An Efficient Solution for One-To-Many Multi-Modal Journey Planning

Jonas Sauer
Karlsruhe Institute of Technology (KIT), Germany
jonas.sauer2@kit.edu

Dorothea Wagner
Karlsruhe Institute of Technology (KIT), Germany
dorothea.wagner@kit.edu

Tobias Zündorf
Karlsruhe Institute of Technology (KIT), Germany
zuendorf@kit.edu

Abstract

We study the one-to-many journey planning problem in multi-modal transportation networks consisting of a public transit network and an additional, non-schedule-based mode of transport. Given a departure time and a single source vertex, we aim to compute optimal journeys to all vertices in a set of targets, optimizing both travel time and the number of transfers used. Solving this problem yields a crucial component in many other problems, such as efficient point-of-interest queries, computation of isochrones, or multi-modal traffic assignments. While many algorithms for multi-modal journey planning exist, none of them are applicable to one-to-many scenarios. Our solution is based on the combination of two state-of-the-art approaches: ULTRA, which enables efficient journey planning in multi-modal networks, but only for one-to-one queries, and (R)PHAST, which enables efficient one-to-many queries, but only in time-independent networks. Similarly to ULTRA, our new approach can be combined with any existing public transit algorithm that allows a search to all stops, which we demonstrate for CSA and RAPTOR. For small to moderately sized target sets, the resulting algorithms are nearly as fast as the pure public transit algorithms they are based on. For large target sets, we achieve a speedup of up to 7 compared to a naive one-to-many extension of a state-of-the-art multi-modal approach.

2012 ACM Subject Classification Theory of computation → Shortest paths; Mathematics of computing → Graph algorithms; Applied computing → Transportation

Keywords and phrases Algorithm Engineering, Route Planning, Public Transit, One-to-Many

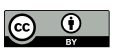
Digital Object Identifier 10.4230/OASIcs.ATMOS.2020.1

Supplementary Material Source code is available at `https://github.com/kit-algo/ULTRA-PHAST`.

Funding This research was funded by the DFG under grant number WA 654123-2.

1 Introduction

Recent years have seen considerable advances in fast route planning algorithms for both road and public transit networks [3]. The combination of both network types into a multi-modal journey planning problem, however, remains challenging [9]. In this work, we consider multi-modal networks that combine a public transit network with a transfer graph that represents one additional mode of non-schedule-based transportation (e.g., walking or cycling). Most existing research on multi-modal journey planning has focused on solving *one-to-one* queries, which ask for journeys between a single source and target vertex. Related to this are the *one-to-many* and *one-to-all* problems, where multiple or all vertices are considered as targets. While studied extensively for road networks, these problems have received little attention



on multi-modal networks so far. In this paper, we close this gap by adapting the recently proposed ULTRA [8] algorithm family, which solves one-to-one queries in multi-modal networks, to one-to-many and one-to-all scenarios.

There are many potential applications of one-to-many and many-to-many search in both road networks and multi-modal networks. These include extended query scenarios such as building distance tables for vehicle routing and traveling salesman problems [32, 17], point-of-interest (POI) queries (e.g., finding the k nearest stores) [17], and isochrones, which are the set of vertices and/or edges reachable from a given point within a given distance or time limit. Isochrones have been subject to algorithmic research on both road networks [23, 6, 7, 5] and multi-modal networks [24, 25, 33], but so far algorithms for multi-modal isochrones are limited to Dijkstra search on a graph representation of the network. Another area where one-to-many algorithms can be applied are preprocessing techniques for the one-to-one problem. On road networks, a prominent example is Arc-Flags [29], whose preprocessing phase can be significantly sped up by using the one-to-all algorithm PHAST [13]. Examples of public transit algorithms whose preprocessing phase involves one-to-many search are Transfer Patterns [2] and Access Node Routing [15]. So far, no comparable speedup technique for multi-modal networks has been developed, partly due to prohibitively high preprocessing costs. A more efficient one-to-many search algorithm for multi-modal networks could be a first step towards developing such a technique. Finally, many-to-many routing is used as a component in simulation-based traffic assignment algorithms, such as the CSA-based approach presented in [10]. A multi-modal variant based on ULTRA was proposed in [38], but it uses a naive adaptation of ULTRA to a many-to-one setting, which is only feasible if the set of source vertices where passenger demand is located is fairly small. A scalable multi-modal one-to-all algorithm could enable the computation of full door-to-door assignments.

Related Work. Public transit routing algorithms can be divided into graph-based approaches (e.g., [35, 22, 28, 30, 31]) and algorithms that exploit the structure of public transit timetables to achieve faster query times. Prominent examples of the latter include RAPTOR [16], CSA [18, 19], and Trip-Based Routing [40]. A technique that utilizes heavy preprocessing to achieve very fast query times is Transfer Patterns [2, 4]. Common to these algorithms is that they only consider non-schedule-based transport in the form of a restricted transfer graph, which is often required to be transitively closed. However, recent experiments have shown that the availability of unrestricted walking significantly reduces travel times [39, 37, 34].

Multi-modal algorithms lift these restrictions on the transfer graph by interleaving existing public transit algorithms with an exploration of the unrestricted transfer graph. UCCH [20] and MCR [11] combine graph-based techniques and RAPTOR, respectively, with Dijkstra [21] searches on a contracted transfer graph. HLRaptor and HLCSA [34] explore the transfer graph with two-hop searches based on Hub Labeling [1]. The most recent approach is ULTRA [8], which utilizes the observation that the number of unique *intermediate* transfers, i.e., transfers between two public transit vehicles, that occur in optimal journeys is much lower than the number of *initial* and *final* transfers, which connect the source and target vertex to the public transit network. This is exploited by precomputing a small set of shortcuts representing all necessary intermediate transfers. The initial and final transfers are computed at query time using Bucket-CH [32, 26, 27], a technique for fast one-to-many searches on road networks. Together, this enables existing public transit algorithms, such as RAPTOR and CSA, to handle multi-modal networks without specific adjustments or a significant performance loss. Unfortunately, none of these multi-modal algorithms support one-to-many queries because they all involve bidirectional search from the source and target vertex, which is inherently a one-to-one technique.

