisfied when inserting pickup p immediately after s_i and dropoff d immediately after s_j, $i \leq j$. Since this operation is frequently used within LOUD (and even more frequently within competitors such as MATSim), it should be as fast as possible. This section shows how to check all constraints in constant time, independent of the number of stops and the number of requests assigned to the vehicle. Note that current approaches such as MATSim and T-Share take time linear in the length of the route.

For each stop s on each route, we maintain the departure time $t^{\min}_{\text{dep}}(s)$ at stop s when no further stops are inserted into the route. Moreover, we maintain the latest arrival time $t^{\max}_{\text{arr}}(s)$ at stop s so that all following pickups and dropoffs are on time. When we add a request $r' = (p', d', t^{\min\prime}_{\text{dep}})$, yielding a route $\langle s'_0, \dots, s'_{i'} = p', \dots, s'_{j'} = d', \dots, s'_{k'} \rangle$, we loop over all s'_ℓ, $i' \leq \ell \leq k'$, in forward order and set

$$t^{\min}_{\text{dep}}(s'_\ell) = t^{\min}_{\text{dep}}(s'_{\ell-1}) + \mathit{dist}(s'_{\ell-1}, s'_\ell) + t_{\text{stop}}.$$

Furthermore, we set $t^{\max}_{\text{arr}}(s'_{i'}) = t^{\max}_{\text{dep}}(r') - t_{\text{stop}}$ as well as $t^{\max}_{\text{arr}}(s'_{j'}) = t^{\max}_{\text{arr}}(r')$. We propagate these wait and trip constraints to all preceding stops on the route by looping over all s'_ℓ, $0 < \ell \leq j'$, in reverse order and setting

$$t^{\max}_{\text{arr}}(s'_\ell) = \min\{t^{\max}_{\text{arr}}(s'_\ell), t^{\max}_{\text{arr}}(s'_{\ell+1}) - \mathit{dist}(s'_\ell, s'_{\ell+1}) - t_{\text{stop}}\}.$$

The $t^{\min}_{\text{dep}}$ and $t^{\max}_{\text{arr}}$ values allow us to check all service, wait and trip time constraints on a route in constant time. We are given a vehicle ν with route $\langle s_0, \dots, s_k \rangle$, a request $(p, d, t^{\min}_{\text{dep}})$, where p is to be inserted immediately after s_i and d immediately after s_j, and the distances $\mathit{dist}(s_i, p)$, $\mathit{dist}(p, s_{i+1})$, $\mathit{dist}(s_j, d)$, and $\mathit{dist}(d, s_{j+1})$. We first compute the pickup detour time $\delta_{\text{p}} = \mathit{dist}(s_i, p) + t_{\text{stop}} + \mathit{dist}(p, s_{i+1}) - \mathit{dist}(s_i, s_{i+1})$ and the dropoff detour time $\delta_{\text{d}} = \mathit{dist}(s_j, d) + t_{\text{stop}} + \mathit{dist}(d, s_{j+1}) - \mathit{dist}(s_j, s_{j+1})$. Note that there is no need to store $\mathit{dist}(s_i, s_{i+1})$ and $\mathit{dist}(s_j, s_{j+1})$ explicitly, as they can be obtained from the $t^{\min}_{\text{dep}}$ values. An insertion satisfies all time constraints if and only if

$$\begin{gathered} t^{\min}_{\text{dep}}(s_{i+1}) - t_{\text{stop}} + \delta_{\text{p}} \leq t^{\max}_{\text{arr}}(s_{i+1}) \text{ and} \\ t^{\min}_{\text{dep}}(s_{j+1}) - t_{\text{stop}} + \delta_{\text{p}} + \delta_{\text{d}} \leq t^{\max}_{\text{arr}}(s_{j+1}) \text{ and} \\ t^{\min}_{\text{dep}}(s_k) + \delta_{\text{p}} + \delta_{\text{d}} \leq t^{\max}_{\text{serv}}(\nu). \end{gathered}$$

An implementation needs to treat several special cases. For example, p or d can coincide with an existing stop, p or d can be inserted after s_k, or d can be inserted immediately after p. All these cases are straightforward to implement and we do not discuss them here. The correctness of our approach follows directly from Theorem 6.1.

Lemma 6.1. *All pickups and dropoffs at each stop s_j, $j \geq i$, on a vehicle's route are on time if and only if the vehicle arrives at s_i no later than $t_{\text{arr}}^{\max}(s_i)$.*

Proof. Let t be the arrival time at s_i. We claim that all pickups and dropoffs at each subsequent stop s_j are on time if $t \leq t_{\text{arr}}^{\max}(s_i)$. Assume otherwise, that is, there exists a request r with either $p(r) = s_j$ and $t_{\text{dep}}^{\max}(r) < t + t_{\text{stop}} + \sum_{k=i}^{j-1}(\mathit{dist}(s_k, s_{k+1}) + t_{\text{stop}})$ or $d(r) = s_j$ and $t_{\text{arr}}^{\max}(r) < t + \sum_{k=i}^{j-1}(\mathit{dist}(s_k, s_{k+1}) + t_{\text{stop}})$. In the former case, we have

$$t_{\text{arr}}^{\max}(s_i) \leq t_{\text{dep}}^{\max}(r) - t_{\text{stop}} - \sum_{k=i}^{j-1}(\mathit{dist}(s_k, s_{k+1}) + t_{\text{stop}}) < t,$$

where the first inequality follows from the construction of $t_{\text{arr}}^{\max}(s_i)$ and the second inequality is the assumption. This contradicts $t \leq t_{\text{arr}}^{\max}(s_i)$. In the latter case, we have

$$t_{\text{arr}}^{\max}(s_i) \leq t_{\text{arr}}^{\max}(r) - \sum_{k=i}^{j-1}(\mathit{dist}(s_k, s_{k+1}) + t_{\text{stop}}) < t,$$

where the first inequality follows from the construction of $t_{\text{arr}}^{\max}(s_i)$ and the second inequality is the assumption. Again, this contradicts that $t \leq t_{\text{arr}}^{\max}(s_i)$.

It remains to prove the "only if" part. Assume conversely that all pickups and dropoffs at each subsequent stop s_j are on time. By construction of the $t_{\text{arr}}^{\max}$ values, there is a ride request r with either $t_{\text{arr}}^{\max}(s_i) = t_{\text{dep}}^{\max}(r) - t_{\text{stop}} - \sum_{k=i}^{j-1}(\mathit{dist}(s_k, s_{k+1}) + t_{\text{stop}})$ or $t_{\text{arr}}^{\max}(s_i) = t_{\text{arr}}^{\max}(r) - \sum_{k=i}^{j-1}(\mathit{dist}(s_k, s_{k+1}) + t_{\text{stop}})$. In both cases, we have $t_{\text{arr}}^{\max}(s_i) \geq t$ by assumption, which completes the proof. □

Capacity Constraints. Besides service, wait and trip time constraints, we have to handle capacity constraints. To this end, we maintain, for each stop $s \in R$ on each vehicle route R, the occupancy $o(s)$ (the number of occupied seats) when the vehicle departs from s. Whenever we insert a request $r' = (p', d', t_{\text{dep}}^{\min\prime})$, yielding a route $\langle s'_0, \ldots, s'_{i'} = p', \ldots, s'_{j'} = d', \ldots, s'_{k'-1} \rangle$, we update the occupancies as follows. We first set $o(s'_{i'}) = o(s'_{i'-1})$ (if $s'_{i'}$ was not present before the insertion of r') and then $o(s'_{j'}) = o(s'_{j'-1})$ (if $s'_{j'}$ was not present before). Then, we loop over all s'_ℓ, $i' \leq \ell < j'$, and increment $o(s'_\ell)$. We use the o values in Section 6.3.4.

Implementation Details. We maintain one dynamic *value array* per stop attribute (such as the stop location l, the earliest departure time $t_{\text{dep}}^{\min}$, and the latest arrival time $t_{\text{arr}}^{\max}$), which stores the attribute's value for all stops on all routes. The values for stops on the same route are stored consecutively in memory, in the order in which the stops appear on the route. In addition, all value arrays share a single *index array*, which stores the starting point and ending point of each route's *value block*.

When we remove a stop from a route, we move the resulting hole in the value arrays to the end of the route's value block, and decrement the block's ending point in the index array. Consider an insertion of a stop into a route. If the element immediately after the route's value block is a hole, we insert the new stop's value into the value block and move the values after the insertion point one position to the right. Analogously, if the element before the value block is a hole, we move the values before the insertion point one position to the left. Otherwise, we move the entire value block to the end of the value arrays, and additionally insert a number of holes after the value block (the number is a constant fraction of the block size). Then, there is a hole after the block, and we proceed as described above.

6.3.2 Elliptic Pruning

We use BCHs to obtain the shortest-path distances needed to compute insertion costs, but carefully prune the source and target buckets. Let s and s' be two consecutive stops on a vehicle's route and let v be a new pickup or dropoff spot. The leeway $\lambda(s, s')$ we have to insert v between s and s' is bounded by $t_{\text{arr}}^{\max}(s') - t_{\text{dep}}^{\min}(s) - t_{\text{stop}}$. More precisely, inserting a new pickup or dropoff at v between s and s' is feasible only if $dist(s, v) + dist(v, s') \leq \lambda(s, s')$. Therefore, we only need to find shortest paths from all s to v such that $dist(s, v) + dist(v, s') \leq \lambda(s, s')$. We now show which bucket entries are necessary and sufficient for the reverse BCH search from v to find the needed distances. The case of the forward BCH search from v is symmetric.

Theorem 6.2. *Let s and s' be two consecutive stops on a vehicle's route with leeway λ between them. Consider the following two propositions:*

(1) For each vertex $h \in V$, there is an entry $(s, d_s(h))$ in the source bucket $B_s(h)$ if

(a) h is the highest-ranked vertex on all shortest s–h paths and

(b) $d_s(h) + dist(h, s') \leq \lambda$.

(2) A reverse BCH search from v finds a shortest s–v path for each vertex $v \in V$ with $dist(s, v) + dist(v, s') \leq \lambda$.

Then (1) is a necessary and sufficient condition for (2).

Proof. Assume that (1) holds and let v be a vertex with $dist(s, v) + dist(v, s') \leq \lambda$ (see Figure 6.2 for an illustration). We say that a path P is *higher* than a path Q if

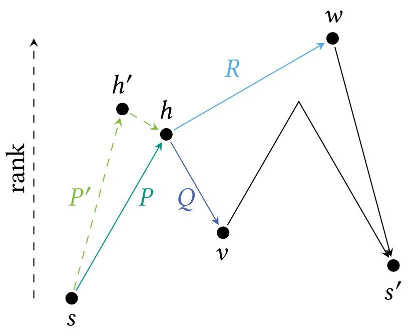


Figure 6.2: A possible pickup or dropoff at vertex v inserted between the consecutive stops s and s'. The contraction order of the vertices is given by their y-coordinates.

$\max_{w \in P} \pi^{-1}(w) > \max_{w \in Q} \pi^{-1}(w)$, where $\pi^{-1}(w)$ is the rank of w. Let h be the highest-ranked vertex on a highest of the shortest s–v paths. By construction, there is a shortest s–h path P containing only upward edges and a shortest h–v path Q containing only downward edges, and hence $P \cdot Q$ is an up-down path. We have

$$d_s(h) + dist(h, s') = dist(s, h) + dist(h, s') \le dist(s, v) + dist(v, s') \le \lambda,$$

where the equality follows from the fact that P contains only upward edges, the first inequality comes from the triangle inequality $dist(h, s') \le dist(h, v) + dist(v, s')$, and the second inequality uses the definition of v. Then $(s, d_s(h)) \in B_s(h)$ by (1), and a reverse BCH search from v finds the shortest s–v path $P \cdot Q$.

Assume conversely that (2) holds and let h be a vertex such that h is the highest-ranked vertex on all shortest s–h paths and $d_s(h) + dist(h, s') \le \lambda$. By construction, there is a shortest s–h path P containing only upward edges. We have

$$dist(s, h) + dist(h, s') = d_s(h) + dist(h, s') \le \lambda,$$

where the equality follows from the fact that P contains only upward edges and the inequality uses the definition of h. Then, by proposition (2), a reverse BCH search from h finds a shortest s–h path, i.e., there is a shortest s–h path P' that is an up-down path with highest-ranked vertex h' and $(s, d_s(h')) \in B_s(h')$. We have

$$\pi^{-1}(h) \le \pi^{-1}(h') \le \pi^{-1}(h),$$

where the first inequality uses the fact that h' is the highest-ranked vertex on P' and the second inequality follows from h being the highest-ranked vertex on all shortest s–h paths. Thus $h' = h$ and $(s, d_s(h)) \in B_s(h)$, which completes the proof. □

Bucket Entry Generation. To exploit Theorem 6.2 in practice, we need an algorithm that can efficiently check the conditions (a) and (b). Recall that with standard BCHs, we generate source bucket entries $(s, d_s(h))$ by running a forward CH search from s and inserting, for each vertex h settled, an entry $(s, d_s(h))$ into $B_s(h)$ (the case of target bucket entries is symmetric). To check condition (b), we need the distance $dist(h, s')$ for each vertex h in the search space of the forward search. We propose the following approach for obtaining these distances.

We run a *topological* forward CH search from s, i.e., we process vertices in topological order rather than in increasing order of distance. We prune the search at any vertex with a distance label greater than $\lambda(s, s')$ but do not apply stall-on-demand. The search stops when the priority queue becomes empty. Afterwards, we run a standard reverse CH search from s'. We apply stall-on-demand and stop as soon as the minimum key in its priority queue exceeds $\lambda(s, s')$. Finally, we need to propagate the labels of the reverse search down into the search space of the forward search.

We push each vertex settled during the forward search onto a stack. After the reverse search has terminated, we repeatedly pop a vertex u from the stack. For each upward edge (u, u') going out of u, we set $d_{s'}(u) = \min\{d_{s'}(u), \ell(u, u') + d_{s'}(u')\}$. We claim that when the stack becomes empty, we have $d_{s'}(h) = dist(h, s')$ for each vertex h in the search space of the forward search with $d_s(h) + dist(h, s') \leq \lambda(s, s')$, and thus can efficiently check condition (b).

Lemma 6.3. *When the algorithm terminates, we have $d_{s'}(h) = dist(h, s')$ for each vertex h in the search space of the forward search with $d_s(h) + dist(h, s') \leq \lambda(s, s')$.*

Proof. Consider one such h in particular and let w be the highest-ranked vertex on a shortest h–s' path (see Figure 6.2). The reverse CH search is guaranteed to find a shortest w–s' path and to set $d_{s'}(w)$ to its correct value (as shown by [GSSV12]). All we need to show is that the propagation phase finds a shortest h–w path.

By construction, there is a shortest h–w path R containing only upward edges. Since h is by definition in the search space of the forward search, R contains only upward edges, and the distance label of each vertex on R is by definition at most $\lambda(s, s')$, all vertices on R are pushed onto the stack. Since the forward search settles vertices in topological order, the stack contains the vertices in the same order in which they appear on R. Hence, the propagation phase relaxes the edges on R in reverse order and thus finds the h–w path R. □

It remains to check condition (a). Consider a vertex h in the search space of the forward search and let P be a shortest of the s–h paths that contain only upward edges. Condition (a) is violated if and only if there is an up-down s–h path P' with at least one downward edge and $\ell(P') \leq \ell(P)$; see Figure 6.2 for an illustration. We try to find such *witnesses* during the propagation phase.

When we pop h from the stack, we additionally look at all downward edges (h'', h) coming into h and compute $\mu = \min_{(h'',h)} d_s(h'') + \ell(h'', h)$. If $\mu \leq d_s(h)$, we found a witness, condition (a) is violated, and thus we do not insert an entry into $B_s(h)$. Either way, we set $d_s(h) = \min\{d_s(h), \mu\}$. Note that we find a witness if and only if all vertices on it are contained in the search space of the forward search. Therefore, we do not necessarily discover all violations of condition (a). However, we observed that undiscovered violations are quite rare. More importantly, undiscovered violations may yield superfluous entries but do not affect the correctness of the BCH searches.

Bucket Entry Removal. Whenever a vehicle completes a stop, we have to remove the bucket entries referring to this stop. In the following, we show how to efficiently remove the source bucket entries that refer to a stop s. The case of target bucket entries is handled in a symmetrical fashion.

We initialize both a set R of reached vertices and a queue Q with the location $l(s)$ of s. While Q is not empty, we extract a vertex v from the queue and scan its source bucket $B_s(v)$. When we find an entry $(s, d_s(v))$ referring to s, we remove $(s, d_s(v))$ from $B_s(v)$, stop the bucket scan, look at each upward edge (v, w) that leaves v, and finally insert w into both R and Q if $w \notin R$.

The algorithm finds an entry $(s, d_s(w)) \in B_s(w)$ if and only if there is an s–w path P such that P contains only upward edges and $(s, d_s(v)) \in B_s(v)$ for each vertex v on P. There would always be such a path P if we were able to guarantee to discover all violations of condition (a). Since we cannot, we explicitly ensure that there is always such a path P. Whenever we insert an entry into a source bucket $B_s(w)$, we also insert a corresponding entry into $B_s(\mathit{parent}(w))$, where $\mathit{parent}(w)$ is the parent pointer of w computed by the forward search. Our experiments will show that this almost never inserts additional bucket entries.

Implementation Details. Bucket entries must identify the stop they refer to. Therefore, we maintain an initially empty list of free stop IDs. Whenever we insert a stop into a vehicle's route, we take an ID from the list and assign it to the new stop. If the list is empty, we set the ID of the new stop to the maximum stop ID assigned so far plus one. Whenever we remove a stop from a route, we insert its ID into the list of free stop IDs. Bucket entries are stored and maintained in a way similar to how we handle stop attribute values (see Section 6.3.1).

6.3.3 Shortest-Path Searches for Special Cases

We use BCHs to obtain most of the shortest-path distances needed to compute insertion costs. However, three special cases have to be treated separately. We discuss each of them in detail in this section.

From Vehicles to Pickup. Consider an insertion (ν, r, i, j) with $R(\nu) = \langle s_0, \dots, s_k \rangle$ and $0 = i < k$. Here, the new pickup is inserted before the next scheduled stop on a vehicle's route. In this case, the vehicle is immediately diverted to the new pickup. To compute the cost of the insertion, we need the distance $dist(l_c(\nu), p(r))$ from the current location $l_c(\nu)$ of the vehicle ν to the pickup spot $p(r)$. Our BCH searches do not find shortest paths from the vehicle's current location. Since the current location changes continuously, we cannot precompute bucket entries for it. However, the BCH searches provide us with a lower bound on the actual pickup detour.

The time from s_0 to s_1 via $p(r)$ is $dist(s_0, l_c(\nu)) + dist(l_c(\nu), p(r)) + dist(p(r), s_1)$. The inequality $dist(s_0, p(r)) \leq dist(s_0, l_c(\nu)) + dist(l_c(\nu), p(r))$ then yields a lower bound of $dist(s_0, p(r)) + dist(p(r), s_1)$ on the travel time from s_0 to s_1 via $p(r)$. Since we have source bucket entries for s_0 and target bucket entries for s_1, this lower bound can be obtained from the BCH searches. We can then compute lower bounds on the pickup detour and finally on the cost of the insertion. Only in the rare case that the latter lower bound is better than the best insertion seen so far, we have to compute the exact shortest-path distance $dist(l_c(\nu), p(r))$ by running a standard CH query. As our experiments will show, we typically only need a single CH query per request.

From Last Stops to Pickup. Consider an insertion (ν, r, i, j) with $R(\nu) = \langle s_0, \dots, s_k \rangle$ and $i = k$. Here, the new pickup is inserted after the last stop on a vehicle's route. Observe that this case also covers currently idle vehicles. To compute the cost of such insertions, we need the shortest-path distance $dist(s_k, p(r))$ from the last stop s_k to the pickup spot $p(r)$. However, our BCH searches do not find shortest paths from the last stop. The reason is that we do not generate source bucket entries for the last stop, since we cannot apply elliptic pruning in this case (the leeway is unbounded).

Instead, we defer all insertions (ν, r, i, j) with $R(\nu) = \langle s_0, \dots, s_k \rangle$ and $i = k$. After having tried all candidate insertions (ν', r, i', j') with $R(\nu') = \langle s'_0, \dots, s'_{k'} \rangle$ and $j' \neq k'$, we perform a reverse Dijkstra search from $p(r)$. Whenever we settle the last stop of a vehicle ν with $R(\nu) = \langle s_0, \dots, s_k \rangle$, we check whether the insertion (ν, r, k, k) improves the currently best insertion. Note that the detour (i.e., the increase in operation time) for each such insertion is $\delta = dist(s_k, p(r)) + t_{\text{stop}} + dist(p(r), d(r)) + t_{\text{stop}}$, and thus its cost is at least δ. Therefore, we can stop the search when the sum of the minimum key κ in its priority queue and $t_{\text{stop}} + dist(p(r), d(r)) + t_{\text{stop}}$ is at least as large as the cost of the best insertion found so far. We can do even better by taking into account lower bounds on the violations of the wait and trip time constraint. More precisely, we can stop the search as soon as the sum

$$\begin{aligned}
&\kappa + t_{\text{stop}} + dist(p(r), d(r)) + t_{\text{stop}} \\
&\quad + \gamma_{\text{wait}} \cdot \max\{\kappa + t_{\text{stop}} - t^{\max}_{\text{wait}}, 0\} \\
&\quad + \gamma_{\text{trip}} \cdot \max\{\kappa + t_{\text{stop}} + dist(p(r), d(r)) - t^{\max}_{\text{trip}}(r), 0\}
\end{aligned}$$

is at least as large as the cost of the currently best insertion. Stopping the Dijkstra search early makes it practical and fast enough for real-time applications.

From Last Stops to Dropoff. Lastly, consider a candidate insertion (ν, r, i, j) with $R(\nu) = \langle s_0, \ldots, s_k \rangle$ and $i < j = k$. Here, the new pickup is inserted before and the new dropoff is inserted after the last stop on a vehicle's route. To compute the cost of that insertion, we need the shortest-path distance $dist(s_k, d(r))$ from the last stop s_k to the dropoff spot $d(r)$. As discussed before, our BCH searches do not find shortest paths from the last stop (we do not generate source bucket entries for the last stop). We treat this special case similarly to the previous one.

After running a reverse Dijkstra search from $p(r)$, we also run one from $d(r)$. Whenever we settle the last stop of a vehicle ν with $R(\nu) = \langle s_0, \ldots, s_k \rangle$, we check whether any insertion (ν, r, i, k) with $i < k$ improves the best insertion seen so far. Since the cost of each such insertion is at least $dist(s_k, d(r)) + t_{\mathrm{stop}}$, we can stop the search when the sum of the minimum key κ in its priority queue and t_{stop} is at least as large as the cost of the currently best insertion. Again, we can do better by taking into account a lower bound on the violation of the request's trip time constraint. Then, we can stop the search as soon as the sum

$$\kappa + t_{\mathrm{stop}} + \gamma_{\mathrm{trip}} \cdot \max\{t_{\mathrm{stop}} + \kappa - t_{\mathrm{trip}}^{\max}(r), 0\}$$

is as large as the cost of the best insertion found so far.

6.3.4 Putting Everything Together

In this section we assemble the basic LOUD algorithm from the building blocks introduced in the preceding sections. Given a ride request $r = (p, d, t_{\mathrm{dep}}^{\min})$, the algorithm inserts it into any vehicle's route such that the vehicle's detour plus the violations of the soft constraints (if any) is minimized. A request is resolved in four phases, and we explain each in turn in this section. In addition, Algorithm 6.1 gives high-level pseudocode for each of the phases.

Computing Shortest-Path Distances. We start by computing the shortest-path distance from the pickup p to the dropoff d with a standard CH query. From this distance, we compute the latest time $t_{\mathrm{dep}}^{\max}(r)$ when r should be picked up as well as the latest time $t_{\mathrm{arr}}^{\max}(r)$ when r should be dropped off. Next, we compute all shortest-path distances that we need to calculate the costs of all *ordinary* insertions, i.e., insertions (ν, r, i, j) with $0 < i \leq j < |R(\nu)| - 1$. We do this by running two forward BCH searches (from p and d) that scan the target buckets, and two reverse BCH searches (from p and d) that scan the source buckets.

