UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution

Moritz Baum
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
moritz.baum@kit.edu

Valentin Buchhold
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
buchhold@kit.edu

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

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

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

Abstract

We study a multi-modal route planning scenario consisting of a public transit network and a transfer graph representing a secondary transportation mode (e.g., walking or taxis). The objective is to compute all journeys that are Pareto-optimal with respect to arrival time and the number of required transfers. While various existing algorithms can efficiently compute optimal journeys in either a pure public transit network or a pure transfer graph, combining the two increases running times significantly. As a result, even walking between stops is typically limited by a maximal duration or distance, or by requiring the transfer graph to be transitively closed. To overcome these shortcomings, we propose a novel preprocessing technique called ULTRA (UnLimited TRAnsfers): Given a complete transfer graph (without any limitations, representing an arbitrary non-schedule-based mode of transportation), we compute a small number of transfer shortcuts that are provably sufficient for computing all Pareto-optimal journeys. We demonstrate the practicality of our approach by showing that these transfer shortcuts can be integrated into a variety of state-of-the-art public transit algorithms, establishing the ULTRA-Query algorithm family. Our extensive experimental evaluation shows that ULTRA is able to improve these algorithms from limited to unlimited transfers without sacrificing query speed, yielding the fastest known algorithms for multi-modal routing. This is true not just for walking, but also for other transfer modes such as cycling or driving.

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

Keywords and phrases Algorithms, Optimization, Route Planning, Public Transportation

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.14

Related Version A full version of the paper is available at `https://arxiv.org/abs/1906.04832`.

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

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

1 Introduction

Research on efficient route planning algorithms has seen remarkable advances in the past decades. For many types of transportation networks, queries can be solved in a few milliseconds, even on a continental scale [3]. However, combining schedule-based (i.e., public transit) and non-schedule-based (e.g., walking, cycling, driving) transportation modes and solving the resulting multi-modal routing problem is still a challenge [26]. In this work, we consider a multi-modal problem that augments public transit with a *transfer graph*, which represents an arbitrary non-schedule-based mode of transportation that can be used for transferring between public transit stops. Given a source and target vertex in the transfer graph and a departure time, we want to compute all Pareto-optimal journeys regarding travel time and number of used public transit trips.

Related Work. Most algorithms for public transit routing either impose technical restrictions on the included transfers or have only been evaluated on networks featuring very sparse transfer graphs. Algorithms that were only evaluated for limited transfers include the graph-based techniques in [23] and [18], frequency-based search [5], Transfer Patterns [2] and its accelerated version, Scalable Transfer Patterns [4], Public Transit Labeling [11], and SUBITO [9]. A common restriction, employed by CSA [14, 15], RAPTOR [13], and their corresponding speedup techniques ACSA [25, 15] and HypRAPTOR [12], is to require that the transfer graph is transitively closed. This eliminates the need to search within the transfer graph, as every possible destination can be reached with a single edge. To ensure a reasonably sized transfer graph, transfers are typically limited by a maximal duration (e.g., 15 minutes of walking) or distance before the transitive closure is computed. As shown in [26], choosing a higher limit for the maximal transfer duration increases the size of the resulting transitively closed graph significantly. A limit of only 20 minutes on the maximal transfer duration already leads to a graph that is unsuitable for practical applications. A special case is Trip-Based Routing [27], which precomputes transfers between pairs of trips. This precomputation involves enumerating all possible transfers and then using a limited set of pruning rules to omit some, but not all unnecessary transfers. Trip-Based Routing was only evaluated for transitively closed transfer graphs and likely has prohibitively high preprocessing times on unrestricted transfer graphs.

Using a restricted transfer graph is often justified with the argument that long transfers are rarely useful. However, experiments performed in [26, 22] show that unrestricted walking often significantly reduces the travel time of optimal journeys. This effect is likely even stronger for faster transportation modes, such as bicycle or car. The only algorithms that can handle unrestricted transfer graphs so far are multi-modal techniques such as MCR [10] and UCCH [16]. These techniques work by interleaving a public transit routing algorithm with Dijkstra's algorithm [17] on a contracted transfer network. Accordingly, they are fairly slow compared to pure public transit algorithms. Most recently HLRaptor and HLCSA [22] have been published. Here, RAPTOR/CSA are interleaved with Hub Labeling queries [1] instead of Dijkstra. While requiring more than an hour of preprocessing, this significantly improves query times. However, it is still not as efficient as pure public transit algorithms.