By contrast, several algorithms have been proposed for one-to-many, one-to-all or many-to-many search on road networks: A popular solution for adapting speedup techniques that were originally developed for one-to-one queries is a bucket-based approach, which has been applied to Highway Hierarchies [32], Contraction Hierarchies (CH) [26, 27] and Hub Labeling [1, 14]. The Customizable Route Planning [12] technique has also been adapted to one-to-many search, resulting in the GRASP algorithm [23], and to the closely related setting of POI queries [17]. For one-to-all search, PHAST [13] employs vertex reordering and GPU parallelization to create a fast, memory-efficient sweeping algorithm. RPHAST [14] extends this approach to one-to-many search by adding a target selection phase.

Our Contribution. We combine ULTRA with ideas adapted from RPHAST to create an algorithm scheme called ULTRA-PHAST, which is the first efficient approach for one-to-many queries in multi-modal networks. Like ULTRA, ULTRA-PHAST uses precomputed shortcuts for intermediate transfers. Our main contribution is adapting RPHAST to efficiently explore the final transfers to the target vertices, which is more challenging than a normal one-to-many shortest path problem as every stop reached via the public transit network may be a potential source vertex. As with ULTRA, ULTRA-PHAST is an algorithmic framework that can be combined with any public transit algorithm that supports one-to-all search. We combine ULTRA-PHAST with two state-of-the-art public transit algorithms, CSA and RAPTOR. We evaluate the performance of the resulting algorithms, UP-CSA and UP-RAPTOR, on the networks of Switzerland and Germany.

2 Preliminaries

In this section we introduce the basic definitions used throughout the paper, the routing problems we consider, and the algorithms upon which our work is based.

Public Transit Network. A public transit network is a 3-tuple $(\mathcal{S}, T, G)$ consisting of a set of *stops* $\mathcal{S}$, a timetable T, and a directed, weighted transfer graph $G = (\mathcal{V}, \mathcal{E})$. A stop is a location where passengers can enter or exit a public transit vehicle (e.g., bus, train, ferry). The timetable T defines how the vehicles move between the stops. Since different algorithms model the timetable in different ways, and ULTRA can be combined with any public transit algorithm, we treat the timetable as a black box. The only terminology we require is that of the *trip*, which represents a vehicle traveling along a sequence of stops at a specific point in time. The transfer graph $G = (\mathcal{V}, \mathcal{E})$ consists of a set of *vertices* $\mathcal{V}$ with $\mathcal{S} \subseteq \mathcal{V}$, and a set of *edges* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. It may represent any non-schedule-based mode of transportation. For each edge $e = (u, v) \in \mathcal{E}$, the *transfer time* $w(e)$ is the time required to transfer from u to v.

Problem Statement. Given source and target vertices $s, t \in \mathcal{V}$, an *s-t-journey* J represents the movement of a passenger from s to t through the public transit network. The modeling of the journey's components depends on the modeling used for the timetable, so again we view it as a black box. The attributes we use for evaluating a journey are the departure time $\tau_{\text{dep}}(J)$ at s, the arrival time $\tau_{\text{arr}}(J)$ at t, and the number of trips used by the passenger during the journey. We say that a journey J *dominates* another journey J' if $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}(J')$, $\tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J')$ and J does not use more trips than J'. A journey is *Pareto-optimal* if it is not dominated by another journey. A *Pareto set* is a minimal set of journeys such that every possible journey from s to t is dominated by a journey in the Pareto set.

We consider two variants of the one-to-many routing problem: the *earliest arrival problem* and the *Pareto optimization problem*. In both cases, we are given a public transit network $(\mathcal{S}, T, G = (\mathcal{V}, \mathcal{E}))$, a source vertex $s \in \mathcal{V}$, a set of target vertices $\mathcal{T} \subseteq \mathcal{V}$, and a departure time τ_{dep}. The objective of the *earliest arrival problem* is to find, for each target $t \in \mathcal{T}$, an s-t-journey that departs no earlier than τ_{dep} and minimizes the arrival time at t. The *Pareto optimization problem* instead asks for a Pareto set of s-t-journeys departing no earlier than τ_{dep}, using arrival time and number of trips as the optimization criteria.

Algorithms. We now give an overview of the algorithms our work is based on, namely (R)PHAST and ULTRA. PHAST is itself an extension of Contraction Hierarchies (CH). The basic operation of the CH preprocessing phase is *vertex contraction*, which removes a vertex from the graph and inserts shortcut edges between its neighbors to preserve distances. This is done iteratively until all vertices are contracted. The order in which the vertices are contracted is called the *contraction order*. The *rank* of a vertex is its position in the contraction order. This iterative contraction yields two graphs: The *upward graph* $G^{\uparrow}$ consists of all original edges and shortcuts whose head vertex has a higher rank than the tail vertex (i.e., was contracted later). Conversely, the *downward graph* $G^{\downarrow}$ contains the edges whose head vertex has a lower rank than the tail vertex.

A PHAST query begins with an upward search from s in $G^{\uparrow}$. This is followed by a *downward sweep* that scans the vertices of $G^{\downarrow}$ in some topological order (i.e., the tail vertex of each edge is scanned before its head vertex). An example of a topological order is the contraction order, but any other topological order is valid as well. For each scanned vertex v and each incoming downward edge $e = (u, v)$, the distance $\text{dist}(v)$ of v is set to the minimum of $\text{dist}(u) + w(e)$ and $\text{dist}(v)$. To make the downward sweep cache-efficient, the vertices of $G^{\downarrow}$ are stored in memory in the same order in which they are scanned. In a many-to-all scenario, where more than one source vertex is given, the memory locality of PHAST can be further improved by combining k one-to-all searches into a single sweep (for a fixed k). Instead of a single distance value per vertex, the algorithm then stores an array of k distance values, one for each of the k sources, which are updated consecutively during each edge relaxation.