Algorithm 6.1: Routine for resolving a received ride request $r = (p, d, t_{\text{dep}}^{\min})$.

```
                                                     Computing Shortest-Path Distances
run a CH query from pickup $p$ to dropoff $d$
$t_{\text{dep}}^{\max}(r) \leftarrow t_{\text{dep}}^{\min}(r) + t_{\text{wait}}^{\max}$
$t_{\text{arr}}^{\max}(r) \leftarrow t_{\text{dep}}^{\min}(r) + \alpha \cdot dist(p, d) + \beta$
run forward and reverse BCH searches from pickup spot $p$ and dropoff spot $d$
                                                     Trying Ordinary Insertions
let $\hat{\iota} = (\hat{\nu}, r, \hat{i}, \hat{j}) \leftarrow \perp$ be the best insertion found so far
foreach vehicle $\nu \in C$ do
    let $\langle s_0, \ldots, s_k \rangle$ be the route of vehicle $\nu$
    for $i \leftarrow 1$ to $k - 1$ do
        if $o(s_i) = c(\nu)$ then continue
        try to improve $\hat{\iota}$ with insertion $(\nu, r, i, i)$
        for $j \leftarrow i + 1$ to $k - 1$ do
            if $o(s_j) = c(\nu)$ then
                if $l(s_j) = d$ then try to improve $\hat{\iota}$ with insertion $(\nu, r, i, j)$
                break
            try to improve $\hat{\iota}$ with insertion $(\nu, r, i, j)$
                                                     Trying Special-Case Insertions
foreach vehicle $\nu \in C$ do
    try to improve $\hat{\iota}$ with any insertion $(\nu, r, 0, j)$ with $0 \le j < |R(\nu)| - 1$
search for insertions better than $\hat{\iota}$ that insert the pickup at the end of a route
search for insertions better than $\hat{\iota}$ that insert the dropoff at the end of a route
if no feasible insertion has been found then return $\perp$
                                                     Updating Preprocessed Data
let $\langle s_0, \ldots, s_k \rangle$ be the route of vehicle $\hat{\nu}$
$\langle s'_0, \ldots, s'_{i'} = p, \ldots, s'_{j'} = d, \ldots, s'_{k'} \rangle \leftarrow$ perform insertion $\hat{\iota}$
if vehicle $\hat{\nu}$ is diverted while driving from $s_0$ to $s_1$ then
    remove source bucket entries for stop $s'_0$
    $l(s'_0) \leftarrow l_c(\hat{\nu})$
    $t_{\text{dep}}^{\min}(s'_0) \leftarrow$ current point in time
    generate source bucket entries for stop $s'_0$
if the pickup is not inserted at an existing stop then
    generate source and target bucket entries for stop $s'_{i'}$
if the dropoff is not inserted at an existing stop then
    generate target bucket entries for stop $s'_{j'}$
    if the dropoff is inserted before the last stop then
        generate source bucket entries for stop $s'_{j'}$
    else
        generate source bucket entries for stop $s_k$
return $\hat{\iota}$
```

Trying Ordinary Insertions. Next, we try all ordinary insertions. To do so, we look at the set C of vehicles that have been seen while scanning the buckets (recall that we store in each bucket entry the identifier of the vehicle to which the entry belongs). Note that vehicles that are not contained in C allow no feasible ordinary insertions, and thus we do not have to consider them during this phase of the algorithm.

For each vehicle $\nu \in C$, we enumerate all possible ordinary insertions that satisfy the capacity constraints, using the occupancy values $o(\cdot)$ computed in Section 6.3.1. Let $\langle s_0, \ldots, s_k \rangle$ be the route of ν. We loop over all pickup insertion points i, $0 < i < k$, in increasing order. If the number $o(s_i)$ of occupied seats when ν departs from s_i is equal to the capacity $c(\nu)$ of ν, then all insertions $(\nu, r, i, \cdot)$ are infeasible, and we continue with the next pickup insertion point. Otherwise, we loop over all dropoff insertion points j, $i \leq j < k$, in increasing order. If $o(s_j) < c(\nu)$, then the insertion (ν, r, i, j) satisfies the capacity constraints. Otherwise, all insertions (ν, r, i, ℓ) with $\ell > j$ are infeasible, and we continue with the next pickup insertion point. The insertion with $\ell = j$ satisfies the constraints only if d coincides with s_j.

For each insertion ι satisfying the capacity constraints, we check whether the remaining hard constraints are also satisfied and compute the insertion cost according to Equation (6.1). This can be done in constant time as discussed in Section 6.3.1. Finally, if ι improves the best insertion $\hat{\iota}$ found so far, we update $\hat{\iota}$ accordingly.

Trying Special-Case Insertions. Next, we try all possible special-case insertions, i.e., insertions whose cost depends on some shortest-path distances not computed by the BCH searches. First, we try all insertions $(\nu, r, 0, j)$ with $0 \leq j < |R(\nu)| - 1$. Such insertions insert the pickup before the next scheduled stop on a vehicle's route. Since vehicles $\nu' \notin C$ allow no feasible insertions $(\nu', r, 0, j)$ with $0 \leq j < |R(\nu')| - 1$, it suffices to look at each vehicle $\nu \in C$. Let $\langle s_0, \ldots, s_k \rangle$ be the route of ν. If $o(s_0) = c(\nu)$, then ν is currently fully occupied, and thus we cannot pick up another request before the next scheduled stop. If $o(s_0) < c(\nu)$, then we loop over all dropoff insertion points j, $0 \leq j < k$, terminating the loop when $o(s_j) = c(\nu)$. For each j, we handle the insertion $(\nu, r, 0, j)$ as described in Section 6.3.3.

Second, we search for insertions better than $\hat{\iota}$ that insert both the pickup and the dropoff after the last stop on a vehicle's route. We do this by performing a reverse Dijkstra search from p, as discussed in Section 6.3.3. Finally, we search for insertions better than $\hat{\iota}$ that insert only the dropoff after the last stop on a vehicle's route. To do that, we run a reverse Dijkstra search from d, as described also in Section 6.3.3.

Updating Preprocessed Data. If we have found a feasible insertion, we need to update the preprocessed data in order to be ready to resolve the next ride request. We start by actually *performing* the best insertion $\hat{\iota} = (\hat{\nu}, r, \hat{i}, \hat{j})$ into the current route $\langle s_0, \dots, s_k \rangle$ of $\hat{\nu}$. Let $\langle s'_0, \dots, s'_{i'} = p, \dots, s'_{j'} = d, \dots, s'_{k'} \rangle$ be the route of $\hat{\nu}$ after the insertion. The $t^{\text{min}}_{\text{dep}}$, $t^{\text{max}}_{\text{arr}}$, and o values can be updated in linear time.

If $\hat{\nu}$ is diverted while driving from s_0 to s_1, we update the start s'_0 of its current leg and recompute the source bucket entries for s'_0. (Note that there are no target bucket entries for s'_0 because it is the first stop on the route.) First, we remove the current source bucket entries for s'_0. Then, we update the location of s'_0 to the current location of $\hat{\nu}$, and the departure time at s'_0 to the current point in time. Finally, we generate new source bucket entries for stop s'_0.

Moreover, we generate source and target bucket entries for the stop $s'_{i'}$ at which the pickup is made unless the pickup is inserted at an existing stop. Likewise, we generate target bucket entries for the stop $s'_{j'}$ at which the dropoff is made unless the dropoff is inserted at an existing stop. If the dropoff is inserted before the last stop, we also generate source bucket entries for $s'_{j'}$. Otherwise, we generate source bucket entries for the stop s_k that was at the very end of the route before the insertion. (Whenever a vehicle reaches the next stop on its route, we remove the target bucket entries for this stop, and the source bucket entries for the preceding stop.)

It remains to update one more data structure. For each vertex v, we maintain a list of vehicles that terminate at v, i.e., whose currently last stop is made at v. Whenever the reverse Dijkstra searches from the pickup p and the dropoff d settle a vertex v, they retrieve the last stops at v with these lists. Therefore, we remove $\hat{\nu}$ from the list of vehicles terminating at $l(s_k)$, and we insert $\hat{\nu}$ into the list of vehicles terminating at $l(s'_{k'})$. Note that this step is omitted in Algorithm 6.1.

6.3.5 Extensions

This section shows how LOUD can be extended to meet additional requirements of real-world production systems. We explain each extension in turn, but they can be combined in an actual implementation. Our implementation supports all of them.

Edge-Based Stops. Up to now, we have assumed that stops are made at vertices (i.e., intersections). In real-world applications, however, stops are made anywhere along edges (i.e., road segments). Fortunately, LOUD can be easily extended to work with edge-based stops, following the approach proposed by Delling et al. [DGPW17].

Consider a stop s along an edge $e = (v, w)$ with a real-valued offset $o \in [0, 1]$. To run a forward search (whether it is a Dijkstra, CH, or BCH search) from s, we start from the *head* vertex w and initialize the distance label $d_w(w)$ to $(1 - o) \cdot \ell(e)$ rather than zero. Likewise, to run a reverse Dijkstra, CH, or BCH search from s, we start from the *tail* vertex v and initialize the distance label $d_v(v)$ to $o \cdot \ell(e)$. The special case where source and target are located on the same edge is treated explicitly.

Path Retrieval. In real-world applications, one is often interested not only in the best insertion (ν, r, i, j) but also in the descriptions of the paths from stop s_i to the pickup spot $p(r)$, from $p(r)$ to stop s_{i+1}, from stop s_j to the dropoff spot $d(r)$, and from $d(r)$ to stop s_{j+1}. By maintaining a parent pointer for each vertex, the Dijkstra searches can retrieve complete path descriptions, and the CH searches can retrieve descriptions potentially containing shortcuts. The latter can be unpacked into complete descriptions in time linear in the length of the unpacked path [GSSV12].

Now, consider a path $\langle s, \dots, h, \dots, s' \rangle$ found by a forward BCH search. The case of a reverse BCH search is handled in a symmetrical fashion. Let h be the highest-ranked vertex on the path. Since the s–h path is found by a forward CH search, its description can be retrieved as discussed above. The h–s' path, however, is hidden behind the target bucket entry $(s', d_{s'}(h)) \in B_\mathrm{t}(h)$. Therefore, it remains to retrieve the path description that corresponds to a target bucket entry.

When we generate target bucket entries for s', we could explicitly store the search space of s' as a rooted tree $T_{s'}$. To retrieve the description of the h–s' path, we would traverse the path in $T_{s'}$ from h to s'. Note, however, that to find a best insertion, we need no parent information. That is, $T_{s'}$ is only needed when we insert a new stop immediately before s', which may never be the case. Since it seems wasteful to build a tree that may never be used, we instead retrieve the path description corresponding to a target bucket entry $(s', d_{s'}(h))$ by running a reverse CH search (from s' to h).

Handling Traffic. Today's ridesharing services have to be able to quickly update the routing graph whenever new traffic information is available. On large-scale road networks, however, CH preprocessing is not fast enough to incorporate a continuous stream of traffic information. Hence, we propose combining LOUD with *customizable* contraction hierarchies (CCHs) [DSW16], a CH variant that can incorporate new metrics in few seconds. As a customizable contraction hierarchy *is a* contraction hierarchy, LOUD can be used as is with CCHs, without further modifications.

We can do better by replacing the Dijkstra-based CH searches with elimination tree searches, a query algorithm tailored to CCHs. Elimination tree searches tend to be faster than Dijkstra-based searches for point-to-point queries, however, they have one drawback. Since they do not process vertices in increasing order of distance, it is not clear how to terminate them early. This is an issue because the Dijkstra-based CH searches during bucket entry generation have a tight stopping criterion. However, we observe that we can turn *stopping* criteria for Dijkstra-based CH searches into *pruning* criteria for elimination tree searches.

During bucket entry generation, the Dijkstra-based CH searches stop as soon as they settle a vertex whose distance label exceeds the leeway. We cannot *stop* an elimination tree search at such a vertex v. However, we can *prune* the search at v, i.e., we do not relax edges out of v. As shown in Chapter 5, the edge relaxations are the time-consuming part, whereas the time spent on tree traversal is negligible. Therefore, the pruning criteria are almost as effective as the stopping criteria.

Note that elimination tree searches even simplify bucket entry generation. In Section 6.3.2, we have introduced special *topological* CH searches, which process the vertices in the CH search space in topological order. Since elimination tree searches process vertices in ascending rank order, and the rank order is a topological order, each standard elimination tree search is already a topological search.

There is, however, a potential pitfall associated with customization. Recall that to remove bucket entries for a stop s, we essentially simulate a CH search from s to find the buckets that contain entries referring to s. This requires that the topology of the hierarchy does not change between generation and removal of the bucket entries for s. Fortunately, CCHs compute a metric-independent contraction order during a preprocessing step, i.e., customization does not affect the order. Thus, when using *basic* CCH customization [DSW16], the topology does not change, and we can safely update the edge costs between bucket entry generation and removal.

For smaller search spaces, we can apply a more sophisticated customization algorithm (*perfect* customization [DSW16]). This additionally removes superfluous edges from the customizable contraction hierarchy. Therefore, although the contraction order remains the same, the topology of the customizable contraction hierarchy may change. Hence, when using perfect customization, we have to clear and rebuild the source and target buckets after each customization step.

Other Objective Functions. Our precise objective function is taken from the popular transport simulation MATSim [HNA16, BMN17], and can be parameterized as discussed in Section 6.2. We stress, however, that LOUD is not restricted to this objective function but can work with other functions as well. Note that elliptic pruning (and therefore bucket entry generation) does not depend on the objective function, only on the hard constraints for requests already matched to a vehicle. Hence, it will perform similarly for *any* objective function. The only ingredients that depend on the actual objective function are the stopping criteria for the reverse Dijkstra searches from the received pickup and dropoff spot, respectively.

6.4 Experiments

This section presents a thorough experimental evaluation of LOUD on the state-of-the-art Open Berlin Scenario [ZKN19] and a second, even larger benchmark instance, including a comparison to related work. We also integrate LOUD into a transport simulation software, and evaluate it within this software.

6.4.1 Experimental Setup

Our source code is written in C++17 and compiled with the GNU compiler 9.3 using optimization level 3. We use 4-heaps [Joh75] as priority queues. To ensure a correct implementation, we make extensive use of assertions (disabled during measurements). Our benchmark machine runs openSUSE Leap 15.2 (kernel 5.3.18), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each with eight cores clocked at 3.50 GHz and 8×64 KiB of L1, 8×1 MiB of L2, and 24.75 MiB of shared L3 cache. Note that we consider only single-core implementations.

Inputs. Our main benchmark instances are taken from the recent Open Berlin Scenario [ZKN19], a publicly available transport simulation scenario for the Berlin metropolitan area implemented in MATSim [HNA16]. The transport simulation MATSim works in iterations, with each iteration simulating the movement of the given population (including departure time, route, mode and destination choice) and outputting each inhabitant's 24-hour travel pattern. Over the course of iterations, the activity-travel patterns become more and more realistic.

To obtain a set of realistic requests, we build on the Open Berlin Scenario 5.5 with demand-responsive transport (DRT). By default, only a few trips use DRT. Therefore, we change three parameters. We halve the DRT fare per kilometer from 35 to 18 cents, halve the minimum DRT fare per trip from 2 to 1 euro, and double the daily cost per private car from 5.30 to 10.60 euros. This primarily replaces car trips by DRT trips.

The Open Berlin Scenario has been published in two versions. The 1 % scenario simulates 1 % of all adults living in Berlin and Brandenburg, while the 10 % scenario simulates 10 % of them. For our benchmark instance *Berlin-1pct*, we take all DRT requests from the 500th iteration of the 1 % scenario (500 is the number of iterations recommended for realistic travel patterns). For our instance *Berlin-10pct*, we take all DRT requests from the 250th iteration of the 10 % scenario (since one iteration takes more than four hours, performing 500 is not feasible). Both instances take the network from the Open Berlin Scenario, which builds on OpenStreetMap.

To evaluate LOUD on even larger instances, we build two additional instances that comprise the Rhine-Ruhr area, the largest metropolitan area in Germany. The construction is guided by the Open Berlin Scenario. We start by taking the network from OpenStreetMap. Besides all roads in the city of Berlin, the Open Berlin Scenario includes all main roads in Brandenburg (the state that surrounds Berlin). For our Rhine-Ruhr instances, we therefore take all roads in the Rhine-Ruhr metropolitan area (as defined by the Landesentwicklungsplan NRW from 1995) and all main roads in the surrounding state of North Rhine-Westphalia.

As for the Open Berlin Scenario, we build a sparser instance *Ruhr-1pct* and a denser instance *Ruhr-10pct*. Since the population in the Rhine-Ruhr area is roughly three

Table 6.1: Key figures of our benchmark instances.

input	$\|V\|$	$\|E\|$	veh	req
Berlin-1pct	73 689	159 039	1 000	16 569
Berlin-10pct	73 689	159 039	10 000	149 185
Ruhr-1pct	394 049	840 587	3 000	49 707
Ruhr-10pct	394 049	840 587	30 000	447 555

times larger than in the Berlin area, we scale the numbers of vehicles and requests for Berlin-1pct (Berlin-10pct) by a factor of three to obtain the numbers for Ruhr-1pct (Ruhr-10pct). The initial vehicle locations are uniformly distributed in the Rhine-Ruhr area (no vehicle starts in the surroundings). As the population density correlates with the density of the graph, vehicles tend to start in densely populated areas.

We choose the pickup and dropoff spot for a request as follows. First, we choose the pickup spot uniformly at random from the Rhine-Ruhr area. Next, we draw the trip duration from a geometric distribution with probability parameter $p = 1/(\mu + 1)$. Finally, we run Dijkstra's algorithm from the pickup spot until we settle a vertex whose distance label is greater than or equal to the trip duration drawn before *and* that vertex is contained in the Rhine-Ruhr area. The expected trip duration μ is set to the average trip duration on the corresponding Berlin instance (12 minutes on Berlin-1pct and 11 minutes on Berlin-10pct).

The earliest departure time for a request is drawn according to the distribution of the earliest departure times on the Berlin instances. More precisely, we group the departure times on the Berlin instances into five-minute bins b_i. To choose the time for a request on the Rhine-Ruhr instances, we first draw a bin B from the discrete distribution determined by the probability function $\Pr[B = b_i] = |b_i|/\sum_{b_j} |b_j|$, and then choose the departure time uniformly at random from the interval corresponding to B. Key figures of the Berlin and Rhine-Ruhr instances are shown in Table 6.1.

Methodology. We implemented a discrete-event simulation that simulates a given set of vehicles servicing a given set of requests. The simulation maintains each vehicle's current state (out of service, idling, driving, or stopping) and an addressable priority queue of pending events. Each event happens at some scheduled point in time and may generate a new event in the future. We repeatedly extract the next event from the queue and process it. The transport simulation stops as soon as the event queue becomes empty (i.e., all events are processed).

For each ride request r in the input, we process a *request receipt event* at $t^{\min}_{\text{dep}}(r)$. To do so, we match request r to some vehicle ν. If ν is currently idling, we set its state to driving and insert a vehicle arrival event at $t_{\mathrm{now}} + \mathit{dist}(l_c(\nu), p(r))$ into the queue, where t_{now} is the current point in time. If vehicle ν is currently driving and r is inserted before the next scheduled stop, we update the scheduled time of ν's existing vehicle arrival event to $t_{\mathrm{now}} + \mathit{dist}(l_c(\nu), p(r))$.

For each vehicle ν in the input, we process a *vehicle startup event* at $t_{\mathrm{serv}}^{\mathrm{min}}(\nu)$ and a *vehicle shutdown event* at $t_{\mathrm{serv}}^{\mathrm{max}}(\nu)$. To process the former, we check whether there are already any requests matched to ν. If so, we set ν's state to driving and insert a vehicle arrival event into the queue. Otherwise, we set the state to idling and generate no new event. To process the vehicle shutdown event, we set ν's state to out of service and notify the dispatching algorithm about the vehicle shutdown. Note that all request receipt, vehicle startup and vehicle shutdown events are known in advance and form the initial content of the event queue.

Whenever a vehicle ν reaches a stop, we process a *vehicle arrival event*. To do so, we set ν's state to stopping and add a vehicle departure event at $t_{\mathrm{now}} + t_{\mathrm{stop}}$ to the queue. Moreover, we notify the dispatching algorithm about the vehicle arrival so that ν's route (and preprocessed data) can be updated. Finally, whenever a vehicle ν is ready to depart from a stop, we process a *vehicle departure event*. To do so, we check whether there are currently any ride requests matched to ν. If so, we set its state to driving and insert a vehicle arrival event into the queue. Otherwise, we set the state to idling and generate no new event.

Parameters. We take the default model parameters from MATSim. The stop time t_{stop} is set to 1 min, the maximum wait time $t_{\mathrm{wait}}^{\mathrm{max}}$ to 5 min, the maximum trip time model parameters α and β to 1.7 and 2 min, the wait time violation weight γ_{wait} to 1, and finally the trip time violation weight γ_{trip} to 10.

CH preprocessing is taken from the open-source library RoutingKit[17]. We use the partitioning algorithm Inertial Flow [SS15] to compute a CCH order, with the balance parameter b set to 0.3. CH preprocessing and CCH order computation take less than one second each on the Berlin network. On the Rhine-Ruhr network, the former takes 4 seconds and the latter takes 6 seconds. For smaller search spaces, we apply the more sophisticated perfect CCH customization algorithm [DSW16].

6.4.2 Elliptic Pruning

We start by evaluating the effectiveness and efficiency of elliptic pruning. Table 6.2 shows the reduction in search-space size achieved by conditions (a) and (b) from Theorem 6.2. The average unpruned CH search space contains roughly 210 vertices on the Berlin instances and 240 vertices on the Ruhr instances. Only 25 % of them satisfy condition (a), and even less than 10 % satisfy condition (b). When combined, they decrease the average search-space size (and thus the number of bucket entries)

[17] `https://github.com/RoutingKit/RoutingKit`

Table 6.2: Bucket entry generation on various benchmark instances with standard and customizable CHs. We report the total number of vertices v in the search space of a newly inserted stop s with neighboring stop s'. We also report those that are the highest-ranked vertex on all shortest paths between s and v (i.e., satisfy condition (a)), those that lie inside the shortest-path ellipse around s and s' (i.e., satisfy condition (b)), and those that satisfy both conditions. Moreover, we report the number of bucket entries inserted, the running time for the search from the new stop, the search from its neighbor, the propagation of distance labels, and the total running time.

		# vertices in search space				#	running time [μs]			
input	CH	total	highest	ellipse	both	entries	stop	neigh	prop	total
Berlin	std	210.37	54.54	16.90	9.87	9.87	4.33	3.61	2.24	10.17
1pct	cust	186.63	136.63	15.50	12.49	12.50	2.66	2.85	2.21	7.72
Berlin	std	210.64	54.66	14.05	8.72	8.73	4.03	3.35	1.99	9.37
10pct	cust	186.76	136.35	13.20	10.84	10.84	2.49	2.66	1.95	7.10
Ruhr	std	241.09	54.38	15.67	8.70	8.71	3.73	3.35	2.32	9.40
1pct	cust	228.91	165.20	13.67	11.42	11.42	3.00	3.22	3.05	9.26
Ruhr	std	241.58	54.46	14.25	8.43	8.43	3.46	3.11	2.09	8.66
10pct	cust	228.91	165.26	12.88	10.90	10.90	2.77	2.95	2.53	8.25

by a factor of more than 20. With CCHs, condition (a) prunes significantly less vertices. However, as condition (b) still prunes more than 90 % of the vertices, the number of bucket entries is about the same as with standard CHs. Moreover, recall that, whenever we insert an entry into a bucket $B(v)$, we also insert a corresponding entry into $B(parent(v))$, in order to simplify bucket entry removal. We observe that this almost never inserts additional bucket entries. The time to generate (source or target) bucket entries for a new stop is divided roughly equally between the search from the new stop, the search from its neighbor, and the propagation of the distance labels of the latter search into the search space of the former search.

Table 6.3 shows the performance of BCH searches and bucket entry removal. Due to elliptic pruning, BCH searches scan relatively few bucket entries, and are therefore very fast. On Berlin-1pct, a BCH search takes merely 15 microseconds. On Berlin-10pct, where we have 10 times more vehicles and 9 times more ride requests, the running time doubles with standard CHs, and triples with CCHs. On Ruhr-10pct, our largest and densest instance, a BCH search takes 65 microseconds with standard CHs and twice as much with CCHs. Since we need four BCH searches per ride request, this makes at most half a millisecond, fast enough for interactive applications. Taking merely a few microseconds, the time spent on bucket entry removal is negligible.

Table 6.3: Time (in microseconds) for BCH searches and bucket entry removal on various benchmark instances with standard and customizable CHs. We also report the number of vertices and bucket entries visited during a BCH search and while removing bucket entries referring to a completed stop.

		BCH searches			bucket entry removal		
input	CH	# vertices	# entries	time	# vertices	# entries	time
Berlin	std	62.87	564.16	14.94	25.72	149.54	1.21
1pct	cust	186.65	1 331.91	16.24	46.16	293.23	1.68
Berlin	std	62.94	3 994.05	35.55	23.57	905.88	1.73
10pct	cust	186.66	9 149.83	52.88	42.22	1 764.32	2.61
Ruhr	std	65.80	1 191.24	19.84	19.62	140.86	1.85
1pct	cust	229.15	4 031.66	33.09	41.06	446.38	2.93
Ruhr	std	65.85	7 953.42	65.80	18.61	820.10	2.67
10pct	cust	229.18	25 475.08	833.56	38.38	8 540.68	8.90

6.4.3 Resolving Ride Requests

We next evaluate the performance of the matching algorithm. Table 6.4 reports the time for each of its phases. Recall that LOUD tries only ordinary insertions into vehicles that have been seen during the BCH searches. We observe that this (exact) filter works very well, with less than 5 % of the vehicles passing through in all cases. Consequently, it takes only a few microseconds to try all ordinary insertions. Note that the search for special-case insertions that insert the pickup before and the dropoff after the last stop on a vehicle's route takes up the largest fraction of the total time (60 % on Berlin-10pct, 80 % on the sparser Berlin-1pct, and even almost 95 % on Ruhr-1pct). Interestingly, the total time on Berlin is always between 600 and 700 microseconds, although it is divided differently between the phases depending on the sparsity of the vehicles and ride requests. On the Rhine-Ruhr benchmark instances, we observe that the running times are around 3 milliseconds.

Table 6.5 reports detailed statistics about the special-case treatments within LOUD. Recall that LOUD discards as many insertions before the next scheduled stop as possible using cheap lower bounds on the pickup detour, in order to avoid costly extra CH queries. We observe that these lower bounds work very well. On average, we only need a single extra CH query per ride request on Berlin, and roughly two extra queries on the Rhine-Ruhr instances. Note that the number of insertions tried can be smaller than the number of last stops visited, because we try an insertion $(\nu, r, i, |R(\nu)| - 1)$ with $i < |R(\nu)| - 1$ only if $\nu \in C$ and ν does not arrive fully occupied at its last stop.