Our Contribution. Preliminary experiments [24] have shown that the impact of unrestricted transfers in Pareto-optimal journeys depends heavily on their position in the journey: *Initial transfers*, which connect the source to the first public transit vehicle, and *final transfers*, connecting the final vehicle to the target, are fairly common and often have a large impact on the travel time. In contrast, *intermediate transfers* between public transit trips are only occasionally relevant for optimal journeys. This suggests that the number of unique paths in the transfer graph that occur as intermediate transfers of a Pareto-optimal journey is small. Using this insight, we propose a new preprocessing technique called ULTRA (UnLimited TRAnsfers), which computes a set of shortcut edges representing these paths. The preprocessing step is carefully engineered to ensure that the number of shortcuts remains small. Combined with efficient one-to-many searches for the initial and final transfers, these shortcuts are sufficient to answer all queries in the network correctly. ULTRA shortcuts can be used without adjustment by any algorithm that previously required a transitively closed transfer graph. Our experiments show that this enables unrestricted multi-modal queries with roughly the same performance as restricted queries. In particular, ULTRA-CSA is the first efficient multi-modal variant of CSA. Source code for ULTRA and our experiments is available at `https://github.com/kit-algo/ULTRA`.

2 Preliminaries

In this section we establish the basic notation and terminology used in this work. Moreover, we introduce the RAPTOR and Bucket-CH algorithms, on which our work is founded.

Public Transit Network. A public transit network is a 4-tuple $(\mathcal{S}, \mathcal{T}, \mathcal{R}, G)$ consisting of a set of *stops* $\mathcal{S}$, a set of *trips* $\mathcal{T}$, a set of *routes* $\mathcal{R}$, and a directed, weighted *transfer graph* $G = (\mathcal{V}, \mathcal{E})$. Every stop in $\mathcal{S}$ defines a location in the network where passengers can board or disembark a vehicle (such as buses, trains, ferries, etc.). Furthermore, we associate with each stop $v \in \mathcal{S}$ a non-negative *departure buffer time* $\tau_{\text{buf}}(v)$, which defines the minimum amount of time that has to pass after arriving at the stop before a vehicle can be boarded. A trip $\mathrm{T} = \langle v_0, \dots, v_k \rangle \in \mathcal{T}$ is a sequence of at least two stops which are served consecutively by the same vehicle. For each stop v in the sequence, $\tau_{\text{arr}}(\mathrm{T}, v)$ denotes the *arrival time* of the vehicle at v, and $\tau_{\text{dep}}(\mathrm{T}, v)$ denotes its *departure time*. This, of course, implies that $\tau_{\text{arr}}(\mathrm{T}, v) \leq \tau_{\text{dep}}(\mathrm{T}, v)$ holds for every trip T and stop v. The i-th stop of a trip T is denoted as $\mathrm{T}[i]$. The set of routes $\mathcal{R}$ defines a partition of the trips such that two trips are part of the same route if they have the same stop sequence and do not overtake each other. A trip $\mathrm{T}_a \in \mathcal{T}$ overtakes the trip $\mathrm{T}_b \in \mathcal{T}$ if two stops $u, v \in \mathcal{S}$ exist such that T_a arrives at or departs from u before T_b and T_a arrives at or departs from v after T_b.

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* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. For each edge $e = (u, v) \in \mathcal{E}$ we define the *transfer time* $\tau_\theta(e)$ as the time required to transfer from u to v. The notion of transfer time carries over to paths $P = (v_1, \dots, v_k)$ in G, using the definition $\tau_\theta(P) := \sum_{i=1}^{k-1}((v_i, v_{i+1}))$. Unlike in restricted walking scenarios, we require no special properties for the transfer graph G. It does not need to be transitively closed, it may be strongly connected, and transfer times may represent walking, cycling, or some other non-schedule-based mode of travel.