If we are only interested in distances to a subset $\mathcal{T} \subseteq \mathcal{V}$ of vertices, and $\mathcal{T}$ does not change between queries, RPHAST (restricted PHAST) improves on PHAST by performing a *target selection* phase before queries are run. This involves running a backward breadth-first search (BFS) on $G^{\downarrow}$, initializing the queue with all target vertices at once. The downward sweep is then run on $G^{\downarrow}[\mathcal{T}]$, the subgraph of $G^{\downarrow}$ induced by the vertices visited by the BFS.

Finally, we recapitulate the algorithmic framework of ULTRA: A preprocessing phase computes shortcuts for all intermediate transfers (i.e, transfers between two trips) that occur in an optimal journey. The initial and final transfers are handled via a Bucket-CH query. Bucket-CH is an extension of CH for one-to-many queries that stores a *bucket* of distances to the target vertices at each vertex. The buckets are computed via a backward search in $G^{\downarrow}$ for each target $t \in \mathcal{T}$. For each vertex v reached by this search, an entry storing the distance to t is added to the bucket of v. A Bucket-CH query consists of an upward search from s in $G^{\uparrow}$, followed by scanning the bucket of each reached vertex to compute the distances to the targets. ULTRA runs a forward Bucket-CH query from s to all stops and a backward Bucket-CH query from all stops to t. Afterwards, a public transit algorithm, such as RAPTOR and CSA, is run from the stops reached by the forward Bucket-CH search, using the precomputed shortcuts to explore intermediate transfers. Whenever it reaches a vertex v that was reached by the backward Bucket-CH search, the resulting arrival time at t is computed as the sum of the arrival time at v and the distance between v and t.

3 Algorithm

Before we propose our one-to-many adaptation of ULTRA, we examine why the original ULTRA algorithm cannot answer one-to-many queries. The ULTRA query algorithm uses a bidirectional Bucket-CH search to explore the initial and final transfers. This requires a single target vertex to run the backward search from. A naive solution [38] to this problem is to perform multiple backward searches, one from each target vertex. However, this is only viable for very small target sets, as the running time is proportional to the number of targets. We therefore replace the backward search for the final transfers with a forward search inspired by PHAST. We first outline our approach in detail for the arrival time problem. Afterwards, we show how it can be generalized for the Pareto optimization problem.

3.1 Earliest Arrival Queries

The naive approach of performing one Bucket-CH search per target solves a many-to-many problem, computing the distances between all stops and all targets. This is more information than is required in our case: For each target t, we only require the distance from a single stop, namely the stop where the last used trip is exited in the optimal journey to t. The difficulty lies in the fact that we do not know this stop in advance. However, we can reformulate the final transfer search as a one-to-many problem and solve it using PHAST in the following manner: First, we compute the earliest arrival time at each stop $v \in \mathcal{S}$ using a standard ULTRA query without the backward Bucket-CH search and without target pruning. Afterwards, we insert a temporary edge (s, v) with weight $\tau_{\text{arr}}(v) - \tau_{\text{dep}}$ into the PHAST upward graph $G^{\uparrow}$. We can then find the earliest arrival time at every target with a single PHAST search that uses our augmented graph $G^{\uparrow}$. If we are also interested in the corresponding journey, we can simply substitute the temporary edge (s, v) with the journey to v found by the ULTRA query. In practice, we do not actually insert temporary edges into $G^{\uparrow}$. Instead, we initialize the priority queue used for the search in $G^{\uparrow}$ by directly inserting each stop v with $\tau_{\text{arr}}(v)$ as its distance.

As presented thus far, our approach still has a performance issue: The efficiency of the upward search in $G^{\uparrow}$, which comprises the first phase of PHAST, relies on the fact that the upward search space of a single source vertex is small. However, we perform an upward search from all reached stops simultaneously. Hence, the search space of our upward search will be the union of the search spaces of all stops, which is a large portion of the graph.

Efficient Upward Search. In order to improve the efficiency of the upward search, we optimize its memory and cache usage. First, we note that only vertices in the upward search space of a stop are relevant for our algorithm. Since the set of stops does not change between queries and is known beforehand, we can perform a *stop selection* analogous to the target selection in RPHAST: We run a forward BFS on $G^{\uparrow}$ from all stops simultaneously, and remove all vertices that are not visited. The resulting stop-selected upward graph is denoted as $G^{\uparrow}[\mathcal{S}]$. Furthermore, we observe that if the transfer graph is strongly connected, every query will reach every stop, regardless of the source vertex. Thus, every vertex in the stop-selected upward graph will be visited during the upward search. We can therefore replace the Dijkstra search in $G^{\uparrow}[\mathcal{S}]$, which requires a priority queue, with a more efficient upward sweep that is done analogously to the downward sweep of PHAST. If the transfer graph is not strongly connected, such a sweep might scan many unreachable stops. Thus, we modify the ULTRA query to keep track of the stop with the lowest rank that has been reached and start the upward sweep at this stop.

Algorithm 1 ULTRA-PHAST query algorithm.

```
1 Dijkstra search from s in G↑                    // initialize the arrival time at s as τ_dep
2 Downward sweep in G↓[S]
3 Initialize stops for the public transit query   // using the arrival times found in line 2
4 Run the public transit query                    // without target pruning
5 Upward sweep in G↑[S]                           // initialized with arrival times found in line 4
6 Downward sweep in G↓[T]
```

Algorithm Overview. The algorithmic framework for our one-to-many approach, which we call ULTRA-PHAST, is outlined in Algorithm 1. The original ULTRA query explored initial transfers with a Bucket-CH search from s, using the results of a backward Bucket-CH search from the target vertex to prune the search space. Since this pruning technique is no longer applicable in a scenario with multiple target vertices, the initial transfer search will reach all stops that are reachable from s. In this case, it is more efficient to explore the initial transfers with an RPHAST search to $\mathcal{S}$ instead of Bucket-CH. The RPHAST search consists of an upward search from s in the CH upward graph $G^{\uparrow}$ (line 1), and a downward sweep on the stop-selected downward graph $G^{\downarrow}[\mathcal{S}]$ (line 2). The public transit part of the network is then explored using a black-box public transit algorithm without target pruning. The public transit query is initialized with the arrival times at the stops found by the RPHAST search in line 3 and then run in line 4. It yields minimal arrival times for all stops in the network, which we then propagate to the target set using a final upward and downward sweep in lines 5 and 6. Since the upward sweep is equivalent to an RPHAST downward sweep in reverse, its running time should be comparable. Thus, the total running time of an ULTRA-PHAST query is roughly equal to the combined running time of a public transit query without target pruning, two RPHAST queries to $\mathcal{S}$, and one RPHAST query to $\mathcal{T}$.