Table 6.4: Performance of resolving ride requests on various benchmark instances with standard and customizable CHs. We report the time to compute the shortest direct path from the pickup to the dropoff spot, the time for the BCH searches, the time to try all ordinary candidate insertions, the time to treat the special cases (pickup before the next stop, pickup after the last stop, and dropoff after the last stop), the time to update the preprocessed data (including bucket entry generation), and the total running time. All running times are given in microseconds. In addition, we report the size of the superset C of promising candidate vehicles.

				ordinary insertions		special insertions				
						pickup	pickup	dropoff		
input	CH	direct	BCH	$\lvert C\rvert$	time	at beg	at end	at end	upd	total
Berlin	std	11.0	60.8	48	1.7	9.8	9.6	555.7	45.0	693.6
1pct	cust	8.4	66.0	48	1.7	8.8	9.7	562.3	35.1	692.0
Berlin	std	10.7	143.3	277	20.9	21.9	5.4	379.8	42.1	624.0
10pct	cust	8.0	213.9	280	20.8	20.8	5.2	369.8	33.5	672.0
Ruhr	std	11.0	80.6	118	4.9	25.2	34.3	3 308.3	42.7	3 506.9
1pct	cust	10.0	134.0	117	5.2	28.4	34.9	3 376.7	41.9	3 631.1
Ruhr	std	10.2	264.8	666	50.7	70.8	12.8	1 977.5	40.8	2 427.5
10pct	cust	9.0	536.1	661	51.4	74.0	12.4	2 019.3	39.9	2 742.0

6.4.4 Comparison to Related Work

Comparing running times is often difficult, due to different machines, benchmark instances, and programming skills. In addition, objectives and constraints in dynamic ridesharing come in a wide variety. For a fair comparison, we carefully reimplemented one competitor and run it on the same machine and instances. We choose the dispatching algorithm in MATSim for various reasons.

First, MATSim uses exactly the same problem formulation. Second, since MATSim is actually used in industry and academia, the comparison of LOUD to MATSim is of particular practical relevance. Third, since the code of MATSim is publicly available, there are no unclear implementation details. Fourth, the running times reported by the algorithms mentioned in Section 6.1 are roughly similar. On a benchmark instance comparable to Berlin-10pct, the algorithm by Huang et al. [HBJW14] takes between 10 and 100 milliseconds to process a ride request. For their simulated-annealing algorithm, Jung et al. [JJP16] report running times of 174–257 milliseconds per request (on a much smaller instance). Unfortunately, T-Share [MZW13] does

Table 6.5: Detailed statistics about the special-case treatments on various benchmark instances with standard and customizable CHs. For each special-case treatment, we report the number of insertions tried and the running time (in microseconds). For handling pickups before the next stop, we additionally report the number of CH queries needed per ride request. For handling pickups and dropoffs after the last stop, we additionally report the number of last stops visited during the reverse Dijkstra searches from the pickup and dropoff spot, respectively.

		pickup at beginning			pickup at end			dropoff at end		
input	CH	insert	qy	time	stop	insert	time	stop	insert	time
Berlin	std	69.7	0.80	9.8	1.5	1.5	9.6	120.9	18.1	555.7
1pct	cust	70.4	0.79	8.8	1.5	1.5	9.7	120.9	17.9	562.3
Berlin	std	582.9	0.80	21.9	3.9	3.9	5.4	731.0	100.9	379.8
10pct	cust	585.6	0.80	20.8	3.9	3.9	5.2	731.0	99.4	369.8
Ruhr	std	184.4	1.76	25.2	1.7	1.7	34.3	302.1	26.3	3 308.3
1pct	cust	183.5	1.76	28.4	1.7	1.7	34.9	302.1	25.9	3 376.7
Ruhr	std	1 329.6	2.57	70.8	2.4	2.4	12.8	1 795.8	111.3	1 977.5
10pct	cust	1 317.2	2.59	74.0	2.4	2.4	12.4	1 795.8	109.2	2 019.3

not report any absolute running times. Our MATSim reimplementation takes 14 and 18 milliseconds per request on Berlin-1pct and Berlin-10pct, respectively; see Table 6.6 for details. Note that this is 15 times faster than the official MATSim code.

Table 6.7 compares LOUD to the dispatching algorithm in MATSim. Besides a reimplementation of the original heuristic algorithm (MATSim-h), we also consider an exact variant (MATSim-e). Recall that the filtering phase tries all possible insertions into each vehicle's route, where all needed detours are estimated using geometric distances. The travel time between any two vertices is given by $(\sigma_{\text{dist}} \cdot \mu)/(\sigma_{\text{spd}} \cdot v_{\text{veh}})$, where μ is the straight-line distance, v_{veh} is the estimated vehicle speed, and σ_{dist} and σ_{spd} are parameters. MATSim-h (in accordance with the official code) sets the parameters $(v_{\text{veh}}, \sigma_{\text{dist}}, \sigma_{\text{spd}})$ to $(30\,\text{km/h}, 1.3, 1.5)$. MATSim-e sets v_{veh} to the maximum travel speed that occurs in the network, and both σ_{dist} and σ_{spd} to 1.

We observe that LOUD is 30 times (20 times) faster than MATSim-h on Berlin-10pct (Berlin-1pct). On Ruhr-10pct, we even see a speedup of around 45. Since MATSim-e and both LOUD variants are exact algorithms, all three make the same matching decisions, and thus obtain the same solution quality. Interestingly, note that although MATSim-h does not find the best insertion for each individual ride request, it obtains slightly better wait times in total on Berlin-10pct.

Table 6.6: Performance of resolving ride requests on various benchmark instances with the heuristic MATSim algorithm and its exact variant. We report the time for the filtering phase, the search to the pickup, the search from the pickup, the search to the dropoff, the search from the dropoff, the evaluation phase, and the total running time. All running times are given in milliseconds. Moreover, we report the number of insertions tried during the filtering phase, as well as the number of filtered insertions.

		geometric filtering			Dijkstra searches				eval	
input	var	tried	filtered	time	to p	fr p	to d	fr d	time	total
Berlin	heu	1 811	101	0.26	3.5	3.5	3.6	2.9	0.01	13.75
1pct	ex	1 811	1 354	0.30	5.0	4.7	4.6	4.6	0.05	19.29
Berlin	heu	18 006	386	2.26	4.0	4.1	4.1	3.7	0.03	18.18
10pct	ex	18 009	12 708	3.24	5.1	4.8	4.7	4.8	0.43	23.07
Ruhr	heu	5 706	53	1.23	14.4	17.8	19.5	9.2	0.01	62.16
1pct	ex	5 859	3 366	1.45	50.7	50.1	49.5	49.3	0.38	201.49
Ruhr	heu	50 620	126	10.63	20.4	27.1	31.8	18.3	0.04	108.14
10pct	ex	52 474	32 374	14.53	50.5	52.2	50.5	51.7	3.69	223.05

On Ruhr-10pct, MATSim-h achieves both better wait and ride times than the three exact algorithms. Note, however, that the vehicles operate less efficiently. That is, MATSim-h utilizes the fleet under the given constraints worse than the three others. Therefore, MATSim-h actually obtains a worse solution quality on Ruhr-10pct.

6.4.5 Integrating LOUD into the MATSim Software Package

MATSim is a full-fledged software package for transport simulations that is currently used in industry and academia. It offers support for a wide variety of transportation types, including driving, walking, transit, cycling, and ridesharing. In the previous section, we reimplemented the algorithm used by MATSim to dispatch shared taxi-like vehicles and compared it to our LOUD algorithm. In this section, we go the other way around and integrate our LOUD implementation into the MATSim software.

MATSim simulates the movement of each inhabitant (also called *agent*) in the study area. For each agent, MATSim maintains a set of alternative *day plans*, which consist of a sequence of activities at different locations and trips between these locations. Trips can use driving, walking, transit, cycling, ridesharing systems, and more. For each trip, a day plan contains the full route that the agent will take. Moreover, a *score* is associated with each day plan that represents the plan's fitness or attractiveness.

Table 6.7: Comparison of LOUD to the heuristic MATSim algorithm (and its exact variant). We report the average running time per request and statistics about the solution quality. For requests, we report the average and 95th percentile of the wait times, and the average ride and trip time. For vehicles, we report the average time spent driving empty (i.e., without riders), spent driving occupied, spent picking up or dropping off riders, and the average operation time.

			request statistics [m:s]				vehicle statistics [h:m]			
		time	wait		ride	trip	emp	occ	stop	op
input	algorithm	[ms]	avg	95 %						
Berlin	MATSim-h	13.8	4:11	8:21	14:11	18:22	0:35	3:19	0:33	4:27
1pct	MATSim-e	19.3	4:12	8:20	14:11	18:23	0:36	3:19	0:33	4:28
	LOUD-CH	0.7	4:12	8:20	14:11	18:23	0:36	3:19	0:33	4:28
	LOUD-CCH	0.7	4:12	8:20	14:11	18:23	0:36	3:19	0:33	4:28
Berlin	MATSim-h	18.2	3:45	8:21	14:52	18:36	0:14	2:31	0:29	3:14
10pct	MATSim-e	23.1	3:47	8:13	14:51	18:38	0:13	2:31	0:29	3:13
	LOUD-CH	0.6	3:47	8:13	14:51	18:38	0:13	2:31	0:29	3:13
	LOUD-CCH	0.7	3:47	8:13	14:51	18:38	0:13	2:31	0:29	3:13
Ruhr	MATSim-h	62.2	5:57	12:35	13:05	19:02	1:09	3:23	0:33	5:05
1pct	MATSim-e	201.5	5:54	12:22	13:52	19:46	1:04	3:19	0:33	4:56
	LOUD-CH	3.5	5:54	12:22	13:52	19:46	1:04	3:19	0:33	4:56
	LOUD-CCH	3.6	5:54	12:22	13:52	19:46	1:04	3:19	0:33	4:56
Ruhr	MATSim-h	108.1	3:36	8:17	13:12	16:49	0:24	2:41	0:30	3:35
10pct	MATSim-e	223.1	3:43	8:43	13:54	17:38	0:22	2:34	0:30	3:25
	LOUD-CH	2.4	3:43	8:43	13:54	17:38	0:22	2:34	0:30	3:25
	LOUD-CCH	2.8	3:43	8:43	13:54	17:38	0:22	2:34	0:30	3:25

The goal of MATSim is to predict the movement of a population, i.e., to generate realistic day plans. To do so, MATSim operates in iterations. Each iteration consists of the three phases replanning, mobsim (for mobility simulation), and scoring. At the beginning of each iteration, each agent selects one of its day plans based on their scores. A certain fraction of the agents is allowed to modify their selected day plan, for example by changing the time or location of an activity or the route or mode of a trip. During the mobsim, the agents move along the routes determined by their day plan. At the end of each iteration, each agent associates a score with their day plan based on how well the plan worked. MATSim stops when the scores have converged. The final day plans then realistically predict the movement of the population. Typically, the scores need several hundred iterations to converge.

We are mainly interested in the mobsim, which moves the agents along their routes as follows. The mobsim associates a FIFO queue with each edge in the road network. Moreover, each edge has a free-flow travel time, a flow capacity, and a storage capacity. During the mobsim, the agents move from queue to queue along their routes. In each time step, the mobsim repeatedly moves the agent at the head of each queue to the tail of the next queue as long as three conditions hold. First, the agent has spent at least the free-flow travel time in the queue. Second, the number of agents removed from the queue in the current time step is below the flow capacity. Third, the number of agents in the next queue is below the storage capacity.

In addition to moving agents from queue to queue along preplanned routes determined by day plans, the mobsim also processes dynamic ride requests. For each request in a time step, the mobsim computes the best insertion and modifies the planned route of the selected vehicle accordingly. The shared vehicles themselves move through the queues along their route as any other agent.

MATSim is written in the Java programming language. Its functionality is organized into modules called *contributions*. Support for ridesharing systems is provided by the `drt` contribution. At its heart is a component called `DefaultDrtOptimizer`, which is invoked whenever a ride request is received. It picks a vehicle and modifies its route accordingly or rejects the request. To make LOUD accessible within MATSim, we implement a `LoudDrtOptimizer` that resembles the functionality of the `DefaultDrtOptimizer`, but uses LOUD rather than the built-in dispatching algorithm.

We do not reimplement LOUD in Java. Instead, the `LoudDrtOptimizer` accesses our native LOUD implementation through the Java Native Interface (JNI). The native code takes the pickup and dropoff spot and the earliest departure time and returns the best insertion, including the full paths to the pickup, from the pickup, to the dropoff, and from the dropoff. Note that the vehicle routes are maintained twice. The native code needs the routes for the matching decisions and maintains them as discussed in Section 6.3.1. The Java code needs the routes to move the shared vehicles through the queues. To keep implementation complexity low, we reuse the existing Java code for maintaining and modifying the routes in Java.

Table 6.8 compares the running time and solution quality of the mobsim when using the built-in dispatching algorithm, our reimplementations of the built-in algorithm, and our LOUD implementations. We consider each instance without and with traffic. Note that in this experiment, we make matching decisions based on free-flow travel times. That is, all computed insertions and paths are optimal with respect to the free-flow metric. Without traffic (○), all agents move at free-flow speed through the queues. We achieve this by multiplying the flow capacities and storage capacities by 100, which essentially makes traffic congestion impossible, since the flow and storage capacities are never reached. With traffic (●), the agents may get stuck in traffic jams, leading to delays in the arrival of some agents.

Table 6.8: Running time and solution quality of the MATSim mobsim when using the built-in dispatching algorithm, our reimplementations of the built-in algorithm, and our LOUD implementations. We consider various benchmark instances, each without traffic (vehicles travel at free-flow speed) and with traffic (travel speeds depend on how many vehicles are on a road segment). For requests, we report the average and 95th percentile of the wait times, and the average ride and trip time. For vehicles, we report the average distance driven empty, occupied, and in total.

				request stats [m:s]				vehicle stats [km]		
			time	wait		ride	trip	emp	occ	tot
input	traf	algorithm	[h:m]	avg	95 %					
Berlin 1pct	○	built-in	0:59	4:15	8:44	14:44	19:00	11.9	91.9	104
		MATSim-h	0:07	4:16	8:35	14:32	18:48	12.2	91.8	104
		MATSim-e	0:08	4:17	8:34	14:32	18:50	12.3	91.7	104
		LOUD-CH	0:03	4:16	8:35	14:33	18:49	12.1	91.5	104
		LOUD-CCH	0:03	4:17	8:36	14:32	18:49	12.3	91.8	104
Berlin 1pct	●	built-in	0:56	5:48	14:08	19:05	24:53	13.0	92.1	105
		MATSim-h	0:07	5:50	13:50	18:52	24:42	13.9	92.6	106
		MATSim-e	0:09	5:53	14:14	18:50	24:42	14.1	92.3	106
		LOUD-CH	0:03	5:58	14:35	18:45	24:43	13.9	92.4	106
		LOUD-CCH	0:03	5:51	14:10	18:43	24:33	13.8	92.3	106
Berlin 10pct	○	built-in	12:41	3:53	8:53	16:30	20:24	3.8	64.8	69
		MATSim-h	1:10	3:59	8:50	15:50	19:49	4.5	66.6	71
		MATSim-e	1:25	4:01	8:42	15:49	19:50	4.3	66.5	71
		LOUD-CH	0:29	4:02	8:46	15:50	19:52	4.4	66.5	71
		LOUD-CCH	0:29	4:02	8:47	15:46	19:48	4.4	66.5	71
Berlin 10pct	●	built-in	12:57	4:34	10:42	19:28	24:02	4.2	65.4	70
		MATSim-h	1:13	5:01	11:15	19:34	24:35	5.1	67.6	73
		MATSim-e	1:27	5:11	11:21	19:37	24:48	5.1	67.5	73
		LOUD-CH	0:32	5:18	11:35	19:43	25:01	5.1	67.7	73
		LOUD-CCH	0:32	5:18	11:30	19:37	24:55	5.2	67.5	73

MATSim reports the running time for each phase (replanning, mobsim, scoring) of an iteration separately. However, it does not further subdivide the time of the mobsim. Therefore, the running time reported in Table 6.8 includes not only the time for the matching decisions but also the effort to maintain the vehicle routes in Java and to move the agents of *all* transportation types through the queues. That is, we cannot expect to see the speedups reported in the previous section in this experiment.

On Berlin-1pct, we decrease the total running time of the mobsim from around one hour to only three minutes. Given that we replaced only the dispatcher for the ridesharing system (and reuse any other code as is), this is a considerable improvement. Likewise, the running time on Berlin-10pct is significantly reduced from 13 hours to half an hour, again by replacing only the dispatching algorithm. When using our LOUD implementation within MATSim, the matching decisions are no longer the performance bottleneck of the mobsim.

We observe that the three mobsim variants based on MATSim-e, LOUD-CH and LOUD-CCH obtain slightly different request and vehicle statistics. This may be surprising because all three are exact algorithms and thus should make the same matching decisions. The reason for the divergence is that shortest paths generally are not unique. Even when MATSim-e, LOUD-CH and LOUD-CCH obtain the same insertion, they can still return different paths to and from the pickup and dropoff spot. The actual paths, however, can affect all subsequent ride requests, since the detour to service a request depends on the current locations of the vehicles, which in turn depend on the actual paths of the vehicles.

We do not run into such trouble in our discrete-event simulation, since we essentially teleport the vehicles from stop to stop rather than moving them along their routes. When the location of a vehicle currently driving from stop s to stop s' is needed, we run a CH search from s to s', retrieve the actual path, and traverse the path (starting at the departure time at s) until we reach the current point in time. Since we use the same method to retrieve the current location for all three algorithms, the matching decisions do not diverge, and the solution quality is the same.

6.5 Conclusion

We presented LOUD, a novel algorithm for large-scale dynamic ridesharing. Unlike most competitors, we do not require a huge number of calls to Dijkstra's algorithm, but adapt a modern route planning technique developed for the many-to-many problem (bucket-based contraction hierarchies). Our experiments on the Open Berlin Scenario with 10 000 vehicles and more than 100 000 ride requests show that LOUD answers a request in less than a millisecond, which is 30 times faster than current algorithms. This gives plenty of leeway for interactive applications on cities even larger than Berlin. For transport simulations, LOUD is even more important. Since simulators process each request hundreds of times, running time is an even bigger issue than in interactive applications, and requests cannot be answered "fast enough".

Since the special-case treatments take up the largest fraction of the running time, it would be interesting to eliminate the two remaining (local) Dijkstra searches. A possible approach would be to maintain *additional* buckets that store the *unpruned* forward CH search spaces of the ends of the current vehicle routes. Note that we cannot apply elliptic pruning because the leeway is unbounded. Instead, we can keep the buckets sorted (e.g., using search trees), which allows us to stop a bucket scan when we visit an entry that cannot possibly yield an insertion better than the best one so far encountered in any previous phase.

Parallelization could also be a key to better performance. Most likely, this would be a combination of fine-grained parallelism and parallelization over several requests. Independent of the internals of LOUD, the main issue here is that a change caused by an earlier request can affect all subsequent requests. Therefore, it would be interesting to investigate how independent requests can be identified or alternatively how dependencies can be detected and repaired. One could also study to what extent certain dependencies can be ignored without severely affecting solution quality.

Finally, it would be interesting to increase the solution space. For example, one could allow requests already matched to a vehicle to be reordered or moved to a different vehicle. Another interesting project are variable pickup and dropoff spots, where riders agree to walk a short distance to a location where it is more efficient to pick them up or drop them off (e.g., on main roads rather than in traffic-calmed areas). We believe that the techniques developed for LOUD might be key ingredients for such generalized systems that promise higher overall solution quality.

7 Turn Costs and Restrictions

So far, we have ignored turn restrictions and turn costs. There are two common ways to represent turn costs and restrictions. The edge-based model expands the network so that road segments become vertices and allowed turns become edges. The compact model keeps intersections as vertices, but associates a turn table with each vertex. Although CCHs can be used as is on the edge-based model, the performance of preprocessing and customization is severely affected. While the expanded network is only three times larger, both preprocessing and customization time increase by up to an order of magnitude. In this chapter, we carefully engineer customizable contraction hierarchies to exploit different properties of the expanded graph. We reduce the increase in customization time from up to an order of magnitude to a factor of about 3. The increase in preprocessing time is reduced even further. Moreover, we present a CCH variant that works on the compact model, and show that it performs worse than the variant on the edge-based model. Surprisingly, the variant on the edge-based model even uses less space than the one on the compact model, although the compact model was developed to keep the space requirement low.

This chapter is based on joint work with Dorothea Wagner, Tim Zeitz and Michael Zündorf [BWZZ20].

7.1 Introduction

Motivated by computing driving directions, the last two decades have seen intense research on speedup techniques [Bas+16] for Dijkstra's algorithm [Dij59], which rely on a slow preprocessing phase to enable fast queries. Almost all previous experimental studies (e.g., [GH05, HKMS09, Lau09, DSW16, Gut04, ADGW11, ALS13, BFSS07]) are restricted to the simplified model, where vertices represent intersections, edges represent road segments, and turn costs are ignored. While it has been widely believed that turn costs and restrictions are easy to incorporate, Delling et al. [DGPW17] show that most algorithms have a significant performance penalty. For long-range queries, one may argue that turn costs are negligible. When analyzing intracity traffic (as in Chapter 5) or dispatching vehicles operating within a particular city (as in Chapter 6), however, taking turn costs into account is of utmost importance.

A fairly recent development in the area of route planning are customizable speedup techniques, which split preprocessing into a slow metric-independent part, taking only the network structure into account, and a fast metric-dependent part (the *customization*), incorporating edge costs (the *metric*). Customizable route planning (CRP) [DGPW17] and customizable contraction hierarchies (CCHs) [DSW16] are the most prominent among them, and are both used in commercial and research software. While CRP was developed with turn costs in mind, CCHs were not. In this chapter, we incorporate turn costs and restrictions into CCHs.

Related Work. Turns can be encoded into the network structure by expanding the network so that road segments become vertices and allowed turns become edges [Cal61, Win02]. This is known as the *edge-based model* [Bas+16]. While any speedup technique can work on an expanded network, some are more robust than others [DGPW17]. We are aware of two algorithms that have been tailored to handle turn costs and restrictions. First, Geisberger and Vetter [GV11] present a turn-aware version of (non-customizable) contraction hierarchies (CHs) [GSSV12]. Second, Delling et al. [DGPW11] develop CRP with turns in mind. Both independently proposed a different turn representation. The *compact model* keeps intersection as vertices, but associates a *turn table* with each vertex.

Our Contribution. The contribution of this chapter is twofold. First, we propose several optimizations that accelerate CCHs on the edge-based model by exploiting properties of the expanded network. We reduce the increase in customization time from up to an order of magnitude to a factor of about three (which is reasonable since the expanded network is three times larger than the original network, which ignores turn costs). The increase in preprocessing time is reduced even further.

Second, we introduce a CCH variant that works on the compact model, and discuss various issues we found. An extensive experimental evaluation shows that the edge-based variant significantly outperforms the compact variant. Surprisingly, the variant on the edge-based model even uses less space than the one on the compact model.

Outline. Section 7.2 formally defines the problem we solve and has background information. Section 7.3 presents optimizations that accelerate CCHs on the edge-based model. Section 7.4 introduces a CCH variant that works on the compact model. Section 7.5 presents an extensive experimental evaluation and comparison of both variants. Section 7.6 concludes with final remarks.

7.2 Preliminaries

We are given a directed graph $G = (V, E)$ where vertices represent intersections and edges represent roads. A cost function $\ell : E \longrightarrow \mathbb{R}_{\geq 0}$ assigns an arbitrary cost to each edge. We are also given two functions $r : E \times E \longrightarrow \{0, 1\}$ and $c : E \times E \longrightarrow \mathbb{R}_{\geq 0} \cup \{\infty\}$. If $r(e, f) = 0$, the head of e is the tail of f and the turn from e to f is allowed. The cost of the turn is given by $c(e, f)$. Note that r and c have to be *consistent*, i.e., $r(e, f) = 1$ implies $c(e, f) = \infty$. Since r depends on the network topology, it is part of the input to the preprocessing phase. The turn cost function c is part of the input to the customization since it depends on the traffic situation and personal preferences.

A path P from a point along an edge e_0 to a point along an edge e_k is a triple that consists of a sequence of edges $\langle e_0, \ldots, e_k \rangle$ with $r(e_i, e_{i+1}) = 0$, a real-valued offset $o_0 \in [0, 1]$ on the source edge e_0, and a real-valued offset $o_k \in [0, 1]$ on the target edge e_k. The cost of a path is the sum of the costs of its constituent edges and turns, i.e., $\ell(P) = (1 - o_0) \cdot \ell(e_0) + \sum_{i=1}^{k}(c(e_{i-1}, e_i) + \ell(e_i)) - (1 - o_k) \cdot \ell(e_k)$. Given a source edge e_s with offset o_s and a target edge e_t with offset o_t, the problem we consider is computing a shortest path from the start point along e_s to the end point along e_t. For simplicity, we assume that $o_s = 1$ and $o_t = 1$ in the rest of this chapter.

In the following, we discuss both common ways to represent turn costs and restrictions. After that, we describe tailored implementations of Dijkstra and CHs that operate on compact graphs. The standard versions of Dijkstra, CHs, and CCHs, which operate on simplified or edge-based graphs, were discussed in Chapter 2.

7.2.1 Turn Representation

The *simplified model* ignores turn costs and restrictions; see Figure 7.1 (left). To actually incorporate them, there are two common ways. We explain each in turn.

Edge-based Model. The *edge-based model* [Cal61, Win02] expands the network so that road segments become vertices and allowed turns become edges; see Figure 7.1 (middle) for an example. More precisely, the edge-based graph $G_e = (V_e, E_e)$ is obtained from G as follows. The vertices of G_e are the edges of G, i.e, $V_e = E$. The edges of G_e are the allowed turns of G, i.e., $E_e = \{(e, f) : e, f \in E, r(e, f) = 0\}$. The

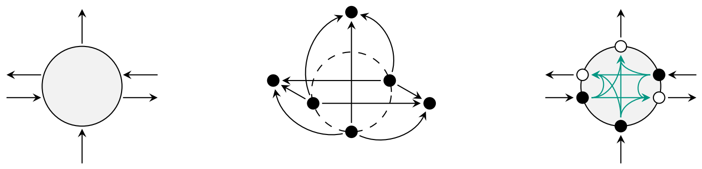


Figure 7.1: Turn representations (from left to right): the simplified model, the edge-based model, and the compact model.

cost of an edge $(e, f) \in E_e$ is defined as $\ell_e(e, f) = c(e, f) + \ell(f)$. The main advantage of the edge-based model is that most route planning algorithms can be used as is on it, without further modifications to the algorithm.