Journeys. A *trip leg* T^{ij} is a subsequence of the trip T, representing a passenger boarding the trip T at the i-th stop and disembarking at the j-th stop. The departure time of T^{ij} is the departure time at the first stop of the trip leg, i.e., $\tau_{\text{dep}}(\mathrm{T}^{ij}) := \tau_{\text{dep}}(\mathrm{T}, \mathrm{T}[i])$. Similarly, the arrival time is defined as $\tau_{\text{arr}}(\mathrm{T}^{ij}) := \tau_{\text{arr}}(\mathrm{T}, \mathrm{T}[j])$. An *intermediate transfer* ϑ between two trip legs T_a^{ij} and T_b^{mn} is a path in the transfer graph such that: (1) the path ϑ begins with the last stop of T_a^{ij}, (2) the path ends with the first stop of T_b^{mn}, and (3) the transfer time of the path is sufficient to reach T_b^{mn}. The transfer time is sufficient if after vacating T_a^{ij} and transferring to the departure stop of T_b^{mn}, there is still sufficient buffer time to enter T_b^{mn}. We can express this formally as $\tau_{\mathrm{arr}}(\mathrm{T}_a^{ij}) + \tau_\theta(\vartheta) + \tau_{\mathrm{buf}}(\mathrm{T}_b^{mn}[m]) \leq \tau_{\mathrm{dep}}(\mathrm{T}_b^{mn})$. An *initial transfer* ϑ before a trip leg T^{ij} is a path in G from the source s to the first stop of T^{ij}. Correspondingly, a *final transfer* ϑ after a trip leg T^{ij} is a path in G from the last stop of T^{ij} to the target t. We use the term *transfer* on its own to denote the union of all transfer types, or if the actual type of the transfer can be deduced from context. We define a *journey* $J = \langle \vartheta_0, \mathrm{T}_0^{ij}, \ldots, \mathrm{T}_{k-1}^{mn}, \vartheta_k \rangle$ as an alternating sequence of transfers and trip legs. Note that some or all of the transfers may be empty. The departure time of the journey is defined as $\tau_{\mathrm{dep}}(J) := \tau_{\mathrm{dep}}(\mathrm{T}_0^{ij}) - \tau_{\mathrm{buf}}(\mathrm{T}_0[i]) - \tau_\theta(\vartheta_0)$ and the arrival time as $\tau_{\mathrm{arr}}(J) := \tau_{\mathrm{arr}}(\mathrm{T}_{k-1}^{mn}) + \tau_\theta(\vartheta_k)$. The number of trips used by the journey is k. A journey J (weakly) *dominates* a journey J' if $\tau_{\mathrm{dep}}(J) \geq \tau_{\mathrm{dep}}(J')$, $\tau_{\mathrm{arr}}(J) \leq \tau_{\mathrm{arr}}(J')$, and J does not use more trips than J'. For strict domination, at least one criterion must be strictly better. A journey J is called *Pareto-optimal* if no other journey exists that dominates J.

In our journey definition, the departure buffer time at a stop models the time required to reach the right platform and board a trip, regardless of how the stop was reached. Many other works on public transit routing instead use a *minimum transfer time*, which only needs to be observed if the stop was reached directly via a trip instead of a transfer. This is reasonable for settings with direct transfers between stops, where the buffer time can simply be included in the transfer time. When allowing arbitrary transfers, however, it can lead to inconsistencies. For instance, given a stop with minimum transfer time τ, if a path starting and ending at this stop with a transfer time less than τ exists, then taking that path would allow passengers to circumvent the minimum transfer time.

Algorithms. Since our algorithm is strongly influenced by the RAPTOR algorithm family, we now introduce the basic concepts of these algorithms. The RAPTOR [13] algorithm can be used to solve one-to-one and one-to-many queries on a public transit network with limited transfers. The algorithm operates in *rounds*, where the i-th round finds all journeys using exactly i trips. For this, each round extends journeys found in the previous round by one trip, which can be done via a single scan of all routes in the network. An extension of this algorithm for multi-modal scenarios with unlimited transfers is MCR [10]. In this algorithm the RAPTOR rounds are alternated with Dijkstra's algorithm on a contracted transfer graph, in order to propagate arrival times through the transfer graph. Another extension, rRAPTOR [13], can be used to answer *range queries*, which ask for all Pareto-optimal journeys that depart within a given time interval. The rRAPTOR algorithm operates in *iterations*, where every iteration handles a possible departure time using the basic RAPTOR algorithm. The possible departure times are handled in descending order, and the data structures used by RAPTOR are not cleared in between iterations. As a result, journeys found by the current iteration are implicitly pruned by journeys that depart later and neither arrive later nor have more trips. This property of the rRAPTOR algorithm is called *self-pruning*.