Optimized Contraction Order. The three sweeps can be further sped up by delaying the contraction of stops and targets during the CH computation. Specifically, delaying the contraction of stops will reduce the number of vertices in $G^{\downarrow}[\mathcal{S}]$ and $G^{\uparrow}[\mathcal{S}]$, while delaying the contraction of targets will reduce the number of vertices in $G^{\downarrow}[\mathcal{T}]$. However, this is only useful up to a certain point, since eventually the quality of the contraction order will degrade. This will either lead to an unreasonable preprocessing time or cause too many shortcuts to be inserted, which will in turn slow down the sweeps. We take this into account by introducing tuning parameters f_{s} and f_{t} that determine how much the contraction of stops and targets is delayed, respectively. Initially, only vertices that are neither a stop or a target may be contracted. Once fewer than $f_{\mathsf{t}}|\mathcal{S} \cup \mathcal{T}|$ uncontracted vertices remain, we also allow targets to be contracted. Stops remain uncontractable until fewer than $f_{\mathsf{s}}|\mathcal{S}|$ vertices remain.

Vertex Reordering. As demonstrated in [36] and [13], the order in which the vertices of a graph are stored in memory can have a significant impact on the performance of a routing algorithm. In particular, the order in which vertices are settled by a DFS has been shown to lead to good memory locality for Dijkstra-like searches. For the sweeps on the upward graph $G^{\uparrow}[\mathcal{S}]$ as well as the downward graphs $G^{\downarrow}[\mathcal{S}]$ and $G^{\downarrow}[\mathcal{T}]$, the vertices must be scanned in a topological order, to ensure that the tail vertex of each edge is scanned before its head vertex. We obtain a topological order via DFS on $G^{\uparrow}$ and reorder the vertices according to it. Preliminary experiments have shown that this order performs at least as well as the level order used by PHAST, which was chosen primarily because it allows for easy parallelization.

Algorithm 2 Downward sweep to targets.

```
1  timestamp++
2  for i ← 0, ..., |V↓[T]| − 1 do
3      v ← ID of the vertex in V that corresponds to i
4      if timestamp[v] ≠ timestamp then
5          timestamp[v] ← timestamp
6          τ_arr[v] ← ∞
7      foreach e ← (j, i) ∈ E↓[T] do
8          u ← ID of the vertex in V that corresponds to j
9          τ_arr^new ← τ_arr[u] + w[e]
10         update ← τ_arr^new < τ_arr[v]
11         τ_arr[v] ← τ_arr^new if update                    // conditional move
12         parent[v] ← parent[u] if update                   // conditional move
```

Implementation Details. While the topological ordering of the vertices improves the performance of the sweeps, it is inefficient for the public transit part of the query. Many public transit algorithms, such as RAPTOR or CSA, achieve a large part of their efficiency by keeping stop data consecutive in memory. One way to achieve this in multi-modal scenarios is to assign vertex IDs between 0 and $|\mathcal{S}| - 1$ to the stops, and IDs between $|\mathcal{S}|$ and $|\mathcal{V}| - 1$ to the remaining vertices. However, this conflicts with the topological order used for the RPHAST-like sweeps. Thus, we use different vertex orderings and IDs for the public transit data structures and the RPHAST data structures, translating between them whenever we switch between RPHAST and public transit searches. For the public transit data structures, we assign IDs from 0 to $|\mathcal{S}| - 1$ to the stops, such that the relative ordering of the stops in the topological order is preserved. This ensures that the two orders are as similar as possible, and that sweeping over one ID range still requires only a single sweep over the other.

Detailed pseudocode for one of the three sweeps (line 6 from Algorithm 1) is given in Algorithm 2. The translation between vertex IDs used within the target-selected downward graph $G^{\downarrow}[\mathcal{T}]$ and general vertex IDs can be seen in lines 3 and 8. Another important observation is that parent pointers (required for journey unpacking) and arrival times are updated frequently in the inner loop, but only if an earlier arrival time has been found. It is crucial for the performance of the sweep to avoid branching operations within this inner loop. We therefore use conditional move operations to update the arrival time and parent pointer branchlessly in lines 11 and 12. Finally, we use timestamping in order to avoid initializing the arrival times for all vertices before each sweep. Since vertices are processed in topological order during the sweep, the timestamps of tail vertices of incoming edges do not need to be checked in the inner loop. Thus, timestamps are only checked in line 4.

3.2 Optimizing Number of Trips

We proceed with describing how our approach for computing one-to-many journeys can be extended to find a Pareto set of journeys (optimizing arrival time and number of trips) for every target. Since the maximum number of trips required by any Pareto-optimal journey is usually quite low, it is feasible to simply perform the final upward and downward sweep of our algorithm once for every possible number of trips. Furthermore, we can apply an optimization that was originally proposed for speeding up multiple PHAST searches from different source vertices [13]: Given a fixed parameter k, we no longer explore the final

Table 1 Sizes of the used public transit networks and the number of ULTRA shortcuts.