Compact Model. The *compact model* [GV11, DGPW17] keeps intersections as vertices, but associates a $p \times q$ *turn table* T_v with each vertex v, where p and q are the numbers of incoming and outgoing edges, respectively. The entry $T_v(i, j)$ represents the cost of the turn from the i-th incoming edge e to the j-th outgoing edge f, i.e., $T_v(i, j) = c(e, f)$. For each edge (v, w), its tail corresponds to an *exit point* at v and its head corresponds to an *entry point* at w. Note that the entry points in the compact model translate directly to the vertices in the edge-based model; see Figure 7.1 (right) for an example. We denote by $v|i$ the i-th exit (or entry) point at v and by $(v|i, w|j)$ the edge whose tail corresponds to the i-th exit point at v and whose head corresponds to the j-th entry point at w. The main advantage of the compact model is its low space overhead since turn tables can be shared among vertices (the number of distinct turn tables for continental instances such as the road network of Western Europe used in our experiments is in the thousands rather than millions [DGPW17]).

7.2.2 Dijkstra's Algorithm on the Compact Model

Recall that on standard graphs, Dijkstra's algorithm computes shortest-path distances from a source vertex to all other vertices by scanning them in increasing order of distance. On compact graphs, we must work on entry points instead of vertices. That is, we maintain a distance label $d(v|i)$ for each entry point $v|i$ and a queue Q of unsettled entry points. Initially, $d(s|i) = 0$ for the entry point $s|i$ corresponding to the head of the source edge, $d(v|j) = \infty$ for all other entry points $v|j$, and $Q = \{s|i\}$. To settle an entry point $v|i$, we set $d(w|k) = \min\{d(w|k), d(v|i) + T_v(i, j) + \ell(e)\}$ for each outgoing edge $e = (v|j, w|k)$. Each entry point is settled at most once, however, each vertex can be visited multiple times. Note that Dijkstra's algorithm on the compact model essentially simulates the execution on the edge-based model.

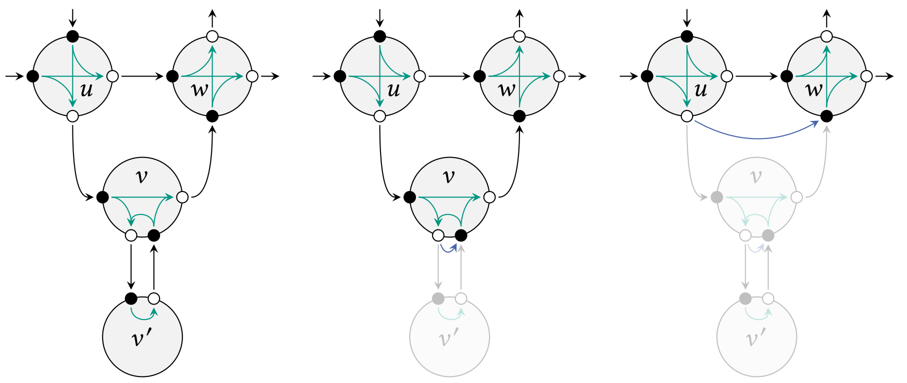


Figure 7.2: Vertex contraction on the compact model. Original edges are shown in black, turn edges are shown in green, and shortcut edges are shown in blue. Each original edge and each right-, left- and U-turn movement has cost 1. Each through movement has cost 10. Left: A subgraph before contraction. Middle: Contracting vertex v' creates a self-loop at v (cost 3). Right: Contracting v creates a shortcut edge between u and w (cost 7), resulting in two parallel edges between them.

7.2.3 Contraction Hierarchies on the Compact Model

Recall from the previous section that we must maintain and compute distance labels for entry points (rather than vertices) in the compact model. Therefore, when contracting a vertex v, we need to preserve the distances between all entry points in the remaining graph (without v). In general, we cannot avoid self-loops and parallel edges. See Figure 7.2 for an example. Contracting vertex v' creates a self-loop at vertex v, because the through movement from v's left entry point to its right exit point is costlier than the path via v'. Analogously, contracting v results in two parallel edges between vertices u and w. When entering u from the west and leaving w to the east, the shortest path is via v. In contrast, when entering u from the north and leaving w to the north, it is better to traverse the edge between u and w.

Self-loops make the computation of shortcuts more complicated. Each shortcut is no longer a concatenation of exactly two edges, but can also include one or more self-loops at the middle vertex. For example, in Figure 7.2, the shortcut between u and w includes the self-loop at v. Therefore, Geisberger and Vetter [GV11] use the witness search (the local Dijkstra) not only to decide whether a shortcut is necessary but also to compute the cost of the shortcut.

More precisely, to contract a vertex v, they run a witness search for each exit point $u|i$ such that there is at least one incoming edge $(u|i, v)$. Initially, the authors set $d(v'|j) = \ell(e)$ for each edge $e = (u|i, v'|j)$. Moreover, each entry point $v'|j$ is inserted into the queue. The witness search stops when each entry point $w|l$ such that there is at least one edge $(v, w|l)$ has been settled. A shortcut $s = (u|i, w|l)$ is only added if it is built from an edge $(u|i, v)$, zero or more self-loops at v, and an edge $(v, w|l)$. The shortcut has cost $\ell(s) = d(w|l)$.

The query phase runs a bidirectional version of the turn-aware Dijkstra described above, but does not relax edges leading to lower-ranked vertices. Note that the stall-on-demand optimization can also be applied in the compact model, as discussed in more detail by Geisberger and Vetter [GV11].

7.3 CCHs on the Edge-Based Model

Although customizable contraction hierarchies can be used without further modifications on the edge-based model, both preprocessing and customization have a significant performance penalty. In this section, we present several acceleration techniques that exploit properties of edge-based graphs. We describe each optimization in turn, but they can be combined in an actual implementation.

Contraction Order. The most straightforward approach to compute a vertex order for an edge-based graph is to pass the edge-based graph to a standard CCH ordering algorithm. We call a contraction order obtained in this way an *edge-based order*. Unfortunately, as our experiments will show, ordering takes over an order of magnitude longer on edge-based graphs (although they are only three times larger).

For better performance, we exploit the fact that the vertices of an edge-based graph G_e are the edges of the corresponding input graph G. More precisely, instead of computing a vertex order for G_e, we obtain an edge order for G. To do so, we use a standard CCH ordering algorithm as a black box to compute a vertex order for G (as usual) and order the edges of G by the rank of their tail. The order of edges with the same tail is arbitrary. Note that if a set $X \subseteq V$ separates the sets $A \subseteq V$ and $B \subseteq V$ in G, then the set $\{(v, w) \in V_e : v \in X\}$ separates the sets $\{(v, w) \in V_e : v \in A\}$ and $\{(v, w) \in V_e : v \in B\}$ in G_e. We call a contraction order obtained in this way a *vertex-based order*. While vertex-based orders can be computed much faster than edge-based orders, they are of rather poor quality.

A closer look at vertex-based orders shows that their computation is needlessly complicated. The idea behind all CCH ordering algorithms is as follows. First, they compute a cut in the input graph. Second, this cut is transformed into a vertex separator (e.g., by picking the endpoints of the cut edges on the source or sink side [HS18]). Third, the ordering algorithms assign the highest ranks to the separator vertices, remove them from the graph, and recurse on the resulting connected components.

When computing vertex-based orders, each cut is first transformed into a vertex separator and then back into an edge set. We can do better by omitting the two transformations. More precisely, we propose to compute an edge order on the input graph as follows: Compute a cut in the input graph, assign the highest ranks to the cut edges, remove them from the input graph, and recurse on the resulting connected components. As discussed before, an edge order on the input graph is equivalent to a vertex order on the edge-based graph. We call an order obtained in this way a *cut-based order*. As our experiments will show, cut-based orders can be computed as fast as vertex-based orders and are as good as edge-based orders.

Infinite Shortcuts. CCH preprocessing views the input graph as an undirected graph. Directions are only taken into account in the customization phase (and during queries), which associates two costs (upward and downward) with each edge. Note that a one-way road segment can be modeled by setting the cost of the reverse direction (i.e., the forbidden direction) to ∞.

Due to this workflow, CCHs can contain shortcuts to which customization always assigns a cost of infinity, independent of the input metric. Note that customization associates a finite cost with a shortcut (v, w) if and only if there is a path from v to w in the input graph with all intermediate vertices ranked lower than v and w. If there is no such path, the shortcut (v, w) is always assigned infinite cost. See Figure 7.3 for an example. We call such edges *infinite shortcuts*.

Infinite shortcuts are necessary neither in the customization phase nor during queries. Therefore, we can remove them from the CCH during metric-independent preprocessing. We do so by simulating a customization phase where each edge in the input graph is assigned a cost of zero. Each edge in the CCH that has infinite cost after the simulated customization is removed. Since CCHs store a directed edge and its reversal as a single undirected edge, we can remove an edge $\{v, w\}$ only if both (v, w) and (w, v) are infinite shortcuts. Note that the reason to remove infinite shortcuts is that we want to reduce the number of triangles in the CCH, since the customization time is essentially linear in the number of triangles.

Standard (vertex-based) graphs are almost bidirected, since one-way road segments are in the minority. Therefore, most edges in vertex-based CCHs have finite cost in both directions, and infinite shortcuts are rare. In contrast, edge-based graphs contain much more edges for which the reversal is not present. Hence, edge-based CCHs contain much more infinite shortcuts. For this reason, this optimization is worth the effort only in the presence of turns.

Directed Hierarchies. Removing each undirected edge $\{v, w\}$ with (v, w) and (w, v) being infinite shortcuts slightly reduces the number of triangles and thus

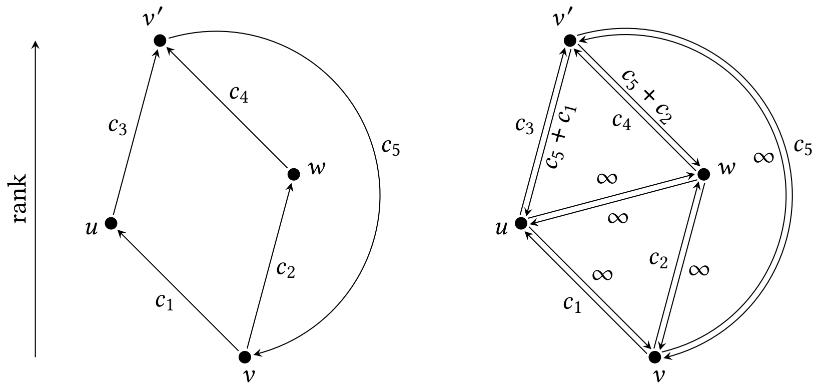


Figure 7.3: An example of a CCH with infinite shortcuts. Left: The input graph with arbitrary edge costs $c_1, \ldots, c_5$. Right: The corresponding CCH after preprocessing and customization. Independent of the metric (the actual values of $c_1, \ldots, c_5$), the shortcuts (u, w) and (w, u) have infinite cost, as there is neither a u–w nor a w–u path in the input graph with all intermediate vertices ranked lower than u.

the customization time. However, for most infinite shortcuts (v, w), the reverse edge (w, v) has finite cost. We therefore propose to store edge-based CCHs as directed graphs rather than undirected graphs. This allows us to remove upward and downward infinite shortcuts independent of each other. When processing a pair $\{v, w\}$ of adjacent vertices during the customization phase, we now enumerate the triangles for (v, w) and (w, v) separately. Note that in this way, no enumerated triangle contains an infinite shortcut, and thus no triangle is provably unnecessary.

Reordering Separator Vertices. Recall that a cut-based order is obtained by computing a cut $X \subseteq E$ in the input graph $G = (V, E)$ that separates G into a source side $A \subseteq V$ and a sink side $B \subseteq V$, assigning the highest ranks to the cut edges, removing them from G, and recursing on the resulting connected components. The order of the cut edges is arbitrary. By choosing it carefully, we significantly increase the number of infinite shortcuts and thus reduce the number of triangles and the customization time (which is essentially linear in the number of triangles).

To obtain an order of the cut edges, we partition X into two blocks X_1 and X_2. The former contains all cut edges that go from the source side to the sink side of the cut, i.e., $X_1 = \{(a, b) \in X : a \in A, b \in B\}$. The latter contains all cut edges that go from the sink side to the source side, i.e., $X_2 = \{(b, a) \in X : a \in A, b \in B\}$. Assume that $|X_1| \geq |X_2|$. We propose to rank all edges in X_1 lower than any edge in X_2.

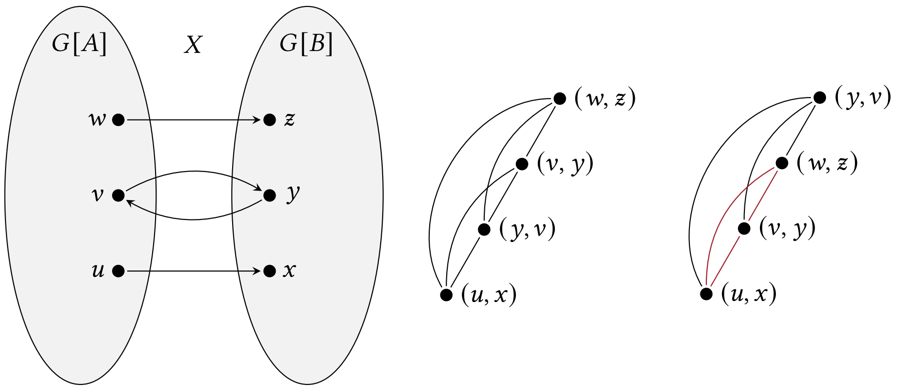


Figure 7.4: Increasing the number of infinite shortcuts in customizable contraction hierarchies. Left: A cut X that separates the input graph into two components $G[A]$ and $G[B]$. Middle: An arbitrary contraction order on the separator vertices in the edge-based graph that correspond to the cut edges in the input graph. Generally, we obtain a complete graph on the separator vertices whose edges have finite cost each. Right: A contraction order in which all separator vertices corresponding to rightward cut edges occur before the separator vertex corresponding to the leftward cut edge. All shortcuts between rightward separator vertices (shown in red) must have infinite cost, since there is no path in the input graph from the right to the left component with all edges ranked no higher than (w, z).

The advantage of this order is that all edges in the hierarchy that connect vertices in X_1 are infinite shortcuts and thus can be removed. Let (u, x) and (v, y) be vertices in X_1 (see Figure 7.4 for an illustration). A shortcut from (u, x) to (v, y) represents a path in G that starts on the edge (u, x) and ends on the edge (v, y). If the shortcut is not an infinite shortcut, then there is a path $\langle u, x, \dots, b, a, \dots, v, y\rangle$ in G with $a \in A$, $b \in B$, and all intermediate edges ranked lower than (u, x) and (v, y). However, since we have $(b, a) \in X_2$, (b, a) is ranked higher than both (u, x) and (v, y), and thus such a path does not exist. Therefore, all edges that connect vertices in X_1 will be removed.

7.4 CCHs on the Compact Model

Recall that all CCH phases do not work on the original directed graph $G = (V, E)$, but on the corresponding bidirected graph $G' = (V, E')$ obtained from G by adding all edges $\{(w, v) : (v, w) \in E, (w, v) \notin E\}$. The cost of each edge in $E' \setminus E$ is ∞, and thus the distance between any two vertices is the same in G and G'. Since most graph-theoretical results for undirected graphs carry over to bidirected graphs, customizable contraction hierarchies can use algorithmic tools for undirected graphs. In particular, CCH preprocessing exploits quotient graphs and CCH queries exploit elimination trees, which are both concepts for undirected graphs.

The compact model, however, is inherently directed. We cannot make a compact graph bidirected, since this would add edges that exit vertices at entry points and enter them at exit points. Therefore, in the compact model, all CCH phases have to work on the original (not necessarily bidirected) graph. This has undesirable consequences. First, we cannot use the efficient CCH preprocessing algorithm based on quotient graphs. Second, we have to use Dijkstra-based queries, since the faster elimination tree queries are also not applicable.

There is one additional issue. Recall that in the compact model, we generally cannot avoid self-loops and multiple parallel edges and that each shortcut is no longer built from exactly two edges, but can also include one or more self-loops at the middle vertex. Standard CHs (on the compact model) deal with this by using the witness searches to determine shortcut costs.

During CCH customization, however, there is no notion of graph searches at all. We enumerate triangles and perform one basic operation for each triangle: adding up the costs of two edges to update the cost of the third edge. Hence, to determine the cost of a shortcut s containing self-loops, we must insert *phantom shortcuts*. These shortcuts are used during customization to incrementally compute the cost of s by repeatedly combining two of its constituent edges.

Preprocessing. Given a nested dissection order on the vertices, we contract them in this order. Whenever contracting a vertex v, we have to add a shortcut between each exit point $u|i$ with $u \neq v$ and $(u|i, v) \in E$ and each entry point $w|l$ with $w \neq v$ and $(v, w|l) \in E$. In addition, as already mentioned, we must add phantom shortcuts, so that the customization phase is able to compute the costs of shortcuts that are built from more than two edges incrementally.

Our approach is as follows. In order to contract a vertex v, we pick an order on the turns at v and contract them in this order. Consider a turn (j, k) at v. For each edge $(u|i, v|j)$ entering v at entry point j and each edge $(v|k, w|l)$ leaving v at exit point k, we add a shortcut $(u|i, w|l)$. Note that these shortcuts are concatenations of two edges, and thus their costs can be customized by enumerating triangles. If $u = v$ or $w = v$, then the shortcut is a phantom shortcut.

Note that this approach adds shortcuts that are not necessary. A shortcut $(u|i, w|l)$ is completely superfluous if $u = v$ and all turns entering exit point i are already contracted, or $w = v$ and all turns leaving entry point l are already contracted. To decide whether a shortcut is necessary, we maintain the number $t(\cdot)$ of uncontracted

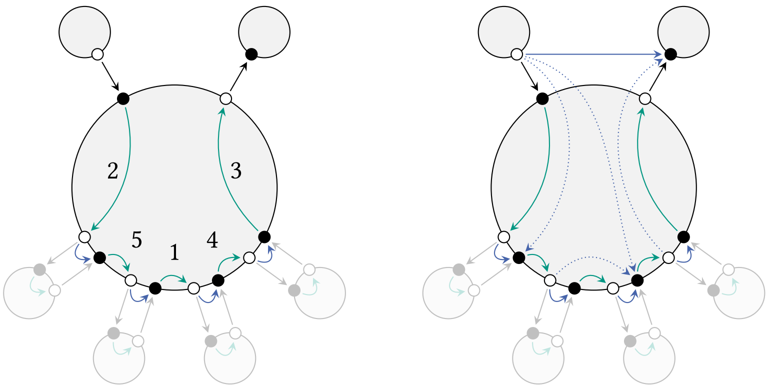


Figure 7.5: Creation of phantom shortcuts. We are about to contract the vertex in the center. Its lower-ranked neighbors (light-colored) are already contracted. Original edges are shown in black, turn edges are shown in green, and shortcut edges are shown in blue. Left: The vertex to be contracted and its neighbors before the contraction. The order on the turns is given by the numbers. Right: The shortcuts added while contracting the turns. Phantom shortcuts are drawn dotted.

turns that enter or leave each exit or entry point of v, respectively. Whenever we contract a turn (j, k), we decrement both $t(j)$ and $t(k)$. A shortcut $(u|i, w|l)$ is only inserted if $u \neq v$ or $t(i) \neq 0$, and $w \neq v$ or $t(l) \neq 0$. See Figure 7.5 for an illustration.

Different turn orders can lead to slightly different numbers of phantom shortcuts. We thus tested some orders on various instances, however, the impact on the performance of all phases was limited. Therefore, any turn order that is easy to implement can be picked, in particular, the order in which the turns are stored in memory.

Customization. We recontract each turn, this time determining shortcut costs. Since we have the CCH topology in place, all we need to do to recontract a turn is to enumerate all triangles spanned by this turn and perform one minimum operation for each triangle. Consider a turn (j, k) at a vertex v and a triangle consisting of three edges $(u|i, v|j)$, $(v|k, w|l)$ and $(u|i, w|l)$; see Figure 7.6 for an illustration of the triangle. We call $(u|i, w|l)$ the *shortcut edge* and the other the *supporting edges* of the triangle. Also, we say that the turn *spans* the triangle.

We recontract the vertices in the given nested dissection order, and within each vertex, we recontract the turns in the same order as during preprocessing. If we pick the order in which the turns are stored in memory, we do not have to store the turn order for each vertex explicitly. For each turn at a vertex v, we enumerate

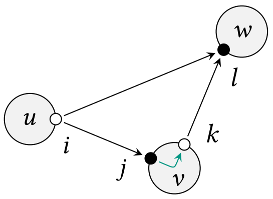


Figure 7.6: A triangle spanned by the turn (j, k) at v. Note that $(u|i, w|l)$ is the shortcut edge, and $(u|i, v|j)$ and $(v|k, w|l)$ are the supporting edges of the triangle.

the triangles spanned by the turn where v is the lowest-ranked vertex, and for each triangle, we add the costs of the two supporting edges and the turn between them, and update the cost of the shortcut edge if needed.

We now show how to efficiently enumerate all triangles spanned by a turn (j, k) where the shortcut edge does not point downwards. The other case is symmetric. We maintain a $|V| \times \Delta$ array W, where Δ is the maximum indegree of the original graph. All values in the array are initialized to ∞. First, we loop over all non-downward edges $e_2 = (v|k, w|l)$ leaving v at k and set $W[w, l] = T_v(j, k) + \ell(e_2)$. Then, we loop through all non-upward edges $e_1 = (u|i, v|j)$ entering v at j. For each such e_1, we loop through all non-downward edges $e = (u|i, w'|l')$ leaving u at i. If $\ell(e_1) + W[w', l'] < \ell(e)$, then we set $\ell(e) = \ell(e_1) + W[w', l']$. Finally, we loop over all edges e_2 again and reset $W[w, l]$ to ∞ for the next enumeration.

Interestingly, a nonturn version of this customization algorithm outperforms the original customization by Dibbelt et al. [DSW16] by a factor of four, and is twice as fast as the engineered customization introduced in Chapter 5. The drawback, on the other hand, is the increase in space consumption.

Queries. Dijkstra-based queries work as in standard CHs on the compact model, however, they do not need to follow phantom shortcuts. Elimination tree queries are not applicable, as elimination trees are defined only for undirected graphs.

7.5 Experiments

In this section, we present our experimental evaluation. Our benchmark machine runs openSUSE Leap 15.1 (kernel 4.12.14), and has 192 GiB of DDR4-2666 RAM and two Intel Xeon Gold 6144 CPUs, each of which has eight cores clocked at 3.5 GHz and 8×64 KiB of L1, 8×1 MiB of L2, and 24.75 MiB of shared L3 cache. Hyper-threading was disabled and parallel experiments use 16 threads. Our code is written in C++ and

Table 7.1: Road networks used for the evaluation our algorithms. The turns column contains the number of allowed turns. It corresponds to the number of edges in the edge-based model. The number of vertices in the edge-based model is equal to the number of edges in the original graph.

	Source	Vertices	Edges	Turns	Turn
		[$\cdot 10^3$]	[$\cdot 10^3$]	[$\cdot 10^3$]	data
Chicago	TNTP	13.0	39.0	135.3	100 s U-Turns
London	PTV	37.0	85.5	137.2	Costs, Restrictions
Stuttgart	PTV	109.5	252.1	394.2	Costs, Restrictions
Europe	DIMACS	17 350.0	39 936.5	106 371.3	100 s U-Turns

compiled with GCC 8.2.1 using optimization level 3.

We implement our algorithms on top of the existing open-source libraries. For CCH, we use the implementation from RoutingKit[18]. We extend it by implementing customization for directed hierarchies and the removal of infinite edges. For the computation of contraction orders, we use InertialFlowCutter[19] [GHUW19] and implement the computation of cut-based orders and the reordering of separator vertices. We publish our extensions to these projects as pull requests on GitHub[20][21]. RoutingKit includes an implementation of Inertial Flow [SS15] for the computation of contraction orders. We perform experiments with both Inertial Flow and InertialFlowCutter. As Inertial Flow is outperformed by InertialFlowCutter, our evaluation focuses on contraction orders obtained by InertialFlowCutter.

Inputs and Methodology. We perform experiments on several graphs with synthetic and real turn cost data. See Table 7.1 for an overview. We use three city-sized instances of the road networks of Chicago [Res], London and Stuttgart. The London and Stuttgart instances were provided by PTV[22] with real turn restrictions and cost data. Our biggest benchmark instance is a graph of the road network of Western Europe made publicly available for the Ninth DIMACS implementation Challenge [DGJ09] with synthetic turn costs. To generate synthetic turn costs, we assign a travel time of 100 s to all U-turns. This number does not model a realistic time but a heavy penalty. All other turns are free. This model has been suggested in [DGPW17] and found to approximate real-world turn cost effects on the routing sufficiently well.

We perform experiments on the biggest strongly connected component of edge-based model representation of each graph and the induced subgraph on the original graph. Preprocessing running times are averages over 10 runs, customization running

[18] `https://github.com/RoutingKit/RoutingKit`

[19] `https://github.com/kit-algo/InertialFlowCutter`

[20] `https://github.com/RoutingKit/RoutingKit/pull/77`

[21] `https://github.com/kit-algo/InertialFlowCutter/pull/6`

[22] `https://ptvgroup.com`

Table 7.2: Performance results for different contraction orders on each graph. We report the number of edges in the augmented graph and running times for preprocessing, customization, and queries. *Orig.* denotes the baseline on the nonturn/compact graph. The other three orders are for the edge-based model. *Deri.* indicates the derived order, *Edge* the order computed on the expanded graph, *Cut* the order obtained by ordering edges in the original graph.