Besides public transit routing algorithms, we also require efficient one-to-many algorithms for the transfer graph. Especially the Bucket-CH [21, 19, 20] algorithm is useful for our purposes. This algorithm is based on Contraction Hierarchies (CH) [19, 20] and operates in three phases. First the CH is computed, requiring only the graph. Second, given the set of targets, a bucket containing distances to the targets is computed for every vertex. This is done by adding every target to the buckets of all vertices in its reverse CH search space. Finally, the distance from a source to all targets is computed by performing the forward part of a CH search. For each vertex v in the forward search space, the bucket is evaluated by combining the distance to v with the distance from v to the targets in the bucket.

3 Shortcut Computation

Our preprocessing technique aims at finding a small number of transfer shortcuts that are sufficient to answer every point-to-point query correctly. This is achieved if for every Pareto-optimal journey there exists a journey with the same departure time, arrival time, and number of trips that uses only the precomputed shortcuts to transfer between trips. Next, we present a high-level overview of the ULTRA preprocessing, followed by an in-depth description of important algorithmic details.

3.1 Overview

The basic idea of ULTRA is as follows. We enumerate all possible journeys that use exactly two trips and require neither an initial nor a final transfer. The transfers between the two trips of these journeys are then considered as *candidates* for shortcuts. For each of these *candidate journeys*, we check if there is another journey that dominates it. If this is the case, we can replace the candidate journey with the dominating journey without losing Pareto-optimality. Note that if the candidate journey is contained in a longer journey, then it still can be replaced without affecting the Pareto-optimality of the longer journey. We call such a dominating journey a *witness* since its existence proves that the candidate shortcut is not needed. Unlike the candidate journey, the witness journey can make use of the transfer graph before the first trip or after the second trip. If no witness is found, then the candidate shortcut is added to the resulting shortcut graph.

A naive implementation of this idea would be to first enumerate all candidate journeys and subsequently search for witnesses. However, this would be impractical due to the sheer number of possible journeys. We therefore propose to interweave the candidate enumeration and the witness search, with the goal of eliminating as many candidates as early as possible. Pseudocode for the result of these considerations is given by Algorithm 1. The algorithm resembles invoking rRAPTOR [13] once per stop, restricted to the first two rounds per iteration. Remember that the original rRAPTOR algorithm already answers one-to-all range queries. Restricting this algorithm to the first two rounds enables an efficient enumeration of candidate journeys. Moreover, many dominated candidates are eliminated early on, due to self-pruning. We will now continue with a detailed discussion of Algorithm 1, showing step by step what has changed in comparison to the original rRAPTOR and how this helps with computing the transfer shortcuts.

3.2 Implementation Details

A first important difference is due to the fact that rRAPTOR requires a transitively closed transfer graph. As we want to allow arbitrary transfer graphs, we replace the RAPTOR that is invoked in every iteration of rRAPTOR with MR-∞, the variant of MCR that optimizes arrival time and number of used trips. Because of this change, the relaxation of transfers in lines 8 and 11 is not done by relaxing outgoing edges of updated stops. Instead, Dijkstra's algorithm is performed in order to propagate arrival times found by the preceding route scanning step. Furthermore, MCR would also use Dijkstra's algorithm in order to collect all routes reachable from the source stop in line 6. In the context of rRAPTOR this leads to many redundant computations, as the source stop does not change between iterations. We therefore compute distances from the source stop to all other stops once in line 3, again using Dijkstra's algorithm. These distances can then be used in line 6.

Algorithm 1 ULTRA transfer shortcut computation.