Network	Stops	Routes	Trips	Stop events	Vertices	Edges	Shortcuts
Switzerland	25 125	13 785	350 006	4 686 865	603 691	1 853 260	135 655
Germany	244 055	231 089	2 387 297	48 495 169	6 872 105	21 372 360	2 077 374

transfer for journeys using between 0 and $k-1$ trips with k separate upward and downward sweeps, but instead perform one upward and downward sweep which update all k arrival time values at once. Note that k must be a fixed value, since the sweeps are only efficient if the arrival times are stored consecutively in arrays of fixed size k. Journeys using k or more trips are not handled by this grouped sweep. However, we observe that only a few stops are reached by Pareto-optimal journeys that require a high number of trips. Propagating such journeys via a PHAST sweep, which always explores the entire graph, will be wasteful, since the arrival times of most vertices will not be improved by such vertices. Thus, for journeys using k or more trips, we switch to Dijkstra searches on a contracted transfer graph which contains all stops and targets, in a similar manner to MCR [11]. Similarly to the sweeps, the Dijkstra searches use timestamps to initialize only the labels of visited vertices. However, when the label of a vertex is initialized, we do not set its arrival time to ∞, but to the best arrival time found during the grouped sweeps. This ensures that journeys that are dominated by journeys with fewer trips get pruned early on.

4 Experiments

All algorithms were implemented in C++17 compiled with GCC version 8.2.1 and optimization flag -O3. All experiments were conducted on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.5 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache. Unless otherwise noted, all experiments were performed on a single core.

Networks. We evaluated our algorithms on the networks of Switzerland and Germany, which were previously used to evaluate ULTRA [8]. An overview of the networks is given in Table 1. The Switzerland network represents the timetable of two successive business days (May 30–31, 2017) and was extracted from a publicly available GTFS feed[1]. The Germany network is based on data from bahn.de and comprises two successive identical days taken from the Winter 2011/2012 timetable. In both cases, parts of the network that lie outside of the country borders were removed. The transfer graphs represent the road networks of Switzerland and Germany, including pedestrian zones and stairs. The data was obtained from OpenStreetMap[2]. Vertices with degree one and two were contracted unless they coincided with stops. We chose walking as a transfer mode, assuming a constant speed of 4.5 km/h. The ULTRA shortcuts were computed using the same settings as in the original ULTRA publication. The transfer graph was contracted up to an average vertex degree of 14 for Switzerland and 20 for Germany. The shortcut computation was performed in parallel on all 16 cores with a witness limit of 15 minutes. Together, the transfer graph contraction and the shortcut computation took 9:52 minutes for Switzerland and 9:00:12 hours for Germany. The number of shortcuts is reported in Table 1.

[1] http://gtfs.geops.ch/
[2] http://download.geofabrik.de/

Baseline Algorithms. Since no multi-modal algorithms which support one-to-many queries have yet been proposed, we created baseline algorithms for comparison by adapting the ideas of MCR to a scenario with multiple target vertices. MCR alternates between the route scanning phases of RAPTOR and Dijkstra searches on a partially contracted *core graph*, which is obtained via a CH computation on G that is not allowed to contract stops. Once the average vertex degree of the remaining graph reaches a certain threshold, the computation is stopped and the remaining graph is used as the core graph. Initial and final transfers are handled by running forward and backward searches on the partially constructed upward and downward graph, followed by Dijkstra searches in the core graph. An analogous multi-modal variant of CSA called MCSA, which alternates between connection scans and Dijkstra searches, was introduced in [8] to evaluate ULTRA.

When adapting MCR and MCSA to a one-to-many scenario, the forward search can be run unchanged, but the backward search is no longer feasible. Instead, we modify the computation of the core graph such that vertices in $\mathcal{S} \cup \mathcal{T}$ may not be contracted, rather than just stops. The backward search then becomes unnecessary, since the Dijkstra searches in the core graph already reach all targets. For our experiments, we contracted up to an average vertex degree of 14, except for very large target sets with $|\mathcal{T}| \geq 4|\mathcal{V}|$, where we used a vertex degree of 10 instead.

Target Sets. For our experiments, we considered three types of target sets: all vertices, all stops, and randomly generated target sets. For the randomly generated target sets, we followed the approach from [14]: We randomly picked a center vertex $c \in \mathcal{V}$ and then ran a Dijkstra search from c to find a *ball* $\mathcal{B} \subseteq \mathcal{V}$ consisting of the $|\mathcal{B}|$ nearest neighbors of c. From that ball, we then picked target vertices at random. We evaluated our algorithms for different combinations of ball size $|\mathcal{B}|$ and target set size $|\mathcal{T}|$, to study the impact of both the number of targets and the distribution of the targets in the graph.

4.1 UP-CSA

For the earliest arrival problem, we implemented UP-CSA, a combination of ULTRA-PHAST and CSA, and compared it to our one-to-many adaptation of MCSA.

Contraction Order. In Figure 1 (left), we evaluate the impact of the tuning parameters f_t and f_s on the performance of the three sweeps performed by ULTRA-PHAST: the downward sweeps in $G^{\downarrow}[\mathcal{S}]$ and $G^{\downarrow}[\mathcal{T}]$, and the upward sweep in $G^{\uparrow}[\mathcal{S}]$. The contraction of stops and targets was prohibited until $f_t|\mathcal{S} \cup \mathcal{T}|$ vertices were left, while stops were further left uncontracted until $f_s|\mathcal{S}|$ vertices remained. We observe that delaying the target contraction can improve the target-related sweep by up to a factor of 2, without significantly impacting the stop-related sweeps. The decrease in stop-related sweep times for $f_s = 10.0$ and $f_t < 3.0$ is explained by the fact that $f_t|\mathcal{S} \cup \mathcal{T}|$ becomes smaller than $f_s|\mathcal{S}|$, and thus stops remain uncontracted for longer than indicated by f_s. Delaying the contraction of stops slightly increases the running time of the sweep in $G^{\downarrow}[\mathcal{T}]$, but this is offset by the significant performance gains for the stop-related sweeps. While the sweeps performed best overall for $f_t = 1.5$ and $f_s = 1.5$, we observed that very low values for f_t negatively impacted the performance of the connection scanning phase due to an unfavorable stop order. Hence, we used $f_t = 2.0$ and $f_s = 1.5$ for all following experiments involving ball target sets. The CH computation time for this configuration was 2:53 minutes, approximately twice as long as a CH computation without delayed contraction.