		CCH Edges	Prepro.	Custom.	Query
		$[\cdot 10^3]$	[s]	[ms]	[μs]
Chicago	Orig.	118	0.2	6	18
	Deri.	1 439	0.2	155	150
	Edge	819	1.1	50	60
	Cut	852	0.2	51	57
London	Orig.	182	0.3	7	20
	Deri.	1 199	0.3	85	111
	Edge	767	1.1	37	52
	Cut	840	0.3	40	51
Stuttgart	Orig.	362	0.5	11	16
	Deri.	2 145	0.6	94	79
	Edge	1 607	2.4	58	41
	Cut	1 680	0.9	60	37
Europe	Orig.	53 521	182.3	2 349	187
	Deri.	414 615	202.1	29 787	1 561
	Edge	311 213	2 321.1	14 787	524
	Cut	331 794	256.3	14 751	577

times averages over 100 runs. We utilize parallelization only for the preprocessing. All other phases are run sequentially. For the queries, we perform 1 000 000 point-to-point queries where both source and target are edges drawn uniformly at random. In the edge-based model, these edges correspond to vertices, which we select as source and target. For the original and compact graph, we use the heads of these edges.

Edge-Based Model. We evaluate the impact of different contraction orders on the performance of the different phases and the size of the augmented graph. Preprocessing includes both computing the order and the contraction but is dominated by the ordering. Table 7.2 depicts the results. Incorporating turns has a significant impact on the running time of all phases of CCH. The number of edges in the hierarchy grows at least by a factor of four to up to more than an order of magnitude. The

Table 7.3: Performance impact of different optimizations on each graph. We report the number of triangles enumerated during the customization as well as customization and query running times. All configurations use a cut-based contraction order. Directed hierarchies imply the removal of infinite shortcuts and reordering separator vertices builds on both directed hierarchies and the removal of infinite shortcuts.

		Triangles	Custom.	Query
		[$\cdot 10^6$]	[ms]	[μs]
Chicago	None	21.6	51	57
	Infinity	19.6	48	56
	Directed	13.3	28	41
	Reorder	8.2	20	31
London	None	12.9	40	51
	Infinity	11.0	36	51
	Directed	7.7	23	40
	Reorder	4.8	18	30
Stuttgart	None	11.4	60	37
	Infinity	8.5	53	37
	Directed	6.2	36	30
	Reorder	4.4	32	22
Europe	None	3 955.7	14 751	577
	Infinity	3 413.6	13 942	582
	Directed	2 319.7	9 590	407
	Reorder	1 514.2	8 180	306

derived order performs the worst on all instances. On Chicago, the customization slows down by a factor of 25. On the other instances, the slowdown is about an order of magnitude. The slowdown for queries is not as strong but still significant (by a factor of 5 to 8). Only the preprocessing stays comparatively fast as it is dominated by the order computation, which can run on the unmodified original graph. We conclude that this approach is not feasible.

With the edge-based order, we achieve a better order at the cost of additional preprocessing time. The slowdown compared to a nonturn CCH is reduced to a factor of five for the customization phase, for queries to 2.5 to 3. However, preprocessing takes up to an order of magnitude longer. Orders computed by Inertial Flow are generally worse than InertialFlowCutter orders (the customization is a factor 1.3 to 1.5 slower) and on graphs of the edge-based model this difference becomes even more pronounced (factor 1.3 to 2.8). Consequentially, we focus on InertialFlowCutter orders.

Cut orders achieve the best trade-off between the running times of the different phases. Customization and query performance is roughly the same as with an edge-based order. The preprocessing slowdown is well below a factor of two for all graphs. InertialFlowCutter has certain optimizations which find optimal vertex orders for certain subclasses of graphs. We did not implement these optimizations for cut-based orders. We expect that implementing them would close the gap in quality between edge-based and cut-based orders.

In Table 7.3, we report performance results depending on the additional optimizations applied. All configurations use cut-based orders. We also report the number of triangles enumerated during the customization as the triangle enumeration dominates the customization running time. The impact of the optimizations is similar across all instances. All optimizations combined roughly achieve a speedup of two on both customization and queries. Removing undirected infinite shortcuts alone yields only small improvements. Combining this with directed hierarchies and removing all directed infinite shortcuts has a much bigger impact. This impact can be further amplified by reordering separator vertices, which produces even more infinite shortcuts. Note that the work per triangle is different for directed hierarchies. For undirected hierarchies, each triangle will be enumerated once and both directed triangles will be relaxed at once. For directed hierarchies, however, both directions will be enumerated separately. Thus, for undirected hierarchies, the number of relaxation operations is twice the number of enumerated triangles and the reduction achieved by directed hierarchies even greater. It is noteworthy that even though our optimizations primarily aim for the customization running time, we also achieve a significant speedup for query running times. The removal of infinite edges also reduces the number of edges in the query search space.

Compact Model. We also evaluate the performance of CCH with the compact model. The implementation is considerable more complex than our optimizations for the edge-based model and sadly does not deliver competitive performance. As we cannot use the efficient quotient graph based contraction routine, preprocessing slows down by an order of magnitude as previously observed in [DSW16]. For the Europe instance, the augmented graphs in the compact model and in the edge-based model contain a similar number of edges. The number of triangles, however, increases by a factor of 43. This leads to a slowdown of the customization by a factor of 34. Queries are even worse. The running time increases by a factor of 53. The reason for this slowdown are vertices with high degrees (several thousand edges) in high-level separators. This happens because we get shortcuts between almost all pairs of entry and exit nodes of separator vertices. When an entry node is popped from the queue, all outgoing edges of that vertex are relaxed. This leads to a tremendous

Table 7.4: Performance of Dijkstra, CH, CRP and CCH in the compact model, in the edge-based model as is and with our optimizations (Edge-based*) on Europe with and without turns. Preprocessing was executed in parallel, customization and query sequentially. For CH and CRP we list unscaled results as reported in [DGPW17].

	No turns			Turns			
	Prepro. [s]	Custom. [s]	Queries [ms]	Repr.	Prepro. [s]	Custom. [s]	Queries [ms]
Dij	-	-	1 061.52	Edge-based	-	-	2 674.72
				Compact	-	-	12 699.32
CH	109	-	0.11	Edge-based	1 392	-	0.19
				Compact	1 753	-	2.27
CRP	654	10.55	1.65	Compact	654	11.12	1.67
CCH	182	2.35	0.19	Edge-based	2 321	14.79	0.52
				Edge-based*	256	8.18	0.31
				Compact	2 542	281.56	16.51

amount of edge relaxations and the observed slowdown. On Stuttgart and London, the slowdowns are around factor 20.

Comparison with Related Work. Table 7.4 summarizes our results and depicts them in comparison to running times achieved by competing approaches as reported in [DGPW17]. The experiments were performed on the publicly available Europe instance which is the only instance also considered in related work. Our experiments were conducted on a newer machine. Thus, the absolute numbers are not perfectly comparable. Using the comparison methodology from [Bas+16], the numbers from [DGPW17] should be scaled down by a factor of 0.79. We observe that incorporating turns has a strong impact on all algorithms except CRP. Dijkstra becomes at least 2.5 times slower. CH queries remain comparatively fast (at least on the edge-based model), but preprocessing slows down by more than an order of magnitude. The CRP nonturn variant is realized as free turns in the compact model which explains why incorporating turns leaves the performance unaffected. While CCH achieves faster running times than CRP in all phases on nonturn graphs, without our modifications, it is outperformed by CRP on graphs with turns. However, when using cut-based orders and all optimizations, CCH again outperforms CRP. CCH with the compact model is outperformed by the optimized edge-based variant in all phases. Note that both the CRP and CCH customization times can be further decreased through parallelization and by two related techniques known as microcode [DW13] (for CRP) and triangle preprocessing [DSW16] (for CCH). However, both techniques require significantly more space, and we choose not to use them to keep the space requirement low.

7.6 Conclusion

We incorporated turn costs and restrictions into customizable contraction hierarchies. We presented several straightforward yet effective optimizations that bring preprocessing and customization times on the expanded graph close to those achieved on the simplified graph. Preprocessing now takes similar time on the simplified and expanded graph, and customization on the expanded graph is only roughly three times slower (down from up to an order of magnitude, e.g., on Chicago).

Adapting customizable contraction hierarchies to the compact model was much harder. We observed that CCHs and the compact model do not match well. CCHs rely heavily on concepts for undirected graphs, whereas the compact model is inherently directed. Moreover, shortcuts built from more than two edges are an issue for CCH customization, where there is no notion of graph searches. Consequently, our experiments showed that the CCH implementation tailored to expanded graphs significantly outperforms the one for compact graphs.

Note that our study focused on the *basic* CCH customization. In principle, our optimizations for the edge-based model can also be combined with the *perfect* customization and perfect witness searches [DSW16], whose goal is to remove all superfluous edges in the hierarchy. However, when the hierarchy passed to the perfect customization algorithm is not chordal, we can no longer guarantee that the perfect witness searches remove *all* superfluous edges. It would be interesting to study how well perfect witness searches perform in practice when infinite shortcuts are removed during the metric-independent preprocessing.

8 Partitioning Evolving Road Networks

The first preprocessing step in many algorithms for route planning is to partition the road network into roughly balanced cells. While the road network of the world is fairly stable on the macroscopic level, small changes are quite frequent, as can be seen with OpenStreetMap. Since current road network partitioners are highly sensitive to small changes, partitioning consecutive network snapshots from scratch often yields quite different partitions. This is a problem, since real-world applications often require the partition to stay roughly the same over time, and that updates are restricted to local areas where the network has changed. In this chapter, we present an algorithm to partition an evolving road network that updates the partition of a previous snapshot without recomputing the whole partition from scratch. Our thorough experimental evaluation on continental road networks shows that our algorithm significantly increases the similarity of consecutive partitions, with limited impact on the partition quality. As a side effect, our algorithm is an order of magnitude faster than partitioning from scratch when the changes are small.

This chapter is based on joint work with Daniel Delling, Dennis Schieferdecker and Michael Wegner [BDSW20].

8.1 Introduction

Graph partitioning is an important subroutine in many applications such as parallel processing, image processing, VLSI design, and route planning. Although being

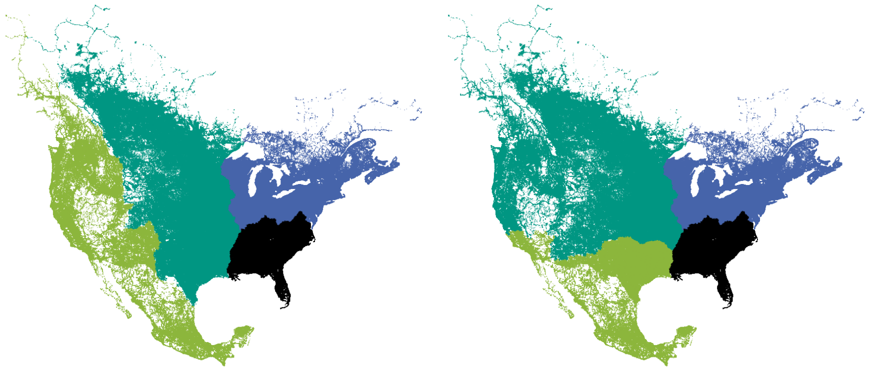


Figure 8.1: Snapshots of the North American road network in OpenStreetMap, one from 2018-08-08 (left) and one from 2018-09-08 (right). We partitioned both snapshots from scratch with the same partitioning algorithm, the same parameters, and the same random seed. Note that, although only 0.36 % of the vertices were added and only 0.21 % were removed between the snapshots, the western half of the North American continent is cut completely differently.

NP-hard in general, there are now many efficient algorithms that perform well on real-world problems (see [Bul+16] for a recent overview). In particular, the road network of the world can be partitioned with high quality on a single multi-core machine in a few hours [DGRW11, SS15, DGPW17].

At first sight it seems that the road network of the world does not change too often. While this is true on the macroscopic level, we observe that small changes are quite frequent. For example, there are several million changes to OpenStreetMap *each day*. Therefore, partition-based map applications need to update and repartition the road network at regular (daily, weekly, monthly) intervals. The most straightforward approach is to partition each network snapshot from scratch. We observe, however, that consecutive partitions are often quite different. See Figure 8.1 for an example.

This is a problem for multiple reasons. Modern map applications support not only point-to-point queries but also many other types of queries, such as finding closest points of interest and computing alternate routes. While the partition has limited impact on the result of point-to-point queries (or no impact when shortest paths are unique), its impact on other types of queries can be huge. We give one example.

The partition-based routing method *customizable route planning* (CRP) [DGPW17] partitions the network into several roughly balanced cells and precomputes *shortcuts* between each pair of boundary vertices in the same cell. The cost of a shortcut corresponds to the shortest path between its endpoints within the cell. Point-to-point queries run a modification of bidirectional search that uses the shortcuts to skip over cells that contain neither the source nor the target. The query stops when the search frontiers meet. As long as the shortest path in the network does not change (and is unique), the result of the query stays the same, even when the partition does change.

A common approach to find alternate routes is to compute a set of *single-via paths* (concatenations of two shortest paths s–v and v–t), which are then ranked [ADGW13]. To compute alternate routes within the CRP framework, we do not stop the query when the search frontiers meet, but advance them a little further. Each (boundary) vertex that has been scanned by both searches is a candidate via vertex. When the partition changes, the candidate via vertices can change, and thus the alternate route returned to the user. Since spurious results may undermine the user's confidence in the entire system, the partition should stay roughly the same over time.

Another drawback of partitioning each snapshot from scratch, besides high sensitivity to small changes, is running time. Recomputing the whole partition from scratch seems like a waste of CPU time, unnecessarily increasing data build times.

Our Contribution. We introduce an algorithm to partition an evolving road network whose goal it is to keep the partition roughly the same over time. Our algorithm updates the partition of a previous snapshot without recomputing the whole partition from scratch. More precisely, we first assign each new vertex to an existing cell and then repair and reoptimize cells that have become infeasible. We experimentally evaluate our algorithms on continental road networks from OpenStreetMap, with snapshots at different intervals. The experiments show that the similarity of consecutive partitions increases significantly, with limited impact on the partition quality (measured by the number of cut edges). Moreover, when the changes are relatively small, we see a speedup of an order of magnitude over partitioning from scratch.

Related Work. Motivated by several important applications, the graph partitioning problem has received considerable attention recently; see e.g. [Bul+16] for an overview. To partition road networks, early work [HKMS09, BD09, Bau+10] used general-purpose partitioners like Scotch [PR96], METIS [KK98], or Party [MS04]. However, one can compute partitions of significantly better quality when using special-purpose partitioners tailored to road networks.

The first such partitioning algorithm is PUNCH [DGRW11]. It introduces and exploits the concept of *natural cuts*, which are natural or man-made obstacles, such as rivers, mountains, and highways. At its heart is the *filtering phase*, which finds natural cuts by local maximum-flow computations and contracts all edges not contained in any natural cut. The *assembly phase* heuristically combines the resulting *fragments* to build a partition. Buffoon [SS12] incorporates the filtering phase of PUNCH into the general-purpose partitioning algorithm KaHIP [SS13].

An alternative approach to the filtering phase of PUNCH is Inertial Flow [SS15], which recursively bisects the network until the fragments are sufficiently small. To bisect the network, it exploits the geometric embedding of the network.

Like Inertial Flow, the FlowCutter algorithm [HS18] recursively bisects the network. For each bisection, it computes a Pareto set of nondominating cuts with respect to the cut size and balance, and picks a cut among those with a good tradeoff between cut size and balance. The recent InertialFlowCutter algorithm [GHUW19] is a variant of FlowCutter that uses geometric information, based on ideas from Inertial Flow. It is about 6 times faster than FlowCutter while preserving the partition quality.

There has also been previous work on the graph *repartitioning* problem, although not in the context of road networks, but scientific computing applications. Various problems in solid and structural mechanics [ZTF14] and fluid dynamics [ZTN14] can be described by partial differential equations (PDEs). Such PDEs can be solved by the finite-difference or finite-element method [Zho93], which discretize the domain of the PDE by a *mesh*. The function value at each discretization point (i.e., vertex in the mesh) is approximately computed from the values at its neighboring vertices. Using an iterative scheme, new approximate values are determined by the values at the neighboring vertices from the previous iteration. Since finite-element meshes can become very large, they are partitioned into well-separated, roughly balanced cells and distributed over multiple processing elements [Bul+16].

During the solution process, the mesh is refined in regions where large errors exist and coarsened in well-behaved regions. To maintain load balance, the mesh is periodically repartitioned. Besides the cut size (which correlates with the *communication volume*), the similarity between the old and new partition (which correlates with the *migration volume*) is an important optimization criterion for this application [Bul+16].

One approach are scratch-remap repartitioners [PSS95, SS94, OB98, SKK01]. These first partition the new mesh using a state-of-the-art partitioner and then compute a migration-minimal mapping between the old and new partition. Since the new partition is produced from scratch, its cut size is small. However, the migration volume is often very high, since this criterion is considered only in the second phase.

Another approach are diffusion-based repartitioners [WCE97, SKK97, SKK01]. These are inspired by the physical process of *diffusion*, i.e., vertices move from a cell of higher to one of lower vertex concentration. While this results in a good migration volume, the number of cut edges is often large.

The unified repartitioning algorithm [SKK00] optimizes both criteria directly by combining the above-mentioned approaches. Following the multilevel graph partitioning approach, it iteratively contracts the new mesh using a variant of the heavy-edge matching algorithm [KK98], which matches two vertices only if they are in the same cell in the old partition. Next, the contracted mesh is partitioned twice using a scratch-remap and diffusion-based algorithm, respectively, and the partition with the better tradeoff between number of cut edges and migration volume is picked. Finally, the mesh is iteratively uncontracted, using an improvement heuristic in each iteration to optimize the mesh partition locally.

A rather simple approach [HLD96, Wal09] is to introduce a zero-weight vertex for each cell in the old partition, which is not allowed to change its cell during repartitioning. This vertex is connected to each other vertex v in its cell by an edge whose weight represents the migration cost for v.

The problem we address is similar to the mesh repartitioning problem in that we optimize both the cut size and similarity. Note, however, that the old and new mesh are nested in the sense that new vertices result from splitting or merging vertices in the old mesh. Hence, there is a natural assignment of all vertices in the new mesh to cells in the old partition [WB95]. In fact, most work [PSS95, SS94, SKK97, OB98, SKK01] considers a mesh with *fixed* topology, where adaptive mesh refinements are handled as vertex weight increases. In contrast, we allow vertices to be freely inserted into and removed from the network. For example, a newly constructed bridge that connects two previously disconnected cells has no natural assignment to any cell. Moreover, in the mesh repartitioning problem, the number of cells is fixed (it is equal to the number of processing elements), while we allow the number of cells to change.

Outline. This chapter is organized as follows. Section 8.2 provides a precise definition of the problem we solve, and briefly reviews current road network partitioners. Section 8.3 describes our approach to partition an evolving road network in detail. Section 8.4 presents an extensive experimental evaluation on continental road networks. Section 8.5 concludes with final remarks.

8.2 Preliminaries

We consider undirected graphs $G = (V, E)$ where each vertex $v \in V$ has a positive size $s(v)$ and each edge $\{u, v\} \in E$ has a positive weight $w(u, v)$. Our focus is on road networks, where vertices represent intersections and edges represent road segments. Partitioning algorithms often use *edge contractions*. To contract an edge $\{u, v\}$, we replace its endpoints by a single vertex w of size $s(w) = s(u) + s(v)$ and relink all edges incident on u or v to the new vertex w. Multiple parallel edges are combined (adding up their weights) and self-loops are removed.

A *partition* of V is a set $P = \{C_1, \ldots, C_k\}$ of cells $C_i \subseteq V$ with the property that each vertex is contained in exactly one cell, i.e., $C_i \cap C_j = \emptyset$ for $i \neq j$ and $V = C_1 \cup \cdots \cup C_k$.

A multilevel partition of V is a sequence $\mathcal{P} = \langle P^1, \dots, P^L \rangle$ of partitions P^l, where l denotes the *level* of the partition. For ease of notation, we set $P^0 = \{\{v\} : v \in V\}$ and $P^{L+1} = \{V\}$. Since we use *nested* multilevel partitions, for each cell C_i^l, there is a cell $C_j^{l'}$ with $C_i^l \subseteq C_j^{l'}$ on all levels above; C_i^l is called a *subcell* of each $C_j^{l'}$. We denote by $c^l(v)$ the cell that contains v on level l. A *boundary* or *cut* edge on level l is an edge $\{u, v\}$ with $c^l(u) \neq c^l(v)$. Its endpoints are *boundary vertices* on level l. We denote by B^l the set of boundary vertices on level l.

8.2.1 Problem Statement

We are given two graphs $G = (V, E)$ and $\bar{G} = (\bar{V}, \bar{E})$, where $\bar{V}$ is obtained from V by inserting some vertices V^+ with $V^+ \cap V = \emptyset$ and removing some vertices $V^- \subseteq V$. Analogously, $\bar{E}$ is obtained from E by inserting some edges E^+ with $E^+ \cap E = \emptyset$ and removing some edges $E^- \subseteq E$. Hence, $\bar{V} = (V \setminus V^-) \cup V^+$ and $\bar{E} = (E \setminus E^-) \cup E^+$. We call G the *old graph* and $\bar{G}$ the *new graph*. In addition, we are given an L-level partition $\mathcal{P}$ of G with maximum cell sizes $U^1, \dots, U^L$. The problem we consider is computing an L-level partition $\bar{\mathcal{P}}$ of $\bar{G}$ such that for each level l, $1 \leq l \leq L$, the size of each cell is bounded by U^l, the number of cut edges in the partition is minimized, and the similarity between P^l and $\bar{P}^l$ is maximized.

It remains to formalize our notion of similarity. For real-world route planning systems following the partition-based overlay approach [SWW00, SWZ02, HSW08, DGPW17], it is often desirable to keep the overlay topology fairly stable, as discussed in the introduction in Section 8.1. Therefore, we define the similarity between two partitions P^l and $\bar{P}^l$ as the fraction of boundary vertices that are boundary vertices of both P^l and $\bar{P}^l$, i.e., $S^l = |B^l \cap \bar{B}^l| / |B^l \cup \bar{B}^l|$.

8.2.2 PUNCH

PUNCH [DGRW11] is a partitioning algorithm tailored to road networks. It has been applied [DGPW17, DGNW13] to various partition-based shortest-path techniques, including CRP [DGPW17], Arc Flags [HKMS09, Lau09], and CHASE [Bau+10]. Given a graph G and a maximum cell size U, PUNCH splits the graph into cells of maximum size U while minimizing the cut size. It works in two phases. At its heart is the *filtering phase*, which finds *natural cuts* (natural or man-made obstacles, such as rivers and railway tracks) and contracts all edges not contained in any natural cut. The *assembly phase* heuristically combines the resulting *fragments* to build a partition.

Filtering Phase. The *natural-cut heuristic* is executed in iterations. In each iteration, it picks a center c at random and builds a breath-first search (BFS) [SMDD19] tree T

rooted at c until the total size of the vertices visited during the reaches αU, where α is a parameter in $(0, 1]$. The neighbors $v \notin T$ of the vertices in T form the *ring* of c. Moreover, the vertices visited by the BFS before the total size reached $\alpha U/f$ form the *core* of c, where $f > 1$ is a second parameter. Then, the natural-cut heuristic temporarily contracts the core and the ring into a single source and sink vertex, respectively, determines a maximum flow between the source and the sink, finds a corresponding minimum cut, and marks all edges in this cut. The procedure stops when each vertex has been contained in at least one core.

To increase the number of marked edges, the iterative procedure is repeated $\mathcal{C}$ times. Afterwards, the natural-cut heuristic contracts all unmarked edges. The vertices in the resulting graph are called *fragments*. Note that each fragment represents a subgraph of G that was never cut and that each two adjacent fragments are separated by a natural cut. Typical parameter values are $\alpha = 1$, $f = 10$, and $\mathcal{C} = 2$.

Assembly Phase. PUNCH runs a *greedy algorithm* to compute an initial partition of G from the fragment graph. It repeatedly contracts two adjacent vertices in the fragment graph until no contraction is possible without violating the upper bound U. The two vertices to be contracted next are picked based on a randomized *score function* [DGRW11]. Intuitively, the algorithm prefers small vertices that are tightly connected. The output of the greedy algorithm is a *contracted graph* H, where each vertex represents a cell in the partition of G.

The initial partition is then improved by an iterative *local search*. In each iteration, it picks two adjacent cells R, S at random from H. Let H_{RS} be the subgraph of H induced by R, S, and their neighbors in H and let G'_{RS} be obtained from H_{RS} by unpacking R and S into their constituent fragments (the neighbors remain contracted). Then, the greedy algorithm is run on G'_{RS}, outputting a contracted graph H'_{RS}. If H'_{RS} represents a better partition (i.e, one with a smaller cut size) than H_{RS}, we replace H_{RS} by H'_{RS} in the current solution H. The local search stops when each pair of adjacent cells in H has been considered φ times in succession without improving the current solution.

Since both the greedy algorithm and the local search are randomized, PUNCH uses a *multistart heuristic*, which runs the greedy algorithm followed by the local search multiple times on the fragment graph. After M candidate solutions have been generated, the best solution seen so far is returned. Alternatively, the candidate solutions generated by the multistart heuristic can be combined using an *evolutionary algorithm* [DGRW11]. Typical parameter values are $\varphi = 16$ and $M = 9$.