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R}, G)$, with unrestricted transfer graph $G = (\mathcal{V}, \mathcal{E})$
Output: Shortcut graph $G' = (\mathcal{S}, \mathcal{E}')$

1 **for each** $s \in \mathcal{S}$ **do**
2   Clear all arrival labels and Dijkstra queues
3   $d(s, *) \leftarrow$ Compute distances from s to all stops in G using Dijkstra
4   $D \leftarrow$ Collect departure times of trips at s
5   **for each** $\tau_{dep} \in D$ in descending order **do** `// rRAPTOR iteration`
6   Collect routes reachable from s at τ_{dep} `// first RAPTOR round`
7   Scan routes
8   Relax transfers
9   Collect routes serving updated stops `// second RAPTOR round`
10   Scan routes
11     $C \leftarrow$ Relax transfers, thereby collecting unwitnessed candidates
12     $\mathcal{E}' \leftarrow \mathcal{E}' \cup C$

Departure Time Collection. In line 4, standard rRAPTOR would collect all departure events that are reachable from the source stop s. However, given a transfer graph without any restrictions, this could possibly be every departure event in the network. Since we are primarily interested in finding candidate journeys, which do not have initial transfers, we collect only those departure events which depart directly at the source stop s. However, in order to find witness journeys, we still need to explore initial transfers in line 6. A naive implementation would check for each stop v reachable from s and for each route containing the stop v whether a trip that was not scanned in a previous iteration can be reached given the departure time τ_{dep} at s.

A more efficient approach combines lines 4 and 6 into a single operation. For this, we first sort all departure triplets $(v, \tau_{\text{dep}}, r)$ of departure stop v, departure time τ_{dep}, and route r by their corresponding departure time at the source, $\tau_{\text{dep}} - \tau_{\text{buf}}(v) - d(s, v)$. Afterwards, we iterate through this sorted list in descending order of departure time. If the next triplet to be processed has a departure stop $v \neq s$, then its route is added to a set $\mathcal{R}'$. In the case that the next triplet actually has the source stop s as departure stop v, we proceed with lines 6 through 12. Now the routes that have to be collected in line 6 are exactly the routes in $\mathcal{R}'$. Thus we simply scan all routes in $\mathcal{R}'$ and then reset $\mathcal{R}' = \emptyset$ for the next iteration.

Limited Transfer Relaxation. Another part of ULTRA that differs from rRAPTOR is the final relaxation of transfers in line 11. This is the part of the algorithm where we actually determine the candidate journeys for which have not found a witness. As usual, relaxing the transfers is done by Dijkstra's algorithm, initialized with the arrival times from the preceding route scanning step. Whenever a stop is settled during this execution of Dijkstra's algorithm, we look at the corresponding journey and check whether it is a candidate journey, i.e., does not require initial or final transfers. If so, we know that there is no witness journey dominating this candidate, because otherwise the search would have reached the stop via this witness journey instead. Thus, we extract the intermediate transfer of the found candidate journey and add it as an edge to the shortcut graph.

We further increase the practical performance of our algorithm by adding a stopping criterion to the final transfer relaxation in line 11. For this purpose, we count the number of stops which were newly reached via a candidate journey in the preceding route scanning step. Whenever such a stop is settled in line 11, we decrease our counter. Once the counter reaches zero, we can stop settling further vertices as we know that no more candidates can be found in this iteration. We can apply a similar stopping criterion to the intermediate transfer relaxation in line 8. In this case, we count the stops which were reached via a route directly from s, without an initial transfer, since only these stops can later become part of a candidate journey. As in line 11, we can stop settling vertices as soon as no such stops are left in the Dijkstra queue. This does not affect the correctness of the algorithm, as we still process all candidates. However, it might cause some witnesses to be pruned and thus lead to superfluous shortcuts in the result. To counteract this, we take the arrival time τ_{arr} of the last stop representing a candidate that is settled. Instead of stopping the transfer relaxation immediately, we continue until the queue head has an arrival time greater than $\tau_{\mathrm{arr}} + \bar{\tau}$ for some parameter $\bar{\tau}$ (which we call *witness limit*). With these changes, the only remaining part of the algorithm that performs an unlimited search on the transfer graph is the initial transfer relaxation in line 3, which is only done once per source stop.

The success of our pruning rule for the transfer relaxation in lines 8 and 11 depends on the presence of candidate journeys in the Dijkstra queues. Fewer candidate journeys could therefore lead to an earlier application of the pruning rule. We exploit this by further restricting the notion of candidate journeys. As before, a candidate journey must not contain any initial or final transfers. In addition, we now require that the intermediate transfer of a candidate journey is not contained in the set of already computed transfer shortcuts.