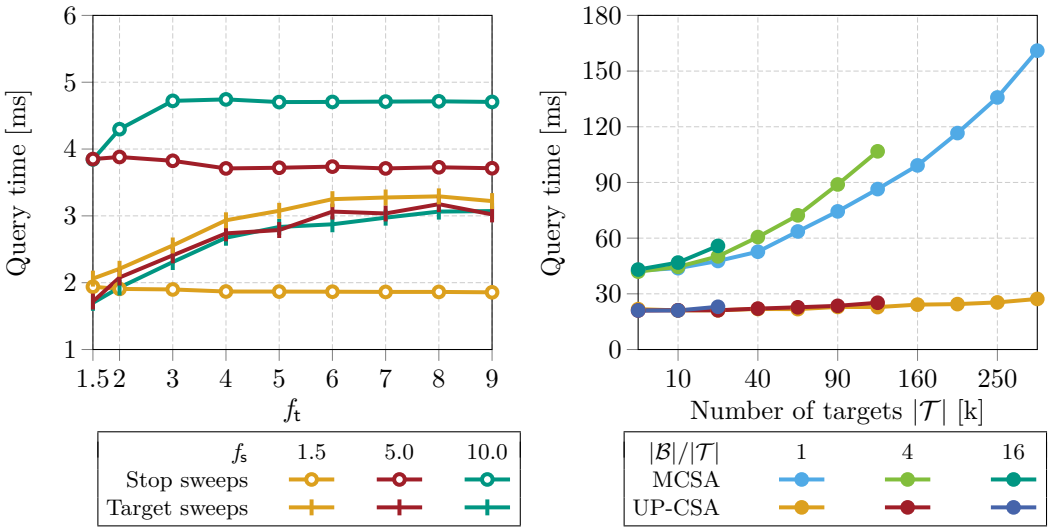


Figure 1 Impact of delayed contraction (left) and number of targets (right), measured on the Switzerland network. All running times are averaged over 1 000 queries each on 10 randomly chosen ball target sets. *Left:* Performance of the three ULTRA-PHAST sweeps depending on f_t and f_s, for ball target sets with $|\mathcal{T}| = 2^{16}$ and $|\mathcal{B}| = 2^{18}$. *Right:* Performance of MCSA and UP-CSA for different values of $|\mathcal{T}|$ and $|\mathcal{B}|$. Configurations with $|\mathcal{B}| > |\mathcal{V}|$ were omitted.

Target Set Size. The impact of target set size and distribution on the performance of MCSA and UP-CSA is measured in Figure 1 (right). For both algorithms, the exploration of transfers becomes more costly as the target set, and thus the size of the search graphs, increases. However, the effect is much more pronounced for MCSA, where the Dijkstra searches eventually take up a majority of the running time. By contrast, UP-CSA scales much better, with only a 30% increase in running time between the fastest and slowest configuration. This is because the portion of the overall running time spent on exploring transfers is much smaller than in MCSA. Increasing the ball size causes the stop and target selection to become less effective, as the targets are spread over a wider area of the graph. However, this only has a small effect on the overall performance of UP-CSA.

Detailed Performance. Table 2 gives a detailed overview of the performance of MCSA and UP-CSA for three types of target sets: all stops, all vertices, and a ball target set of moderate size. For the ball target sets, we used contraction delay factors of $f_t = 2.0$ and $f_s = 1.5$. For the other two sets, where delaying the contraction of targets is pointless, we achieved the best performance with $f_t = 1.5$. The preprocessing time for UP-CSA is naturally much larger than for MCSA, which only requires a contracted transfer graph. The vast majority (around 90%) of the preprocessing time for ULTRA-PHAST is due to the computation of ULTRA shortcuts. Most of the remainder is taken up by the computation of the stop- and target-delayed CH (between 30 and 50 minutes on Germany), while reordering the vertices and performing the stop and target selection only takes about 30 seconds on Germany. In terms of space consumption, both algorithms are lightweight: MCSA requires a core graph and a CH, which are similar in size to the original graph. ULTRA-PHAST requires the set of shortcuts and the three sweep graphs $G^{\downarrow}[\mathcal{S}]$, $G^{\uparrow}[\mathcal{S}]$ and $G^{\downarrow}[\mathcal{T}]$. The size of the latter is listed in Table 2, while the size of the former two can be inferred from the $\mathcal{T} = \mathcal{S}$ configuration, in which case all three graphs are of nearly identical size. On the smaller target sets, UP-CSA

Table 2 Detailed performance of MCSA and UP-CSA for three types of target sets: all vertices, all stops, and vertices randomly chosen from a ball. For the ball configuration, 10 target sets were randomly generated with $|\mathcal{T}| = 2^{14}$ for Switzerland, $|\mathcal{T}| = 2^{17}$ for Germany, and $|\mathcal{B}|/|\mathcal{T}| = 2$ for both networks. Running times are averaged over 10 000 random queries, which were distributed evenly among the 10 target sets for the ball configuration. Due to time constraints, only 1 000 queries were performed on Germany for $\mathcal{T} = \mathcal{V}$. Query times are divided into phases: initialization (including initial transfers), connection scan, final upward sweep, and final downward sweep.