8.2.3 Inertial Flow

Inertial Flow [SS15] is a partitioner tailored to road networks that exploits their geometric embedding. It has been applied [DSS18, SN20] to the partition-based

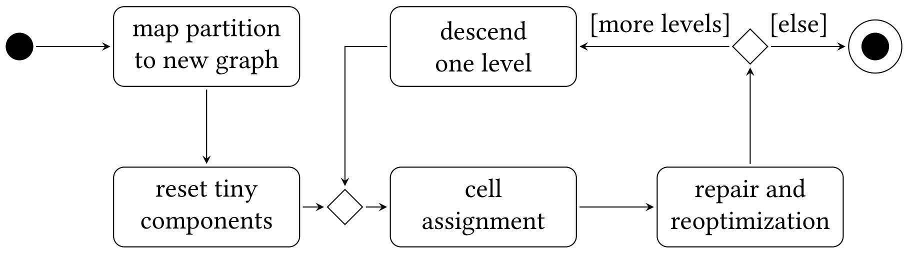


Figure 8.2: Schematic view of our repartitioning approach.

shortest-path algorithms CRP [DGPW17] and CCH [DSW16]. Its core algorithm bisects a graph $G = (V, E)$ with an embedding $\sigma : V \longrightarrow \mathbb{R}^2$ into two balanced parts as follows: (1) Pick a line ℓ with direction $d \in \mathbb{R}^2$. (2) Project each point $\sigma(v)$, $v \in V$, orthogonally onto ℓ. (3) Sort the vertices by their occurrence on ℓ. (4) Determine a maximum flow between the first $\lfloor b|V|\rfloor$ vertices and the last $\lfloor b|V|\rfloor$ vertices. (5) Find a corresponding minimum cut. A typical parameter value is $b = 0.25$.

To find a partition of G with maximum cell size U, the Inertial Flow algorithm recursively bisects G until the resulting parts have a size of at most U. For each bisection, Inertial Flow runs the core algorithm multiple times with parameter d set to $(0, 1)$, $(1, 0)$, $(1, 1)$, and $(-1, 1)$, and picks the smallest cut among those.

Inertial Flow computes partitions of reasonable quality. To improve the quality, Inertial Flow can be used to produce a partition with at most U/f vertices per cell, where $f > 1$ is an additional parameter. Contracting the edges within each cell yields a fragment graph. The fragments can then be combined as in the assembly phase of PUNCH. A typical parameter value is $f = 32$.

8.3 Our Approach

This section discusses our approach to repartition road networks. Instead of partitioning the new graph from scratch, we start from the given partition, incorporate the vertices V^+, and repair and reoptimize the partition (see Figure 8.2). We assume there are stable identifiers associated with the vertices in both graphs that allow us to map vertices in the old and new graph to each other. Both OpenStreetMap and the proprietary data we are aware of provide such identifiers in the form of 64-bit integers. In case there are no stable identifiers available, we can heuristically map the vertices using, for example, their coordinates.

Our approach starts by mapping the partition $\mathcal{P}$ of the old graph G to the new graph $\bar{G}$. More precisely, each vertex $v \in V \cap \bar{V}$ inherits its cell identifiers from $\mathcal{P}$, i.e., we set $\bar{c}^l(v) = c^l(v)$ for all levels l. The vertices V^+ are not assigned to a cell on any level. After a quick preprocessing step (Section 8.3.1), we consider each cell $\bar{C}_i^l$ in the partition in descending level order, starting with the single cell on level $L + 1$, which contains the entire new graph. Each cell $\bar{C}_i^l$, which induces the subgraph $\bar{G}\left[\bar{C}_i^l\right]$ of $\bar{G}$, is processed in two phases. The first phase assigns each vertex $v \in V^+$ to an existing cell on level $l - 1$ (Section 8.3.2). The second phase repairs cells that have become infeasible and reoptimizes the partition (Section 8.3.3).

We handle cells on the same level in parallel if multiple CPU cores are available. Moreover, if there is no change within a cell, we skip its subcells on all levels below.

8.3.1 Detecting Tiny Components

For several reasons (including data errors), there can be distinct components consisting of only a few vertices in the old graph that are connected to their neighborhood in the new graph. For example, consider a newly constructed road. In the old graph, it could still be under construction and not connected to the main network, and therefore a distinct component of its own. A partition could assign this component and its neighborhood to different cells, since there are no cut edges between them. However, connecting the new road to the main network leads to cut edges that are often unsuitably chosen. Hence, before the first cell assignment phase, we reset all cell identifiers for each vertex that belongs to a *tiny component* in the old but not in the new graph. We want such vertices to inherit the cell identifiers of their neighborhood rather than keeping their old identifiers. Formally, a tiny component is a connected component with a size that is below a given threshold. In our experiments, we define tiny components as those that contain at most 25 vertices.

8.3.2 Cell Assignment

Executing the first phase on cell $\bar{C}_i^l$ assigns each vertex $v \in V^+ \cap \bar{C}_i^l$ to a level-$(l-1)$ cell based on the cells of its neighbors. This phase resembles the label propagation algorithm [RAK07] for clustering networks. Each vertex is assigned to the cell to which the majority of its neighbors belong, with ties broken uniformly at random. We perform this process iteratively, where at every step, one vertex updates its cell. Note that the first assignment of a vertex is not necessarily final. Therefore, the process continues until no vertex $v \in V^+ \cap \bar{C}_i^l$ changes its cell anymore. Convergence is guaranteed since we move a vertex v from C_i to C_j only if $N_{C_j}(v)$ is *strictly* greater than $N_{C_i}(v)$, where $N_C(v)$ is the number of neighbors in cell C. Hence, with every move, the sum $\sum_v N_{c(v)}(v)$ increases by $2(N_{C_j}(v) - N_{C_i}(v)) \geq 2$ (note that $N_{c(u)}(u)$

changes not only for $u = v$ but also for each neighbor u of v in $C_i \cup C_j$, causing the factor of 2). As the sum cannot exceed $\sum_v \deg(v)$, the process eventually stops.

To implement this approach efficiently, we keep track of the next vertex to assign with a min-heap, initialized with all new vertices adjacent to at least one old vertex. Every time we assign a vertex to a different cell, we insert its neighbors in V^+ into the min-heap. The priority of a vertex v is given by $key(v) = N_\perp(v) + N_2(v) - N_1(v)$, where $N_i(v)$ is the number of neighbors in the i-th most common neighboring cell (ties broken uniformly at random), and $N_\perp(v)$ is the number of as-yet-unassigned neighbors. The intuition here is that all assignments are both final and unambiguous as long as we only extract vertices v with $key(v) < 0$. This is easy to verify by induction on the number of delete-min operations. When $key(v) = 0$, the assignment is not unambiguous but still final. Therefore, the choice of priorities ensures that we start with as many final assignments as possible, and thus reduces the number of cell corrections and the time to converge. Note that vertices unreachable from any vertex $v \in (V \setminus V^-) \cap \bar{C}_i^l$ remain unassigned after this phase. These vertices will be assigned to cells during the second phase that repairs and reoptimizes the partition.

To process the cell on level $L + 1$, we must run cell assignment on the full input graph. For each cell C on all levels below, cell assignment must be run on $\bar{G}[C]$. For efficiency, we create a temporary copy of $\bar{G}[C]$ and run cell assignment on it. This simplifies cell assignment, allows us to use sequential local IDs, and improves locality.

8.3.3 Repair and Reoptimization

After the cell assignment phase, the partition of $\bar{G}\left[\bar{C}_i^l\right]$ is not necessarily feasible. First, there may be *oversized cells*, i.e., cells containing more than U^{l-1} vertices. Second, vertices unreachable from any vertex $v \in (V \setminus V^-) \cap \bar{C}_i^l$ have not been assigned to a cell yet. In the second phase, we repair both issues and reoptimize the partition.

Let K be a graph whose vertices correspond to the cells in the partition. Note that each as-yet-unassigned vertex in $\bar{G}\left[\bar{C}_i^l\right]$ forms a cell of its own. The size of each vertex in K is the number of vertices in the corresponding cell. There is an edge $\{R, S\}$ in K if there is an edge $\{u, v\}$ in $\bar{G}\left[\bar{C}_i^l\right]$ with $u \in R$ and $v \in S$. Its weight is the total weight of the corresponding edges in $\bar{G}\left[\bar{C}_i^l\right]$.

Let K' be a graph obtained from K by *unpacking* some of the cells (we will discuss cell unpacking in Section 8.3.4). In the following, we compute a partition of K', which can easily be transformed into a partition of $\bar{G}\left[\bar{C}_i^l\right]$. Note that to obtain a feasible partition, we must unpack at least each oversized cell. To increase the similarity between $\mathcal{P}$ and $\bar{\mathcal{P}}$, we can relax our definition of oversized cells, allowing cells to

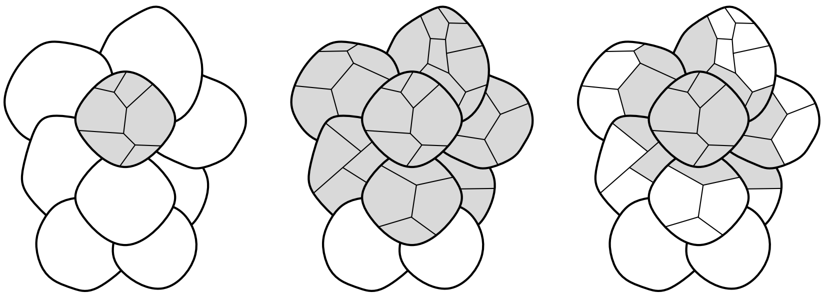


Figure 8.3: Unpacking an oversized cell during the reoptimization phase of our graph repartitioning algorithm. Left: The simple variant unpacks only the oversized cell. Middle: Neighbor unpacking also unpacks all neighboring cells. Right: Partial unpacking unpacks the neighboring cells partially.

contain at most $g^{l-1}U^{l-1}$ vertices, where $g^{l-1} \geq 1$ is the *growth factor* on level $l-1$.

To find an initial feasible partition of K', we run the greedy algorithm from PUNCH on K', yielding a contracted graph H. This partition is then reoptimized by running a variant of the local search from PUNCH on H. That is, we repeatedly pick two adjacent cells R, S at random from H, run the greedy algorithm on the subgraph of H induced by R, S, and their neighbors in H, and update the current solution H accordingly. However, while PUNCH unpacks R and S into their constituent fragments, we unpack them into the corresponding vertices in K'. Since both the greedy algorithm and the local search are randomized, we run them M times on K' with different random seeds and return the partition with the smallest cut size.

We handle cells on a level $l \leq L$ in parallel. On such levels, we run a sequential version of the local search. On level $L+1$ (where there is only a single cell), we parallelize the local search by trying multiple pairs of adjacent vertices concurrently.

8.3.4 Cell Unpacking

We considered three variants of cell unpacking, inspired by PUNCH, which differ in how much they unpack. See Figure 8.3. The simplest variant replaces the single vertex in K representing an oversized level-l cell $\bar{C}_i^l$ with one vertex for each level-$(l-d)$ subcell C_j^{l-d} with $C_j^{l-d} \cap \bar{C}_i^l \neq \emptyset$ (where $d \geq 1$ denotes the *descent step*) and one vertex for each vertex $v \in V^+$ that has been assigned to $\bar{C}_i^l$ during the first phase (cell assignment). The size of these vertices is the number of vertices in the new graph that they represent (possibly one). We call this variant *simple unpacking*.

The second variant, *neighbor unpacking*, gives the second phase more degrees of freedom to reoptimize the partition. Besides unpacking oversized cells, this variant also unpacks all cells that have a common boundary with an oversized cell.

The third variant, which we call *partial unpacking*, unpacks each oversized level-l cell $\bar{C}_i^l$ fully and each level-l cell $\bar{C}_j^l$ that has a common boundary with $\bar{C}_i^l$ partially. More precisely, we replace the single vertex in K representing $\bar{C}_j^l$ with one vertex for each level-$(l-d)$ subcell C_k^{l-d} with $C_k^{l-d} \cap \bar{C}_j^l \neq \emptyset$ that directly borders on $\bar{C}_i^l$, one vertex for each vertex $v \in V^+$ that has been assigned to $\bar{C}_j^l$ during the first phase, and one vertex representing the remaining level-$(l-d)$ subcells of $\bar{C}_j^l$. Again, the size of each vertex is the number of vertices in the new graph represented by the vertex.

8.4 Experiments

8.4.1 Implementation and System

Both the partitioning and repartitioning algorithms are implemented in C++11 and were compiled with GCC 4.8.5 on a system running CentOS 7.7. For parsing the instances we use the RoutingKit library [DSW16]. The machine has 2 NUMA nodes, each equipped with a 10 core/20 threads Intel Xeon CPU E5-2640 v4 clocked at 2.40GHz with 2.5 MiB L2 and 25 MiB L3 cache. It has 192 GiB of DDR4-2400 RAM.

8.4.2 Instances

We evaluate and compare our algorithm with the full partitioning algorithm (*FP*) on road networks extracted from OpenStreetMap. Our *FP* algorithm uses Inertial Flow to find a starting solution and an assembly phase similar to the one from PUNCH to optimize it. OpenStreetMap's contributors edit the map regularly which enables us to test our repartitioning algorithm on snapshots taken at different dates of the same cutout. We test our algorithms on the Australia, North America and Europe instances available from GeoFabrik. Evaluation is performed between snapshots that are one year (1/1/2018 - 1/1/2019) and one month apart (10/1/2019 - 11/1/2019). Shorter time periods (e.g. a week) differ less and thus repartitioning performs at least as good as it does on monthly and yearly instances. More detailed information on the instances can be found in Table 8.1. For the remainder of this section, we reference the graph pairs of the time frames by an M and a Y suffix for the monthly and yearly instances respectively (e.g. $NorthAmerica_Y$ for the North America graph pair on 1/1/2018 and 1/1/2019).

Table 8.1: Instances and their properties. Absolute numbers in millions, relative numbers and vertex churn (VC) in percent. VC is defined as the ratio of $\frac{|V^+ \cup V^-|}{|V|}$.

	1/1/2018		1/1/2019			10/1/2019		11/1/2019		
Input	$\lvert V_{18}\rvert$	$\lvert E_{18}\rvert$	$\frac{\lvert V_{19}\rvert}{\lvert V_{18}\rvert}$	$\frac{\lvert E_{19}\rvert}{\lvert E_{18}\rvert}$	VC_Y	$\lvert V_{10}\rvert$	$\lvert E_{10}\rvert$	$\frac{\lvert V_{11}\rvert}{\lvert V_{10}\rvert}$	$\frac{\lvert E_{11}\rvert}{\lvert E_{10}\rvert}$	VC_M
Aus	1.23	2.85	7.83	6.63	11.33	1.41	3.18	0.45	0.41	0.62
NA	24.90	61.38	1.27	0.98	6.84	26.05	63.94	0.28	0.28	0.99
Eur	30.55	71.46	3.44	3.05	7.12	32.69	76.00	0.54	0.52	1.00

8.4.3 Parameters

Algorithms. We start with a comparison of our repartitioning algorithm variants *SU* (simple unpacking), *NU* (neighbor unpacking) and *PU* (partial unpacking) on the $Australia_Y$ instance with a cell growth of 0% and a descent step $d = 1$ (cf. Section 8.3.3) in Table 8.2. Using a larger d does not result in better quality or similarity based on our experiments. Simple unpacking and partial unpacking perform almost identically, both increasing the overall cut size by 2% compared to the result of the full partitioning algorithm. In our experiments we found that *PU* often has slightly worse similarity on the two lowest levels and identical similarity on the higher levels compared to *NU* while runtimes are comparable. The neighbor unpacking approach considers even more cells than *PU* for distributing unassigned vertices which results in smaller cuts at the cost of a reduced similarity and higher runtimes. Other instances produce similar results. Based on this evaluation, we focus on the simple unpacking approach a descent step $d = 1$ as it produces partitions of good quality and similarity with reasonable runtimes. In the remainder of this section, we use *RP* to denote our repartitioning algorithm using simple unpacking and *FP* to denote the full partitioning (from scratch) algorithm. The level-dependent parameters for both algorithms can be found in Table 8.3. We use the default parameters for our *FP* algorithm, whereas we reduce ϕ_{RP} and M_{RP} on the higher levels. Running more local searches and producing more candidates on these levels reduces similarity since the searches optimize cut size, not similarity and it increases the runtime.

Cell Growth. When new vertices are added to the graph, some cells have to be split in order to hold the size constraint on the cell size. However, most often these additional vertices do not affect the boundary of a cell but are contained in it. So instead of splitting cells, increasing the cut size and decreasing similarity, it is better to allow some cell growth in order to improve similarity. We evaluate the effect of cell growth on $Australia_Y$ and $Australia_M$ in Table 8.4. As expected, the similarity

Table 8.2: Algorithm quality and performance on $Australia_Y$. Best values in bold.

Algorithm	CutSize [%]	S^1 [%]	S^2 [%]	S^3 [%]	S^4 [%]	S^5 [%]	Runtime [s]
SU	2.00	**79.53**	**71.18**	**65.15**	**57.19**	**49.78**	58.57
NU	**0.40**	68.05	63.57	58.25	50.64	22.11	98.06
PU	2.00	**79.53**	**71.18**	**65.15**	**57.19**	**49.78**	**58.35**

Table 8.3: Common partitioning parameters per level of *FP* and *RP*.

Level	U	f	φ_{FP}	φ_{RP}	M_{FP}	M_{RP}
1	25	16	9	9	3	3
2	200	16	9	9	3	3
3	1600	32	16	9	4	3
4	12800	32	16	9	4	3
5	102400	32	32	9	6	3
6	819200	32	32	9	6	3
7	6553600	32	32	9	16	3

increases on all levels with higher cell growth at the cost of a more imbalanced partition - some cells utilizing all the allowed growth. The similarity change is most pronounced on the highest level, on the lowest level the change is less than 2%. There are now oversized cells that exceed the maximum cell size on each level. In the table we report the ratio of the total number of oversized cells over the total number of cells over all levels. The runtime of *RP* decreases with higher cell growth because the local optimizer has less work to do. While the imbalance introduced for the $Australia_M$ instance is always below 2%, it is significantly higher for $Australia_Y$ which is due to the fact that the latter instance has much more churn in both vertices and edges

8.4.4 Comparison with Full Partitioning

Quality and Performance over Time. The more a graph churns over time, the harder it gets to keep the partition stable which is reflected in increased runtimes of our algorithm. Figures 8.4 and 8.5 show the effect of increased churn on the quality and runtime on monthly North America snapshots from 02/01/2018 to 12/01/2018 with 01/01/2018 used as the baseline partition that we want to keep stable. Vertex churn starts at 0.7% for 02/01/2018 and increases strictly monotonously to 6.1% for 12/01/2018. We report the similarity S^1, the total relative amount of oversized cells (over all levels) and the total runtime of *RP* with cell growth parameters 0%, 5% and

Table 8.4: Impact of allowing cell growth when running *RP* on $Australia_Y$ and $Australia_M$.

	Oversized Cells [%]		S^L [%]		Runtime [s]	
Cell Growth [%]	Year	Month	Year	Month	Year	Month
0	0.00	0.00	49.78	77.00	58.57	2.26
1	0.86	0.66	37.24	97.60	63.17	2.05
5	13.25	1.24	61.62	97.60	39.16	1.89
10	19.72	1.70	75.86	94.14	20.39	1.82

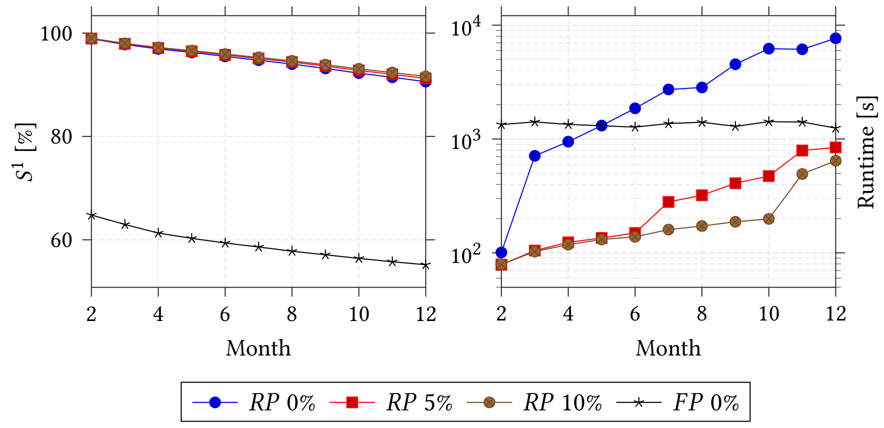


Figure 8.4: Similarity and runtime comparison of *RP* and *FP* for different cell growths on the monthly North America graphs between 02/01/2018 and 12/01/2018 with 01/01/2018 as the baseline partition to be kept stable.

10%. For comparison, we include the *FP* algorithm with a cell growth of 0%. For *FP*, cell growth means increasing the maximum cell size U on each level. Higher cell growths for *FP* do not change the runtime significantly and lead to worse similarity as *FP* optimizes cut size and not similarity, so we exclude them in the figure. Similarity on higher levels follows the same trend as the similarity on level one, just slightly lower as is the case in our other experiments. A cell growth of 10% results in the best similarity values but there is never more than 1% difference between the different cell growth configurations. In contrast, the similarity of *FP* is much worse as *FP* does not optimize this measure. In terms of the partition quality, we compare cut size increases over the partition obtained by *FP* with the same amount of cell growth and notice

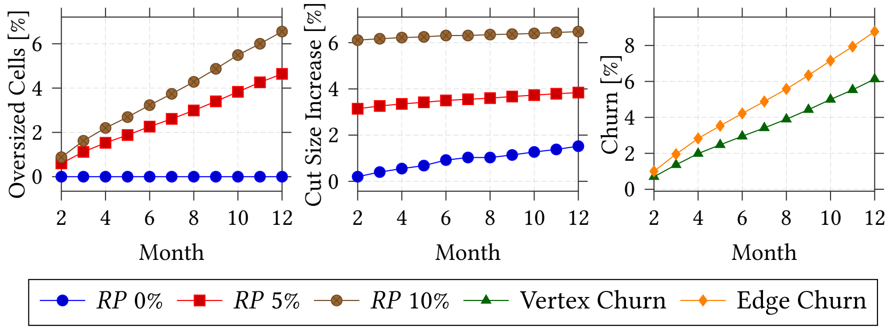


Figure 8.5: Oversized cells, cut size increase of *RP* for difference cell growths and graph churn on the monthly North America graphs between 02/01/2018 and 12/01/2018 with 01/01/2018 as the baseline partition to be kept stable. Cut size is compared to *FP* with the same cell growth.

that the cut size increases by roughly 1%, 3% and 6% for the respective cell growths and does not significantly increase with more churn. This can be explained by the fact that our algorithm mainly optimizes similarity to the input partition whereas allowing *FP* a higher imbalance can lead to different (smaller) cuts. As expected, the ratio of all oversized cells compared to the number of cells increases for a cell growth greater than 0% but stays within reasonable limits for the purpose of road network partitioning. The runtime of *RP* with 0% cell growth is comparable to the higher cell growths for the first month but increases sharply starting with the third month. Allowing cell growths of 5% and 10% yield similar running times up to month 6. Starting with month 7, however, *RP* with cell growth 5% has a significantly higher runtime, while the vertex/edge churn does not have any significant increase during these months. A possible explanation might be the merging of small connected components with bigger ones. In this case, a larger cell growth often allows to assign all vertices of the previously connected component to the now neighboring cell, improving the runtime and retaining the similarity.

Quality and Performance on Monthly Instances. Based on the results of our monthly *RP* evaluation over the course of a year in the previous paragraph, we select a cell growth of 5% for the comparison with *FP* on the monthly instances. For *FP*, we use a cell growth of 0% for comparing similarity for best *FP* results and a cell growth of 5% for a fair comparison of cut sizes. The result of that evaluation can be found in Table 8.5. The similarity to the previous partition is always higher than

Table 8.5: Comparison of *RP* and *FP*, both with 5% cell growth, on the monthly instances.

	CutSize [%]	S^1 [%]		S^L [%]		Runtime [s]	
Instance	*RP*	*RP*	*FP*	*RP*	*FP*	*RP*	*FP*
$Australia_M$	3.09	98.87	64.57	97.60	39.27	2.01	24.27
$NAmerica_M$	3.11	98.64	64.01	87.16	31.07	144.26	1390.57
$Europe_M$	3.24	98.35	62.70	94.70	50.73	197.90	1851.13

Table 8.6: Comparison of *RP* and *FP*, both with 20% cell growth, on the yearly instances.

	CutSize [%]	S^1 [%]		S^L [%]		Runtime [s]	
Instance	*RP*	*RP*	*FP*	*RP*	*FP*	*RP*	*FP*
$Australia_Y$	10.91	85.16	49.60	75.49	23.33	3.50	22.08
$NAmerica_Y$	13.14	91.04	54.55	62.46	32.53	356.81	1459.72
$Europe_Y$	12.83	90.41	52.98	73.64	46.03	251.47	1880.93

98% on the lowest level compared to about 67% for *FP*. *RP* is able to maintain high similarity values on higher levels as well. The cut size is no more than about 3% higher compared to *FP* and the runtime is up to a factor of 12 lower.

Quality and Performance on Yearly Instances. We also include quality and performance figures for the yearly instances where we select a cell growth of 20% to maximize similarity. For *FP*, we chose the same testing methodology as in the monthly comparison in terms of cell growth. While our algorithm is able to maintain good similarity values of 85% or higher on the lowest level, similarity decreases more drastically compared to the monthly instances. The *FP* algorithm only achieves up to 54% similarity on the lowest level and higher levels even worse. In terms of cut size, it shows the limitations of trying to keep a partition stable for a full year with good similarity as our algorithm produces cuts that are overall up to 14% higher compared to *FP*.

8.5 Conclusion

We presented an algorithm to compute multilevel partitions of an evolving road network. Our focus was on keeping the partition roughly the same over time. Compared to partitioning from scratch, our algorithm increases the fraction of unchanged boundary vertices on monthly snapshots from 65 % to 99 % at the finest level, and from 30–50 % to 87–98 % at the coarsest level. With an increase in the cut size by 3 %, the impact on the partition quality is limited. As a side effect, our algorithm is an order of magnitude faster when the changes are small.

Our algorithm works well for moderate changes to areas that already existed in the previous snapshot. However, large areas newly added require special attention. The simplest way is to resort to partitioning from scratch when the graph changes too much. Alternatively, it would be interesting to stick to our repartitioning approach, but preprocess newly added areas as follows. Note that new areas have not yet been searched for natural cuts. Therefore, we first compute a fragment graph for each newly added area using a filtering algorithm such as the natural-cut heuristic, Inertial Flow, or FlowCutter. Then, we replace in the current snapshot the subgraph induced by each new area with its fragment graph. Finally, the repartitioning algorithm is run on the modified snapshot. Moreover, another interesting project is the study of other important classes of evolving networks.

9 Conclusion

Motivated by the issues caused by advancing urbanization (such as traffic jams, accidents, air pollution, and a lack of sufficient parking space), we designed practical algorithms for various problems in the field of urban mobility. Our three main subject areas were the computation of mobility flows, traffic assignment, and dynamic ridesharing. To ensure that our algorithms are easy to implement and have good performance in practice, we employed the algorithm engineering methodology, which complements classic algorithm theory by thorough experimental studies.

We started by studying the efficient computation of mobility flows according to the radiation model. While the radiation model has received significant attention since it was proposed in 2012, a straightforward application to continental networks at the microscopic level is prohibitively expensive. Therefore, we designed a total of four implementations, where each version improves considerably on the previous one with respect to either solution quality or running time. Our CCH-based implementation called CRAD, which gives the best trade-off between solution quality and running time, is orders of magnitude faster than a straightforward implementation of the radiation model. Moreover, a special case of CRAD can be used to find nearest neighbors in road networks. This yielded the first nearest-neighbor algorithm within the CCH framework that supports interactive online queries.

The second main subject area was traffic assignment. While the efficient computation of equilibrium flow patterns has been studied for over 60 years, their use for intelligent real-time traffic management has been prevented by relatively slow running times. To make their computation fast enough for interactive applications, we combined the prototypical link-based traffic assignment algorithm with customizable contraction hierarchies. Moreover, we engineered all aspects of CCHs. Our optimizations accelerate the customization algorithms and both single and batched point-to-point shortest-path queries. They are not restricted to the traffic assignment scenario but significantly speed up CCHs generally.

Ridesharing services such as Uber and Lyft formed the third main subject area. We presented LOUD, a novel framework for dynamic ridesharing based on customizable contraction hierarchies. LOUD is not only 30 times faster than existing algorithms for dynamic ridesharing but also finds a provably optimal insertion for a given ride request. This does not imply that the insertion is optimal from a global point of view, however, finding optimal insertions for the entire set of ride requests is NP-complete. We also integrated LOUD into a widely used transport simulation software package. This underlined the viability and speed of LOUD.

Since CCHs are at the heart of the algorithms from our three main subject areas, and turn restrictions and turn costs are particularly important for inner-city routes, we also studied how to incorporate turn costs and restrictions into CCHs efficiently. We observed a significant performance penalty when running CCHs on edge-based graphs, which encode turn costs and restrictions into the network structure. Therefore, we presented multiple optimizations that exploit the properties of edge-based graphs to accelerate preprocessing and customization on them.

We concluded by designing an algorithm to partition an evolving road network, which is a subroutine in many shortest-path techniques. We observed that network data changes surprisingly frequently in practice and that partitioning consecutive network snapshots from scratch often yields quite different partitions. Since this can be a problem for real-world applications, we presented an approach that updates the partition of a previous snapshot without recomputing the whole partition from scratch. Our experiments showed that this significantly increases the similarity of consecutive partitions, while decreasing the running time by an order of magnitude.

Future research directions in each of our subject areas were already discussed in the corresponding chapters. In summary, it would probably be most interesting to extend the frameworks for traffic assignment and dynamic ridesharing to more complex scenarios. As for traffic assignment, it would be interesting to study the efficient computation of time-dependent traffic flow profiles. With respect to dynamic ridesharing, an interesting next project would be to allow requests already matched to a vehicle to be reordered or moved to a different vehicle. Finally, it would be interesting to integrate demand-responsive ridesharing with schedule-based public transit, or even with a full multimodal scenario.

Bibliography

[Abr+12] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. **HLDB: Location-Based Services in Databases**. In *Proceedings of the 20th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'12)*. Ed. by Isabel F. Cruz, Craig A. Knoblock, Peer Kröger, Egemen Tanin, and Peter Widmayer, pages 339–348. ACM Press, 2012. DOI: 10.1145/2424321.2424365.
Cited on page 95.

[AC17] Tenindra Abeywickrama and Muhammad Aamir Cheema. **Efficient Landmark-Based Candidate Generation for kNN Queries on Road Networks**. In *Proceedings of the 22nd International Conference on Database Systems for Advanced Applications (DASFAA'17)*. Ed. by K. Selçuk Candan, Lei Chen, Torben Bach Pedersen, Lijun Chang, and Wen Hua. Volume 10178 of Lecture Notes in Computer Science, pages 425–440. Springer, 2017. DOI: 10.1007/978-3-319-55699-4_26.
Cited on page 42.

[ACT16] Tenindra Abeywickrama, Muhammad Aamir Cheema, and David Taniar. **K-Nearest Neighbors on Road Networks: A Journey in Experimentation and In-Memory Implementation**. In *Proceedings of the VLDB Endowment* volume 9:6, pages 492–503, 2016. DOI: 10.14778/2904121.2904125.
Cited on pages 40, 42, 47.

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Ed. by Panos M. Pardalos and Steffen Rebennack. Volume 6630 of Lecture Notes in Computer Science, pages 230–241. Springer, 2011. DOI: `10.1007/978-3-642-20662-7_20`.
Cited on pages 16, 17, 62, 94, 126.

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **Alternative Routes in Road Networks**. In *ACM Journal of Experimental Algorithmics* volume 18:1, pages 1.3:1–1.3:17, 2013. DOI: `10.1145/2444016.2444019`.
Cited on page 145.

[AIKK14] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. **Fast Shortest-Path Distance Queries on Road Networks by Pruned Highway Labeling**. In *Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX'14)*, pages 147–154. SIAM, 2014. DOI: `10.1137/1.9781611973198.14`.
Cited on page 42.

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. **Transit Node Routing Reconsidered**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 55–66. Springer, 2013. DOI: `10.1007/978-3-642-38527-8_7`.
Cited on pages 16, 17, 62, 126.

[Bar+18] Hugo Barbosa, Marc Barthelemy, Gourab Ghoshal, Charlotte R. James, Maxime Lenormand, Thomas Louail, Ronaldo Menezes, José J. Ramasco, Filippo Simini, and Marcello Tomasini. **Human Mobility: Models and Applications**. In *Physics Reports* volume 734, pages 1–74, 2018. DOI: `10.1016/j.physrep.2018.01.001`.
Cited on page 12.

[Bar10] Hillel Bar-Gera. **Traffic Assignment by Paired Alternative Segments**. In *Transportation Research Part B: Methodological* volume 44:8–9, pages 1022–1046, 2010. DOI: `10.1016/j.trb.2009.11.004`.
Cited on page 61.

[Bas+10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. **Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns**. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Ed. by Mark de Berg and Ulrich Meyer. Volume 6346 of Lecture Notes in Computer Science, pages 290–301. Springer, 2010. DOI: 10.1007/978-3-642-15775-2_25.
Cited on page 95.

[Bas+16] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. **Route Planning in Transportation Networks**. In *Algorithm Engineering: Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Volume 9220. Lecture Notes in Computer Science. Springer, 2016, pages 19–80. DOI: 10.1007/978-3-319-49487-6_2.
Cited on pages 13, 16, 17, 39, 40, 62, 67, 79, 80, 125, 126, 141.

[Bau+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. **Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm**. In *ACM Journal of Experimental Algorithmics* volume 15, pages 2.3:1–2.3:31, 2010. DOI: 10.1145/1671970.1671976.
Cited on pages 145, 148.

[BCRW16] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner. **Search-space size in contraction hierarchies**. In *Theoretical Computer Science* volume 645, pages 112–127, 2016. DOI: 10.1016/j.tcs.2016.07.003.
Cited on pages 9, 10.

[BD09] Reinhard Bauer and Daniel Delling. **SHARC: Fast and Robust Unidirectional Routing**. In *ACM Journal of Experimental Algorithmics* volume 14, pages 2.4:1–2.4:29, 2009. DOI: 10.1145/1498698.1537599.
Cited on pages 76, 145.

[BDPW13] 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 (SIGSPATIAL'13)*. Ed. by Craig A. Knoblock, Peer Kröger, John Krumm, Markus Schneider, and Peter Widmayer, pages 54–63. ACM Press, 2013. DOI: 10.1145/2525314.2525361.
Cited on page 40.

[BDSW20] Valentin Buchhold, Daniel Delling, Dennis Schieferdecker, and Michael Wegner. **Fast and Stable Repartitioning of Road Networks**. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. Ed. by Simone Faro and Domenico Cantone. Volume 160 of Leibniz International Proceedings in Informatics (LIPIcs), pages 26:1–26:15. Schloss Dagstuhl, 2020. DOI: 10.4230/LIPIcs.SEA.2020.26.
Cited on page 143.

[Ben75] Jon Louis Bentley. **Multidimensional Binary Search Trees Used for Associative Searching**. In *Communications of the ACM* volume 18:9, pages 509–517, 1975. DOI: 10.1145/361002.361007.
Cited on pages 24, 42, 43, 48.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. **Fast Routing in Road Networks with Transit Nodes**. In *Science* volume 316:5824, page 566, 2007. DOI: 10.1126/science.1137521.
Cited on pages 16, 17, 62, 126.

[BM16] Joschka Bischoff and Michal Maciejewski. **Simulation of City-Wide Replacement of Private Cars with Autonomous Taxis in Berlin**. In *Proceedings of the 7th International Conference on Ambient Systems, Networks and Technologies (ANT'16)*. Ed. by Ansar-Ul-Haque Yasar and Jesús Fraile-Ardanuy. Volume 83 of Procedia Computer Science, pages 237–244. Elsevier, 2016. DOI: 10.1016/j.procs.2016.04.121.
Cited on page 13.

[BMN17] Joschka Bischoff, Michal Maciejewski, and Kai Nagel. **City-wide Shared Taxis: A Simulation Study in Berlin**. In *20th IEEE International Conference on Intelligent Transportation Systems (ITSC'17)*, pages 275–280. IEEE Computer Society, 2017. DOI: 10.1109/ITSC.2017.8317926.
Cited on pages 92, 94, 96, 111.

[BMW56] Martin Beckmann, C. Bart McGuire, and Christopher B. Winsten. **Studies in the Economics of Transportation**. Yale University Press, 1956.
Cited on pages 60, 63.

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. **Simple and Space-Efficient Minimal Perfect Hash Functions**. In *Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS'07)*. Ed. by Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh. Volume 4619 of Lecture Notes in Computer Science, pages 139–150. Springer, 2007. DOI: `10.1007/978-3-540-73951-7_13`.
Cited on page 72.

[BRB04] David E. Boyce, Biljana Ralevic-Dekic, and Hillel Bar-Gera. **Convergence of Traffic Assignments: How Much is Enough?** In *Journal of Transportation Engineering* volume 130:1, pages 49–55, 2004. DOI: `10.1061/(ASCE)0733-947X(2004)130:1(49)`.
Cited on pages 66, 85.

[BSW18] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. **Real-Time Traffic Assignment Using Fast Queries in Customizable Contraction Hierarchies**. In *Proceedings of the 17th International Symposium on Experimental Algorithms (SEA'18)*. Ed. by Gianlorenzo D'Angelo. Volume 103 of Leibniz International Proceedings in Informatics (LIPIcs), pages 27:1–27:15. Schloss Dagstuhl, 2018. DOI: `10.4230/LIPIcs.SEA.2018.27`.
Cited on page 59.

[BSW19a] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. **Efficient Calculation of Microscopic Travel Demand Data with Low Calibration Effort**. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'19)*. Ed. by Farnoush Banaei-Kashani, Goce Trajcevski, Ralf Hartmut Güting, Lars Kulik, and Shawn D. Newsam, pages 379–388. ACM Press, 2019. DOI: `10.1145/3347146.3359361`.
Cited on page 11.

[BSW19b] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. **Real-time Traffic Assignment Using Engineered Customizable Contraction Hierarchies**. In *ACM Journal of Experimental Algorithmics* volume 24:2, pages 2.4:1–2.4:28, 2019. DOI: `10.1145/3362693`.
Cited on page 59.

[BSW21] Valentin Buchhold, Peter Sanders, and Dorothea Wagner. **Fast, Exact and Scalable Dynamic Ridesharing**. In *Proceedings of the 23rd SIAM Symposium on Algorithm Engineering and Experiments (ALENEX'21)*. Ed. by Martin Farach-Colton and Sabine Storandt, pages 98–112. SIAM, 2021. DOI: `10.1137/1.9781611976472.8`.
Cited on page 91.

[Bul+16] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. **Recent Advances in Graph Partitioning**. In *Algorithm Engineering: Selected Results and Surveys*. Ed. by Lasse Kliemann and Peter Sanders. Volume 9220. Lecture Notes in Computer Science. Springer, 2016, pages 117–158. DOI: `10.1007/978-3-319-49487-6_4`.
Cited on pages 144, 145, 146.

[Bur64] Bureau of Public Roads. **Traffic Assignment Manual**. U.S. Department of Commerce, 1964.
Cited on page 66.

[BV08] Frédéric Babonneau and Jean-Philippe Vial. **Test instances for the traffic assignment problem**. In tech. rep., 2008.
Cited on page 79.

[BW21] Valentin Buchhold and Dorothea Wagner. **Nearest-Neighbor Queries in Customizable Contraction Hierarchies and Applications**. In *Proceedings of the 19th International Symposium on Experimental Algorithms (SEA'21)*. Ed. by David Coudert and Emanuele Natale. Volume 190 of Leibniz International Proceedings in Informatics (LIPIcs), pages 18:1–18:18. Schloss Dagstuhl, 2021. DOI: `10.4230/LIPIcs.SEA.2021.18`.
Cited on page 39.

[BWZZ20] Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. **Customizable Contraction Hierarchies with Turn Costs**. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*. Ed. by Dennis Huisman and Christos D. Zaroliagis. Volume 85 of OpenAccess Series in Informatics (OASIcs), pages 9:1–9:15. Schloss Dagstuhl, 2020. DOI: `10.4230/OASIcs.ATMOS.2020.9`.
Cited on page 125.

[Cal61] Tom Caldwell. **On Finding Minimum Routes in a Network with Turn Penalties**. In *Communications of the ACM* volume 4:2, pages 107–108, 1961. DOI: `10.1145/366105.366184`.
Cited on pages 126, 127.

[CL07] Jean-François Cordeau and Gilbert Laporte. **The Dial-a-Ride Problem: Models and Algorithms**. In *Annals of Operations Research* volume 153:1, pages 29–46, 2007. DOI: 10.1007/s10479-007-0170-8.
Cited on page 93.

[CMFB16] Giulia Carra, Ismir Mulalic, Mogens Fosgerau, and Marc Barthelemy. **Modelling the Relation between Income and Commuting Distance**. In *Journal of the Royal Society Interface* volume 13:119, pages 1–8, 2016. DOI: 10.1098/rsif.2016.0306.
Cited on page 14.

[Col+07] Vittoria Colizza, Alain Barrat, Marc Barthelemy, A.-J. Valleron, and Alessandro Vespignani. **Modeling the Worldwide Spread of Pandemic Influenza: Baseline Case and Containment Interventions**. In *PLOS Medicine* volume 4:1, pages 95–110, 2007. DOI: 10.1371/journal.pmed.0040013.
Cited on page 12.

[Daf68] Stella Dafermos. **Traffic Assignment and Resource Allocation in Transportation Networks**. PhD thesis. Johns Hopkins University, 1968.
Cited on page 61.

[Dav66] K. B. Davidson. **A Flow Travel Time Relationship for Use in Transportation Planning**. In *Proceedings of the 3rd Australian Road Research Board Conference (ARRB'66)*, 1966.
Cited on page 66.

[DDPW15] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. **Public Transit Labeling**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Ed. by Evripidis Bampis. Volume 9125 of Lecture Notes in Computer Science, pages 273–285. Springer, 2015. DOI: 10.1007/978-3-319-20086-6_21.
Cited on page 95.

[Dev86] Luc Devroye. **Non-Uniform Random Variate Generation**. Springer, 1986. ISBN: 978-1-4613-8645-2. DOI: 10.1007/978-1-4613-8643-8.
Cited on page 25.

[DGJ09] 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. ISBN: 978-0-8218-4383-3.
Cited on pages 35, 51, 80, 137.

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. **PHAST: Hardware-accelerated Shortest Path Trees**. In *Journal of Parallel and Distributed Computing* volume 73:7, pages 940–952, 2013. DOI: 10.1016/j.jpdc.2012.02.007.
Cited on pages 40, 63, 69, 72, 76, 82, 84, 148.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Customizable Route Planning**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Ed. by Panos M. Pardalos and Steffen Rebennack. Volume 6630 of Lecture Notes in Computer Science, pages 376–387. Springer, 2011. DOI: 10.1007/978-3-642-20662-7_32.
Cited on page 126.

[DGPW17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Customizable Route Planning in Road Networks**. In *Transportation Science* volume 51:2, pages 566–591, 2017. DOI: 10.1287/trsc.2014.0579.
Cited on pages 16, 40, 41, 62, 67, 76, 82, 84, 109, 126, 128, 137, 141, 144, 148, 150.

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya P. Razenshteyn, and Renato F. Werneck. **Graph Partitioning with Natural Cuts**. In *25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011. DOI: 10.1109/IPDPS.2011.108.
Cited on pages 144, 145, 148, 149.

[DGW11] 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)*. Ed. by Alberto Caprara and Spyros C. Kontogiannis. Volume 20 of OpenAccess Series in Informatics (OASIcs), pages 52–63. Schloss Dagstuhl, 2011. DOI: 10.4230/OASIcs.ATMOS.2011.52.
Cited on pages 40, 47, 52, 53, 63, 69, 76.

[Dia06] Robert B. Dial. **A Path-Based User-Equilibrium Traffic Assignment Algorithm that Obviates Path Storage and Enumeration**. In *Transportation Research Part B: Methodological* volume 40:10, pages 917–936, 2006. DOI: 10.1016/j.trb.2006.02.008.
Cited on pages 61, 66.

[Dij59] Edsger W. Dijkstra. **A Note on Two Problems in Connexion with Graphs**. In *Numerische Mathematik* volume 1, pages 269–271, 1959.
Cited on pages 7, 17, 39, 62, 94, 125.

[DL13] Florian Drews and Dennis Luxen. **Multi-Hop Ride Sharing**. In *Proceedings of the 6th Annual Symposium on Combinatorial Search (SoCS'13)*. Ed. by Malte Helmert and Gabriele Röger, pages 71–79. AAAI Press, 2013.
Cited on pages 12, 95.

[DSS18] Daniel Delling, Dennis Schieferdecker, and Christian Sommer. **Traffic-Aware Routing in Road Networks**. In *34th IEEE International Conference on Data Engineering (ICDE'18)*, pages 1543–1548. IEEE Computer Society, 2018. DOI: `10.1109/ICDE.2018.00172`.
Cited on page 149.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. **Customizable Contraction Hierarchies**. In *ACM Journal of Experimental Algorithmics* volume 21:1, pages 1.5:1–1.5:49, 2016. DOI: `10.1145/2886843`.
Cited on pages 9, 16, 40, 51, 62, 67, 69, 71, 72, 73, 74, 80, 81, 82, 83, 84, 87, 92, 110, 111, 114, 126, 136, 140, 142, 150, 154.

[DW13] Daniel Delling and Renato F. Werneck. **Faster Customization of Road Networks**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 30–42. Springer, 2013. DOI: `10.1007/978-3-642-38527-8_5`.
Cited on pages 83, 142.

[DW15] Daniel Delling and Renato F. Werneck. **Customizable Point-of-Interest Queries in Road Networks**. In *IEEE Transactions on Knowledge and Data Engineering* volume 27:3, pages 686–698, 2015. DOI: `10.1109/TKDE.2014.2345386`.
Cited on pages 40, 41, 42, 47, 52, 56, 63.

[Eis+11] Jochen Eisner, Stefan Funke, Andre Herbst, Andreas Spillner, and Sabine Storandt. **Algorithms for Matching and Predicting Trajectories**. In *Proceedings of the 13th Workshop on Algorithm Engineering and Experiments (ALENEX'11)*. Ed. by Matthias Müller-Hannemann and Renato F. Werneck, pages 84–95. SIAM, 2011. DOI: `10.1137/1.9781611972917.9`.
Cited on page 53.

[EP13] Alexandros Efentakis and Dieter Pfoser. **Optimizing Landmark-Based Routing and Preprocessing**. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'13)*. Ed. by Monika Sester, Clément Mallet, and John Krumm, pages 25–30. ACM Press, 2013. DOI: `10.1145/2533828.2533838`.
Cited on pages 62, 67.

[EP14] Alexandros Efentakis and Dieter Pfoser. **GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem**. In *Proceedings of the 22th Annual European Symposium on Algorithms (ESA'14)*. Ed. by Andreas S. Schulz and Dorothea Wagner. Volume 8737 of Lecture Notes in Computer Science, pages 358–370. Springer, 2014. DOI: `10.1007/978-3-662-44777-2_30`.
Cited on pages 40, 63.

[EPV15] Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. **SALT. A Unified Framework for All Shortest-Path Query Variants on Road Networks**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Ed. by Evripidis Bampis. Volume 9125 of Lecture Notes in Computer Science, pages 298–311. Springer, 2015. DOI: `10.1007/978-3-319-20086-6_23`.
Cited on pages 40, 63.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael A. Finkel. **An Algorithm for Finding Best Matches in Logarithmic Expected Time**. In *ACM Transactions on Mathematical Software* volume 3:3, pages 209–226, 1977. DOI: `10.1145/355744.355745`.
Cited on pages 24, 42, 43, 48.

[FCF09] Michael Florian, Isabelle Constantin, and Dan Florian. **A New Look at Projected Gradient Method for Equilibrium Assignment**. In *Transportation Research Record* volume 2090:1, pages 10–16, 2009. DOI: `10.3141/2090-02`.
Cited on pages 61, 66.

[FH95] Michael Florian and Donald W. Hearn. **Network Equilibrium Models and Algorithms**. In *Network Routing*. Ed. by Michael O. Ball, Tom L. Magnanti, Clyde L. Monma, and George L. Nemhauser. Volume 8. Handbooks in Operations Research and Management Science. Elsevier, 1995, pages 485–550. DOI: `10.1016/S0927-0507(05)80110-0`.
Cited on pages 60, 61, 66.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. In *Journal of the ACM* volume 34:3, pages 596–615, 1987. DOI: 10.1145/28869.28874.
Cited on page 19.

[FW56] Marguerite Frank and Philip Wolfe. An Algorithm for Quadratic Programming. In *Naval Research Logistics Quarterly* volume 3:1-2, pages 95–110, 1956. DOI: 10.1002/nav.3800030109.
Cited on page 60.

[Gei+10] Robert Geisberger, Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Volker. Fast Detour Computation for Ride Sharing. In *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*. Ed. by Thomas Erlebach and Marco E. Lübbecke. Volume 14 of OpenAccess Series in Informatics (OASIcs), pages 88–99. Schloss Dagstuhl, 2010. DOI: 10.4230/OASIcs.ATMOS.2010.88.
Cited on pages 12, 93, 94.

[Gei11] Robert Geisberger. Advanced Route Planning in Transportation Networks. PhD thesis. Karlsruhe Institute of Technology, 2011. DOI: 10.5445/IR/1000021997.
Cited on pages 40, 41, 51.

[Gen14] Guido Gentile. Local User Cost Equilibrium: A Bush-Based Algorithm for Traffic Assignment. In *Transportmetrica A: Transport Science* volume 10:1, pages 15–54, 2014. DOI: 10.1080/18128602.2012.691911.
Cited on pages 61, 66.

[Geo73] Alan George. Nested Dissection of a Regular Finite Element Mesh. In *SIAM Journal on Numerical Analysis* volume 10:2, pages 345–363, 1973. DOI: 10.1137/0710032.
Cited on page 9.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.
Cited on pages 16, 126.

[GHUW19] Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. **Faster and Better Nested Dissection Orders for Customizable Contraction Hierarchies**. In *Algorithms* volume 12:9, pages 1–20, 2019. DOI: `10.3390/a12090196`.
Cited on pages 9, 137, 146.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. **Exact Routing in Large Road Networks Using Contraction Hierarchies**. In *Transportation Science* volume 46:3, pages 388–404, 2012. DOI: `10.1287/trsc.1110.0401`.
Cited on pages 8, 16, 17, 40, 41, 51, 62, 69, 72, 77, 84, 92, 94, 103, 110, 126.

[Gut04] Ron Gutman. **Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks**. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*. SIAM, 2004.
Cited on page 126.

[Gut84] Antonin Guttman. **R-Trees: A Dynamic Index Structure for Spatial Searching**. In *ACM SIGMOD Record* volume 14:2, pages 47–57, 1984. DOI: `10.1145/602259.602266`.
Cited on page 42.

[GV11] Robert Geisberger and Christian Vetter. **Efficient Routing in Road Networks with Turn Costs**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Ed. by Panos M. Pardalos and Steffen Rebennack. Volume 6630 of Lecture Notes in Computer Science, pages 100–111. Springer, 2011. DOI: `10.1007/978-3-642-20662-7_9`.
Cited on pages 126, 128, 129, 130.