Network	Targets	Algorithm	Preprocessing			Query time [ms]				
			Time [h]	$\lvert\mathcal{V}^{\downarrow}[\mathcal{T}]\rvert$	$\lvert\mathcal{E}^{\downarrow}[\mathcal{T}]\rvert$	Init	Scan	Up	Down	Total
Switzerland	Vertices	MCSA	00:01:19	–	–	74.7	133.5	–	–	208.2
		UP-CSA	00:11:27	603 691	2 360 885	0.9	18.3	0.9	9.1	29.2
	Stops	MCSA	–	–	–	8.1	34.2	–	–	42.3
		UP-CSA	00:11:27	37 669	284 328	0.8	18.0	0.9	0.7	20.4
	Ball	MCSA	00:01:54	–	–	11.0	36.5	–	–	47.5
		UP-CSA	00:12:24	20 031	153 867	1.0	20.6	1.0	0.5	23.2
Germany	Vertices	MCSA	–	–	–	1 500.7	2 831.1	–	–	4 331.8
		UP-CSA	09:30:31	6 872 105	27 716 664	10.8	407.5	13.4	174.0	605.8
	Stops	MCSA	00:22:54	–	–	115.4	655.7	–	–	771.1
		UP-CSA	09:30:25	365 987	3 546 112	10.3	389.6	14.3	7.9	422.0
	Ball	MCSA	00:19:20	–	–	139.3	667.7	–	–	807.0
		UP-CSA	09:50:12	148 398	1 228 965	11.8	380.0	14.5	4.8	411.1

is about twice as fast as MCSA. Roughly 90% of the overall running time is taken up by the connection scanning phase, indicating that the performance is close to the optimum that can be achieved with CSA. For the more challenging scenario where all vertices are targets, we achieve a speedup of slightly more than 7. Here, the main optimization of MCSA, which is to contract the transfer graph, is no longer applicable. By substituting the Dijkstra searches with memory-efficient sweeps, UP-CSA reduces the time that is spent exploring transfers by more than a factor of 20, bringing it down to about a third of the overall running time.

We also evaluated how the RPHAST downward sweep for the initial transfers compares to a Bucket-CH search, which is used by the original ULTRA algorithm: On Switzerland, a Bucket-CH search takes 1.6 ms compared to 0.8 ms for a sweep. On Germany, it takes 36.7 ms compared to 8.9 ms. This is more than both Bucket-CH searches performed by ULTRA combined, which demonstrates that the efficiency of Bucket-CH for ULTRA is only due to effective target pruning. In a one-to-many scenario, RPHAST is clearly preferable.

4.2 UP-RAPTOR

For the Pareto optimization problem, we implemented UP-RAPTOR, a combination of ULTRA-PHAST and RAPTOR, and compared it to one-to-many MCR.

Sweep Grouping. To determine the best choice for the number of grouped sweeps k, we evaluated random queries on Switzerland and Germany, using $\mathcal{S}$ as the target set. On Switzerland, we achieved the best performance for $k = 6$, with 5.0 ms for the grouped sweeps and 0.6 ms for the remaining Dijkstra searches, yielding 5.6 ms for the final transfers altogether. For $k = 8$, the time for the Dijkstra searches became negligible, but at the cost of increasing the sweep time to 6.4 ms. Conversely, choosing $k = 4$ increased the Dijkstra

Table 3 Detailed performance of MCR and UP-RAPTOR, using the same configurations as in Table 2. Query times are divided into phases: initialization (including initial transfers), route collection, route scan, relaxing intermediate transfers, and final transfers (upward and downward sweep for grouped rounds, Dijkstra search for the remainder).

Network	Targets	Algorithm	Time [ms]					
			Init	Collect	Scan	Inter	Final	Total
Switzerland	Vertices	MCR	94.3	24.5	15.7	354.6	–	492.9
		UP-RAPTOR	1.6	10.1	16.1	4.9	42.2	74.9
	Stops	MCR	37.0	18.6	20.9	31.6	–	109.7
		UP-RAPTOR	1.5	7.5	13.0	4.3	5.5	31.7
Germany	Vertices	MCR	1 959.4	690.0	298.4	8 099.3	–	11 177.7
		UP-RAPTOR	18.9	321.3	270.1	90.1	812.9	1 513.4
	Stops	MCR	480.4	350.8	529.7	552.7	–	1 919.4
		UP-RAPTOR	18.9	300.5	267.7	96.1	101.2	784.5

search time to 6.0 ms. On the Germany network, $k = 8$ performed slightly better than $k = 6$, with 102.8 ms and 109.5 ms for the final transfers, respectively. The different results for the two networks can be explained by the fact that journeys are more likely to require a high number of trips on larger networks.

Detailed Performance. A detailed overview of the performance of MCR and UP-RAPTOR is given in Table 3. The experimental setup and the preprocessing phase are identical to Table 2. For the number of grouped sweeps, we chose $k = 6$ for Switzerland and $k = 8$ for Germany, as suggested by the experiments reported above. RAPTOR operates in rounds, with round i computing all optimal journeys using i trips. Each round consists of three phases: collecting routes reached in the previous round, scanning those routes, and relaxing intermediate transfers. Additionally, there is an initialization phase before the first round that includes the exploration of initial transfers. UP-RAPTOR adds a fourth phase to each round which explores the final transfers. This phase is skipped until round $k - 1$, where a grouped upward and downward sweep are performed for rounds 0 to $k - 1$. In all later rounds, final transfers are explored with a Dijkstra search on the same core graph that is also used by MCSA, MCR and the ULTRA shortcut computation.

We observe speedups between 2.4 and 3.5 for $\mathcal{T} = \mathcal{S}$ and between 6.6 and 7.4 for $\mathcal{T} = \mathcal{V}$. The share of the transfer exploration in the overall running time is larger than for UP-CSA, as RAPTOR explores more transfers in general due to optimizing two criteria. Exploring the final transfers takes 3-4 times as long as for UP-CSA, but is here done across the 8 or more rounds of a typical RAPTOR query. On the set of stops, UP-RAPTOR achieves a better speedup than UP-CSA. This is mainly for two reasons: At the start of each new round, MCR copies the arrival times of all vertices from the previous round. By contrast, UP-RAPTOR only copies arrival times from previous rounds during the Dijkstra searches, and only when a vertex is actually visited. The other reason is that UP-RAPTOR explores fewer intermediate transfers due to using ULTRA shortcuts. As a result, fewer stops are visited in the transfer phases and therefore fewer routes are collected and scanned in the following phases. This reduction in the search space has a stronger effect on RAPTOR than on CSA, which always iterates across all connections, regardless of whether they are reachable.

5 Conclusion

In this work, we adapted ULTRA for one-to-many and one-to-all query scenarios. Since ULTRA explores initial and final transfer with a bidirectional search, which is not feasible for a large number of target vertices, we developed a new final transfer search that adapts ideas from RPHAST. We replaced the upward CH search of RPHAST with an efficient upward sweep, since all stops that are reachable via a trip act as potential source vertices for the final transfer search. We also extended our approach to solve the Pareto optimization problem, where multiple final transfer searches are required. The resulting algorithmic framework, ULTRA-PHAST, yields the first algorithms specifically designed for one-to-all and one-to-many searches in multi-modal networks. We evaluated ULTRA-PHAST versions of CSA and RAPTOR on the networks of Switzerland and Germany. For small and moderately sized target sets, the share of the transfer exploration in the overall running time could be reduced to 10-20%, with the rest being equivalent to an uni-modal public transit query. For large target sets, we achieved a speedup of 7 compared to naive adaptations of MCR and MCSA.

For future work, we would like to adapt our approach to extended one-to-many scenarios, such as point-of-interest queries, isochrones and traffic assignments. Some of these scenarios require ULTRA-PHAST to be combined with profile search. For the Pareto optimization problem, the combined sweeps could be sped up further by using vector instructions, such as SSE or AVX. Finally, ULTRA-PHAST could serve as an ingredient in a preprocessing technique that enables even faster multi-modal one-to-one queries than ULTRA.

References

1 Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *International Symposium on Experimental Algorithms*, pages 230–241. Springer, 2011.

2 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 *European Symposium on Algorithms*, pages 290–301. Springer, 2010.

3 Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F Werneck. Route Planning in Transportation Networks. In *Algorithm Engineering*, pages 19–80. Springer, 2016.

4 Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable Transfer Patterns. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 15–29, 2016.

5 Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*, pages 7:1–7:18, 2016.

6 Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. Fast Exact Computation of Isochrones in Road Networks. In *International Symposium on Experimental Algorithms*, pages 17–32. Springer, 2016.

7 Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. Fast Exact Computation of Isocontours in Road Networks. *ACM Journal of Experimental Algorithmics*, 24(1), 2019.

8 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. In *27th Annual European Symposium on Algorithms (ESA 2019)*, pages 14:1–14:16, 2019.

9 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected.

In *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, 2009.

10 Lars Briem, H. Sebastian Buck, Holger Ebhart, Nicolai Mallig, Ben Strasser, Peter Vortisch, Dorothea Wagner, and Tobias Zündorf. Efficient Traffic Assignment for Public Transit Networks. In *16th Symposium on Experimental Algorithms (SEA 2017)*, 2017.

11 Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato Werneck. Computing Multimodal Journeys in Practice. In *International Symposium on Experimental Algorithms*, pages 260–271. Springer, 2013.

12 Daniel Delling, Andrew Goldberg, Thomas Pajor, and Renato Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Springer, 2011.

13 Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

14 Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Faster Batched Shortest Paths in Road Networks. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2011)*, pages 52–63, 2011.

15 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating Multi-modal Route Planning by Access-Nodes. In *Algorithms – ESA 2009*, pages 587–598, 2009.

16 Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based Public Transit Routing. *Transportation Science*, 49(3):591–604, 2014.

17 Daniel Delling and Renato Werneck. Customizable Point-of-Interest Queries in Road Networks. In *IEEE Transactions on Knowledge and Data Engineering*, pages 500–503, 2013.

18 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *International Symposium on Experimental Algorithms*, pages 43–54. Springer, 2013.

19 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithmics*, pages 1.7:1–1.7:56, 2018.

20 Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multimodal Route Planning. *ACM Journal of Experimental Algorithmics*, pages 3.2:1–3.2:19, 2015.

21 Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

22 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, pages 347–361. Springer, 2008.

23 Alexandros Efentakis and Dieter Pfoser. GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem. In *Algorithms – ESA 2014*, pages 358–370. Springer, 2014.

24 Johann Gamper, Michael Böhlen, Willi Cometti, and Markus Innerebner. Defining Isochrones in Multimodal Spatial Networks. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, pages 2381–2384, 2011.

25 Johann Gamper, Michael Böhlen, and Markus Innerebner. Scalable Computation of Isochrones with Network Expiration. In *Scientific and Statistical Database Management*, pages 526–543. Springer, 2012.

26 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, pages 319–333. Springer, 2008.

27 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

28 Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. Multimodal Dynamic Journey-Planning. *Algorithms*, 12(10):213, 2019.

29 Moritz Hilger, Ekkehard Köhler, Rolf Möhring, and Heiko Schilling. *Fast Point-to-Point Shortest Path Computations with Arc-Flags*, pages 41–72. American Mathematical Society, 2009.

30 Jan Hrnčíř and Michal Jakob. Generalised Time-Dependent Graphs for Fully Multimodal Journey Planning. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 2138–2145. IEEE, 2013.

31 Dominik Kirchler. *Efficient Routing on Multi-Modal Transportation Networks*. PhD thesis, Ecole Polytechnique X, 2013.

32 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.

33 Nikolaus Krismer, Doris Silbernagl, Günther Specht, and Johann Gamper. Computing Isochrones in Multimodal Spatial Networks Using Tile Regions. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 2017.

34 Duc-Minh Phan and Laurent Viennot. Fast Public Transit Routing with Unrestricted Walking through Hub Labeling. In *Proceedings of the Special Event on Analysis of Experimental Algorithms (SEA^2)*. Springer, 2019.

35 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

36 Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile Route Planning. In *Algorithms – ESA 2008*, pages 732–743. Springer, 2008.

37 Jonas Sauer. Faster Public Transit Routing with Unrestricted Walking. Master's thesis, Karlsruhe Institute of Technology, 2018.

38 Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. Efficient Computation of Multi-Modal Public Transit Traffic Assignments Using ULTRA. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 524–527, 2019.

39 Dorothea Wagner and Tobias Zündorf. Public Transit Routing with Unrestricted Walking. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, 2017.

40 Sascha Witt. Trip-Based Public Transit Routing. In *Algorithms – ESA 2015*, pages 1025–1036. Springer, 2015.