[HBJW14] Yan Huang, Favyen Bastani, Ruoming Jin, and Xiaoyang Sean Wang. **Large Scale Real-Time Ridesharing with Service Guarantee on Road Networks**. In *Proceedings of the VLDB Endowment* volume 7:14, pages 2017–2028, 2014. DOI: `10.14778/2733085.2733106`.
Cited on pages 94, 96, 117.

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. **Fast Point-to-Point Shortest Path Computations with Arc-Flags**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Ed. by Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. Volume 74. DIMACS Book. American Mathematical Society, 2009, pages 41–72.
Cited on pages 76, 77, 126, 145, 148.

[HLD96] Bruce Hendrickson, Robert W. Leland, and Rafael Van Driessche. **Enhancing Data Locality by Using Terminal Propagation**. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences (HICSS'96)*, pages 565–574. IEEE Computer Society, 1996. DOI: `10.1109/HICSS.1996.495507`.
Cited on page 147.

[HNA16] Andreas Horni, Kai Nagel, and Kay W. Axhausen (editors). **The Multi-Agent Transport Simulation MATSim**. Ubiquity Press, 2016. ISBN: 978-1-909188-75-4. DOI: `10.5334/baw`.
Cited on pages 17, 92, 94, 111, 112.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**. In *IEEE Transactions on Systems Science and Cybernetics* volume 4:2, pages 100–107, 1968. DOI: `10.1109/TSSC.1968.300136`.
Cited on page 62.

[Ho+18] Sin C. Ho, Wai Yuen Szeto, Yong-Hong Kuo, Janny Leung, Matthew E. H. Petering, and Terence W. H. Tou. **A Survey of Dial-a-Ride Problems: Literature Review and Recent Developments**. In *Transportation Research Part B: Methodological* volume 111, pages 1–27, 2018. DOI: `10.1016/j.trb.2018.02.001`.
Cited on page 93.

[HS18] Michael Hamann and Ben Strasser. **Graph Bisection with Pareto Optimization**. In *ACM Journal of Experimental Algorithmics* volume 23:1, pages 1.2:1–1.2:34, 2018. DOI: `10.1145/3173045`.
Cited on pages 9, 80, 130, 146.

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. **Engineering Multilevel Overlay Graphs for Shortest-Path Queries**. In *ACM Journal of Experimental Algorithmics* volume 13, pages 2.5:1–2.5:26, 2008. DOI: `10.1145/1412228.1412239`.
Cited on page 148.

[HW12] Wesam Herbawi and Michael Weber. **A Genetic and Insertion Heuristic Algorithm for Solving the Dynamic Ridematching Problem with Time Windows**. In *Proceedings of the 14th International Conference on Genetic and Evolutionary Computation (GECCO'12)*, pages 385–392. ACM Press, 2012. DOI: `10.1145/2330163.2330219`.
Cited on page 95.

[ITF19] International Transport Forum. ITF Transport Outlook 2019. OECD Publishing, 2019. ISBN: 978-92-82-10388-3. DOI: 10.1787/transp_outlook-en-2019-en.
Cited on page 1.

[JJP16] Jaeyoung Jung, R. Jayakrishnan, and Ji Young Park. Dynamic Shared-Taxi Dispatch Algorithm with Hybrid-Simulated Annealing. In *Computer-Aided Civil and Infrastructure Engineering* volume 31:4, pages 275–291, 2016. DOI: 10.1111/mice.12157.
Cited on pages 94, 96, 117.

[Joh75] Donald B. Johnson. Priority Queues with Update and Finding Minimum Spanning Trees. In *Information Processing Letters* volume 4:3, pages 53–57, 1975. DOI: 10.1016/0020-0190(75)90001-0.
Cited on pages 29, 50, 78, 112.

[JTPR94] R. Jayakrishnan, Wei Kang Tsai, Joseph Prashker, and Subodh Rajadhyaksha. Faster Path-Based Algorithm for Traffic Assignment. In *Transportation Research Record* volume 1443, pages 75–83, 1994.
Cited on page 61.

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. In *SIAM Journal on Scientific Computing* volume 20:1, pages 359–392, 1998. DOI: 10.1137/S1064827595287997.
Cited on pages 145, 147.

[KM77] Albert J. Kinderman and John F. Monahan. Computer Generation of Random Variables Using the Ratio of Uniform Deviates. In *ACM Transactions on Mathematical Software* volume 3:3, pages 257–260, 1977. DOI: 10.1145/355744.355750.
Cited on page 19.

[Kno+07] 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. DOI: 10.1137/1.9781611972870.4.
Cited on pages 10, 40, 63, 92, 94.

[Knu98] Donald E. Knuth. The Art of Computer Programming: Sorting and Searching. Addison-Wesley, 1998. ISBN: 978-0-2018-9685-5.
Cited on page 77.

[KP11] Amit Kumar and Srinivas Peeta. **An Improved Social Pressure Algorithm for Static Deterministic User Equilibrium Traffic Assignment Problem**. In *Proceedings of the 90th Transportation Research Board Annual Meeting (TRB'11)*, 2011.
Cited on page 61.

[KRS13] Moritz Kobitzsch, Marcel Radermacher, and Dennis Schieferdecker. **Evolution and Evaluation of the Penalty Method for Alternative Graphs**. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13)*. Ed. by Daniele Frigioni and Sebastian Stiller. Volume 33 of OpenAccess Series in Informatics (OASIcs), pages 94–107. Schloss Dagstuhl, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.94.
Cited on page 40.

[Kus14] Daniel Kusswurm. **Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX**. Apress, 2014. ISBN: 978-1-4842-0065-0.
Cited on page 78.

[Lau09] Ulrich Lauther. **An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Road Networks with Precalculated Edge-Flags**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Ed. by Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. Volume 74. DIMACS Book. American Mathematical Society, 2009, pages 19–39.
Cited on pages 126, 148.

[LLZ09] Ken C. K. Lee, Wang-Chien Lee, and Baihua Zheng. **Fast Object Search on Road Networks**. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT'09)*. Ed. by Martin L. Kersten, Boris Novikov, Jens Teubner, Vladimir Polutin, and Stefan Manegold, pages 1018–1029. ACM Press, 2009. DOI: 10.1145/1516360.1516476.
Cited on page 42.

[LLZT12] Ken C. K. Lee, Wang-Chien Lee, Baihua Zheng, and Yuan Tian. **ROAD: A New Spatial Object Search Framework for Road Networks**. In *IEEE Transactions on Knowledge and Data Engineering* volume 24:3, pages 547–560, 2012. DOI: 10.1109/TKDE.2010.243.
Cited on page 42.

[LM76] Steven A. Lippman and John J. McCall. **The Economics of Job Search: A Survey**. In *Economic Inquiry* volume 14:2, pages 155–189, 1976. DOI: 10.1111/j.1465-7295.1976.tb00386.x.
Cited on page 14.

[LS11] Dennis Luxen and Peter Sanders. **Hierarchy Decomposition for Faster User Equilibria on Road Networks**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Ed. by Panos M. Pardalos and Steffen Rebennack. Volume 6630 of Lecture Notes in Computer Science, pages 242–253. Springer, 2011. DOI: `10.1007/978-3-642-20662-7_21`.
Cited on pages 16, 62, 63, 66, 67, 87.

[McC70] John J. McCall. **Economics of Information and Job Search**. In *The Quarterly Journal of Economics* volume 84:1, pages 113–126, 1970. DOI: `10.2307/1879403`.
Cited on page 14.

[MJ17] Neda Masoud and R. Jayakrishnan. **A Real-Time Algorithm to Solve the Peer-to-Peer Ride-Matching Problem in a Flexible Ridesharing System**. In *Transportation Research Part B: Methodological* volume 106, pages 218–236, 2017. DOI: `10.1016/j.trb.2017.10.006`.
Cited on page 95.

[MKV13] Nicolai Mallig, Martin Kagerbauer, and Peter Vortisch. **mobiTopp – A Modular Agent-based Travel Demand Modelling Framework**. In *Proceedings of the 2nd International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABM-TRANS'13)*, pages 854–859, 2013. DOI: `10.1016/j.procs.2013.06.114`.
Cited on pages 29, 79.

[ML13] Maria Mitradjieva and Per Olov Lindberg. **The Stiff Is Moving – Conjugate Direction Frank-Wolfe Methods with Applications to Traffic Assignment**. In *Transportation Science* volume 47:2, pages 280–293, 2013. DOI: `10.1287/trsc.1120.0409`.
Cited on pages 61, 65, 66.

[MS04] Burkhard Monien and Stefan Schamberger. **Graph Partitioning with the Party Library: Helpful-Sets in Practice**. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004. DOI: `10.1109/SBAC-PAD.2004.18`.
Cited on page 145.

[MS10] Matthias Müller-Hannemann and Stefan Schirra (editors). **Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice**. Volume 5971 of Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-14865-1. DOI: `10.1007/978-3-642-14866-8`.
Cited on page 2.

[MSJB13] A. Paolo Masucci, Joan Serras, Anders Johansson, and Michael Batty. **Gravity versus Radiation Models: On the Importance of Scale and Heterogeneity in Commuting Flows**. In *Physical Review E* volume 88:2, pages 1–8, 2013. DOI: 10.1103/PhysRevE.88.022812.
Cited on page 15.

[MV15] Nicolai Mallig and Peter Vortisch. **Modeling Car Passenger Trips in mobiTopp**. In *Proceedings of the 4nd International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'15)*, pages 938–943, 2015. DOI: 10.1016/j.procs.2015.05.169.
Cited on pages 29, 79.

[MZW13] Shuo Ma, Yu Zheng, and Ouri Wolfson. **T-Share: A Large-Scale Dynamic Taxi Ridesharing Service**. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE'13)*. Ed. by Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, pages 410–421. IEEE Computer Society, 2013. DOI: 10.1109/ICDE.2013.6544843.
Cited on pages 94, 96, 117.

[Nie10] Yu Marco Nie. **A Class of Bush-Based Algorithms for the Traffic Assignment Problem**. In *Transportation Research Part B: Methodological* volume 44:1, pages 73–89, 2010. DOI: 10.1016/j.trb.2009.06.005.
Cited on page 61.

[OB98] Leonid Oliker and Rupak Biswas. **PLUM: Parallel Load Balancing for Adaptive Unstructured Meshes**. In *Journal of Parallel and Distributed Computing* volume 52:2, pages 150–177, 1998. DOI: 10.1006/jpdc.1998.1469.
Cited on pages 146, 147.

[Pel+15] Dominik Pelzer, Jiajian Xiao, Daniel Zehe, Michael H. Lees, Alois C. Knoll, and Heiko Aydt. **A Partition-Based Match Making Algorithm for Dynamic Ridesharing**. In *IEEE Transactions on Intelligent Transportation Systems* volume 16:5, pages 2587–2598, 2015. DOI: 10.1109/TITS.2015.2413453.
Cited on page 94.

[PERW15] Olga Perederieieva, Matthias Ehrgott, Andrea Raith, and Judith Y. T. Wang. **A Framework for and Empirical Study of Algorithms for Traffic Assignment**. In *Computers & Operations Research* volume 54, pages 90–107, 2015. DOI: 10.1016/j.cor.2014.08.024.
Cited on pages 16, 60, 61, 62, 66, 79, 87.

[PR96] François Pellegrini and Jean Roman. **Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs**. In *Proceedings of the 4th International Conference and Exhibition on High-Performance Computing and Networking (HPCN'96)*. Ed. by Heather M. Liddell, Adrian Colbrook, Louis O. Hertzberger, and Peter M. A. Sloot. Volume 1067 of Lecture Notes in Computer Science, pages 493–498. Springer, 1996. DOI: `10.1007/3-540-61142-8_588`.
Cited on page 145.

[PS98] Stefano Pallottino and Maria Grazia Scutellà. **Shortest Path Algorithms In Transportation Models: Classical and Innovative Aspects**. In *Equilibrium and Advanced Transportation Modelling*. Ed. by Patrice Marcotte and Sang Nguyen. Springer, 1998, pages 245–281. DOI: `10.1007/978-1-4615-5757-9_11`.
Cited on page 95.

[PSS95] Eddy Pramono, Horst D. Simon, and Andrew Sohn. **Dynamic Load Balancing for Finite Element Calculations on Parallel Computers**. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*. Ed. by David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia Torczon, and Layne T. Watson, pages 599–604. SIAM, 1995.
Cited on pages 146, 147.

[PST17] Ruud van der Pas, Eric Stotzer, and Christian Terboven. **Using OpenMP—The Next Step**. MIT Press, 2017. ISBN: 978-0-2625-3478-9.
Cited on page 71.

[PTV14] PTV AG. **PTV Visum 14 Manual**. 2014.
Cited on page 66.

[PZMT03] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. **Query Processing in Spatial Network Databases**. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*. Ed. by Johann-Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, pages 802–813. Morgan Kaufmann, 2003.
Cited on pages 41, 42.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. **Near Linear Time Algorithm to Detect Community Structures in Large-scale Networks**. In *Physical Review E* volume 76:3, 2007. DOI: `10.1103/PhysRevE.76.036106`.
Cited on page 151.

[Res] Transportation Networks for Research Core Team. **Transportation Networks for Research**. URL: `https://github.com/bstabler`.
Cited on page 137.

[San09] Peter Sanders. **Algorithm Engineering – An Attempt at a Definition**. In *Efficient Algorithms*. Ed. by Susanne Albers, Helmut Alt, and Stefan Näher. Volume 5760. Lecture Notes in Computer Science. Springer, 2009, pages 321–340. DOI: `10.1007/978-3-642-03456-5_22`.
Cited on page 2.

[SAS05] Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet. **Efficient Query Processing on Spatial Networks**. In *Proceedings of the 13th ACM International Workshop on Geographic Information Systems (GIS'05)*. Ed. by Cyrus Shahabi and Omar Boucelma, pages 200–209. ACM Press, 2005. DOI: `10.1145/1097064.1097093`.
Cited on page 42.

[Sav85] Martin W. P. Savelsbergh. **Local Search in Routing Problems with Time Windows**. In *Annals of Operations Research* volume 4:1, pages 285–305, 1985. DOI: `10.1007/BF02022044`.
Cited on page 94.

[SBR06] Howard Slavin, Jonathan Brandon, and Andres Rabinowicz. **An Empirical Comparison of Alternative User Equilibrium Traffic Assignment Methods**. In *Proceedings of the 34th European Transport Conference (ETC'06)*, 2006.
Cited on page 66.

[Sch59] Morton Schneider. **Gravity Models and Trip Distribution Theory**. In *Papers in Regional Science* volume 5:1, pages 51–56, 1959. DOI: `10.1111/j.1435-5597.1959.tb01665.x`.
Cited on page 12.

[SGMB12] Filippo Simini, Marta C. González, Amos Maritan, and Albert-László Barabási. **A Universal Model for Mobility and Migration Patterns**. In *Nature* volume 484:7392, pages 96–100, 2012. DOI: `10.1038/nature10856`.
Cited on pages 12, 13, 14, 15, 35, 48.

[She85] Yosef Sheffi. **Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods**. Prentice Hall, 1985. ISBN: 0-1393-9729-9.
Cited on pages 60, 65, 66.

[SHP11] Johannes Schlaich, Udo Heidl, and R. Pohlner. **Verkehrsmodellierung für die Region Stuttgart: Schlussbericht**. Unpublished, 2011.
Cited on pages 29, 79.

[Sin10] Johannes Singler. **Algorithm Libraries for Multi-Core Processors**. PhD thesis. Karlsruhe Institute of Technology, 2010.
Cited on page 72.

[SKK00] Kirk Schloegel, George Karypis, and Vipin Kumar. **A Unified Algorithm for Load-balancing Adaptive Scientific Simulations**. In *Proceedings of the 13th ACM/IEEE Supercomputing Conference (SC'00)*. Ed. by Louis H. Turcotte, pages 1–11. IEEE Computer Society, 2000. DOI: 10.1109/SC.2000.10035.
Cited on page 146.

[SKK01] Kirk Schloegel, George Karypis, and Vipin Kumar. **Wavefront Diffusion and LMSR: Algorithms for Dynamic Repartitioning of Adaptive Meshes**. In *IEEE Transactions on Parallel and Distributed Systems* volume 12:5, pages 451–466, 2001. DOI: 10.1109/71.926167.
Cited on pages 146, 147.

[SKK97] Kirk Schloegel, George Karypis, and Vipin Kumar. **Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes**. In *Journal of Parallel and Distributed Computing* volume 47:2, pages 109–124, 1997. DOI: 10.1006/jpdc.1997.1410.
Cited on pages 146, 147.

[SMDD19] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. **Sequential and Parallel Algorithms and Data Structures – The Basic Toolbox**. Springer, 2019. ISBN: 978-3-030-25208-3. DOI: 10.1007/978-3-030-25209-0.
Cited on pages 7, 75, 148.

[SMN13] Filippo Simini, Amos Maritan, and Zoltán Néda. **Human Mobility in a Continuum Approach**. In *PLOS ONE* volume 8:3, pages 1–8, 2013. DOI: 10.1371/journal.pone.0060069.
Cited on pages 13, 15, 48.

[SN20] Arne Schneck and Klaus Nökel. **Accelerating Traffic Assignment with Customizable Contraction Hierarchies**. In *Transportation Research Record* volume 2674:1, pages 188–196, 2020. DOI: 10.1177/0361198119898455.
Cited on pages 16, 149.

[SS05] Peter Sanders and Dominik Schultes. **Highway Hierarchies Hasten Exact Shortest Path Queries**. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*. Ed. by Gerth Stølting Brodal and Stefano Leonardi. Volume 3669 of Lecture Notes in Computer Science, pages 568–579. Springer, 2005. DOI: 10.1007/11561071_51.
Cited on page 83.

[SS12] Peter Sanders and Christian Schulz. **Distributed Evolutionary Graph Partitioning**. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. Ed. by David A. Bader and Petra Mutzel, pages 16–29. SIAM, 2012. DOI: 10.1137/1.9781611972924.2.
Cited on page 146.

[SS13] Peter Sanders and Christian Schulz. **Think Locally, Act Globally: Highly Balanced Graph Partitioning**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Ed. by Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela. Volume 7933 of Lecture Notes in Computer Science, pages 164–175. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_16.
Cited on pages 80, 146.

[SS15] Aaron Schild and Christian Sommer. **On Balanced Separators in Road Networks**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Ed. by Evripidis Bampis. Volume 9125 of Lecture Notes in Computer Science, pages 286–297. Springer, 2015. DOI: 10.1007/978-3-319-20086-6_22.
Cited on pages 9, 51, 80, 114, 137, 144, 146, 149.

[SS94] Andrew Sohn and Horst D. Simon. **JOVE: A Dynamic Load Balancing Framework for Adaptive Computations on an SP-2 Distributed-Memory Multiprocessor**. In tech. rep., 1994.
Cited on pages 146, 147.

[SSA08] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. **Scalable Network Distance Browsing in Spatial Databases**. In *Proceedings of the 27th ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. Ed. by Jason Tsong-Li Wang, pages 43–54. ACM Press, 2008. DOI: 10.1145/1376616.1376623.
Cited on page 42.

[Sta90] Ernst Stadlober. **The Ratio of Uniforms Approach for Generating Discrete Random Variates**. In *Journal of Computational and Applied Mathematics* volume 31:1, pages 181–189, 1990. DOI: 10.1016/0377-0427(90)90349-5.
Cited on page 19.

[Sto40] Samuel A. Stouffer. **Intervening Opportunities: A Theory Relating Mobility and Distance**. In *American Sociological Review* volume 5:6, pages 845–867, 1940. DOI: 10.2307/2084520.
Cited on page 12.

[SW11] Peter Sanders and Dorothea Wagner. **Algorithm Engineering**. In *it – Information Technology* volume 53:6, pages 263–265, 2011. DOI: 10.1524/itit.2011.9072.
Cited on page 2.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. **Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport**. In *ACM Journal of Experimental Algorithmics* volume 5, pages 1–23, 2000. DOI: 10.1145/351827.384254.
Cited on pages 95, 148.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Using Multi-level Graphs for Timetable Information in Railway Systems**. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*. Ed. by David M. Mount and Clifford Stein. Volume 2409 of Lecture Notes in Computer Science, pages 43–59. Springer, 2002. DOI: 10.1007/3-540-45643-0_4.
Cited on page 148.

[Tiz+14] Michele Tizzoni, Paolo Bajardi, Adeline Decuyper, Guillaume Kon Kam King, Christian M. Schneider, Vincent D. Blondel, Zbigniew Smoreda, Marta C. González, and Vittoria Colizza. **On the Use of Human Mobility Proxies for Modeling Epidemics**. In *PLOS Computational Biology* volume 10:7, pages 1–15, 2014. DOI: 10.1371/journal.pcbi.1003716.
Cited on page 12.

[Ves12] Alessandro Vespignani. Modelling Dynamical Processes in Complex Sociotechnical Systems. In *Nature Physics* volume 8:1, pages 32–39, 2012. DOI: 10.1038/NPHYS2160.
Cited on page 12.

[VRS11] VRS Verband Region Stuttgart. Mobilität und Verkehr in der Region Stuttgart 2009/2010: Regionale Haushaltsbefragung zum Verkehrsverhalten. In *Schriftenreihe Verband Region Stuttgart* volume 29, pages 1–138, 2011.
Cited on pages 29, 79.

[Wal09] Chris Walshaw. Variable Partition Inertia: Graph Repartitioning and Load Balancing for Adaptive Meshes. In *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*. Ed. by Manish Parashar and Xiaolin Li. John Wiley & Sons, 2009, pages 357–380. DOI: 10.1002/9780470558027.ch17.
Cited on page 147.

[War52] John G. Wardrop. Some Theoretical Aspects of Road Traffic Research. In *Proceedings of the Institution of Civil Engineers*, pages 325–362, 1952. DOI: 10.1680/ipeds.1952.11259.
Cited on page 60.

[WB95] Chris Walshaw and Martin Berzins. Dynamic Load-balancing for PDE Solvers on Adaptive Unstructured Meshes. In *Concurrency: Practice and Experience* volume 7:1, pages 17–28, 1995. DOI: 10.1002/cpe.4330070103.
Cited on page 147.

[WCE97] Chris Walshaw, Mark Cross, and Martin G. Everett. Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. In *Journal of Parallel and Distributed Computing* volume 47:2, pages 102–108, 1997. DOI: 10.1006/jpdc.1997.1407.
Cited on page 146.

[WE80] Chak-Kuen Wong and Malcolm C. Easton. An Efficient Method for Weighted Sampling Without Replacement. In *SIAM Journal on Computing* volume 9:1, pages 111–113, 1980. DOI: 10.1137/0209009.
Cited on page 25.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. In *Communications of the ACM* volume 7:6, pages 347–348, 1964. DOI: 10.1145/512274.512284.
Cited on page 19.

[Win02] Stephan Winter. **Modeling Costs of Turns in Route Planning**. In *GeoInformatica* volume 6:4, pages 345–361, 2002. DOI: 10.1023/A:1020853410145.
Cited on pages 126, 127.

[Yan10] Hiroki Yanagisawa. **A Multi-Source Label-Correcting Algorithm for the All-Pairs Shortest Paths Problem**. In *24th IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*, pages 1–10. IEEE Computer Society, 2010. DOI: 10.1109/IPDPS.2010.5470362.
Cited on page 76.

[YHEG14] Yingxiang Yang, Carlos Herrera, Nathan Eagle, and Marta C. González. **Limits of Predictability in Commuting Flows in the Absence of Data for Calibration**. In *Scientific Reports* volume 4:5662, pages 1–9, 2014. DOI: 10.1038/srep05662.
Cited on page 27.

[Zho+15] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong. **G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks**. In *IEEE Transactions on Knowledge and Data Engineering* volume 27:8, pages 2175–2189, 2015. DOI: 10.1109/TKDE.2015.2399306.
Cited on page 42.

[Zho93] Pei-bai Zhou. **Numerical Analysis of Electromagnetic Fields**. Springer, 1993. ISBN: 978-3-642-50321-4. DOI: 10.1007/978-3-642-50319-1.
Cited on page 146.

[Zip46] George Kingsley Zipf. **The P_1P_2/D Hypothesis: On the Intercity Movement of Persons**. In *American Sociological Review* volume 11:6, pages 677–686, 1946. DOI: 10.2307/2087063.
Cited on page 12.

[ZKN19] Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. **The MATSim Open Berlin Scenario: A Multimodal Agent-Based Transport Simulation Scenario Based on Synthetic Demand Modeling and Open Data**. In *Proceedings of the 8th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'19)*. Volume 151 of Procedia Computer Science, pages 870–877. Elsevier, 2019. DOI: 10.1016/j.procs.2019.04.120.
Cited on pages 93, 111, 112.

[ZLTZ13] Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. **G-Tree: An Efficient Index for KNN Search on Road Networks**. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM'13)*. Ed. by Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, pages 39–48. ACM Press, 2013. DOI: `10.1145/2505515.2505749`.
Cited on page 42.

[ZTF14] Olgierd C. Zienkiewicz, Robert L. Taylor, and David D. Fox. **The Finite Element Method for Solid and Structural Mechanics**. Butterworth-Heinemann, 2014. ISBN: 978-1-85617-634-7. DOI: `10.1016/C2009-0-26332-X`.
Cited on page 146.

[ZTN14] Olgierd C. Zienkiewicz, Robert L. Taylor, and Perumal Nithiarasu. **The Finite Element Method for Fluid Dynamics**. Butterworth-Heinemann, 2014. ISBN: 978-1-85617-635-4. DOI: `10.1016/C2009-0-26328-8`.
Cited on page 146.

[ZYC11] Tian-ran Zhang, Chao Yang, and Dong-dong Chen. **Modified Origin-Based Algorithm for Traffic Equilibrium Assignment Problems**. In *Journal of Central South University of Technology* volume 18:5, pages 1765–1772, 2011. DOI: `10.1007/s11771-011-0900-6`.
Cited on page 66.