Cyclic Witnessing. Since witnesses are only required to dominate candidate journeys weakly, there may be journeys J, J' that dominate each other. If J has an initial transfer of length > 0, then J without the initial transfer is not dominated by J' extended by the reverse initial transfer. Therefore, the shortcut required by J will be added. Thus, cyclic domination is only problematic between journeys with initial transfers of length 0. We prevent this by temporarily contracting groups of stops with transfer distance 0 during the preprocessing.

Transfer Graph Contraction. As shown for MCR [10], the transfer relaxation is often the bottleneck of multi-modal routing algorithms. Since ULTRA only needs to compute journeys between stops, rather than arbitrary vertices of the transfer graph, only transfers that start and end at stops are relevant. Therefore, any overlay graph that preserves the distances between all stops can be used instead of the transfer graph in our preprocessing algorithm. An easy way of obtaining such an overlay graph is to construct a partial CH that only contracts vertices that do not correspond to stops of the public transit network. This, of course, leads to a suboptimal contraction order and thus makes it infeasible to contract all vertices that are not stops. As done in many other algorithms [6, 16, 10, 8, 7], we therefore stop the contraction once the uncontracted *core* graph surpasses a certain average vertex degree.

Parallelization. Finally, we observe that ULTRA allows for trivial parallelization. Our algorithm searches for candidate journeys once for every possible source stop (line 1 of Algorithm 1). As these searches are mostly independent of each other, we can distribute them to parallel threads and combine the results in a final sequential step. Only the usage of the restricted candidate notion introduces a dependence between the searches for different source stops. As this is only a heuristic performance optimization, we simply relax the notion of candidate journeys again, only requiring that no shortcut representing the intermediate transfer has been found by the same thread yet.

Algorithm 2 Query algorithm, using transfer shortcuts computed by ULTRA.

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R}, G)$, shortcut graph $G' = (\mathcal{S}, \mathcal{E})$,
Bucket-CH of G, source vertex s, departure time τ_{dep}, and target vertex t
Output: All Pareto-optimal journeys from s to t for departure time τ_{dep}

$d(s, *) \leftarrow$ Run Bucket-CH query from s
$d(*, t) \leftarrow$ Run reverse Bucket-CH query from t
$G'' \leftarrow G'$
for each $v \in \mathcal{S}$ **do**
  Add edge (s, v) to G'' with travel time $d(s, v)$
  Add edge (v, t) to G'' with travel time $d(v, t)$
Run black box public transit algorithm on $(\mathcal{S} \cup \{s, t\}, \mathcal{T}, \mathcal{R}, G'')$

3.3 Proof of Correctness

Before continuing with the query algorithms, we want to justify that ULTRA computes a shortcut graph that is sufficient to answer all queries correctly. For contradiction we assume that a journey $J = \langle \vartheta_0, \mathrm{T}_0^{ij}, \ldots, \mathrm{T}_{k-1}^{mn}, \vartheta_k \rangle$ exists that requires an intermediate transfer not contained in the shortcut graph and cannot be replaced with a journey of equal travel time and number of trips that solely uses transfers from the shortcut graph. In this case, the journey J must contain at least two trips, since otherwise it would not contain any intermediate transfers. Since the journey contains two or more trips, it can be disassembled into candidate journeys $\langle \mathrm{T}_0^{ij}, \vartheta_1, \mathrm{T}_1^{gh} \rangle, \langle \mathrm{T}_1^{gh}, \vartheta_2, \mathrm{T}_2^{pq} \rangle, \ldots, \langle \mathrm{T}_{k-2}^{uv}, \vartheta_{k-1}, \mathrm{T}_{k-1}^{mn} \rangle$. As J requires a transfer not contained in the shortcut graph, at least one of these candidates must also contain a transfer not contained in the shortcut graph. Let $J^c = \langle \mathrm{T}_x^{gh}, \vartheta_{x+1}, \mathrm{T}_{x+1}^{pq} \rangle$ be such a candidate journey. Since the main loop of ULTRA is executed for every stop in the network, it was also executed for the source stop $\mathrm{T}_x[g]$ of this candidate journey. Derived from the correctness of rRAPTOR, we know that for a given source stop our algorithm computes Pareto-optimal arrival labels for all stops reachable with two trips or less. Thus we also reached the target stop $\mathrm{T}_{x+1}[q]$ of the candidate journey. The journey J' corresponding to the target's arrival label is in this case either the candidate journey or a journey that dominates the candidate journey. In the first case, we have added the transfer ϑ_{x+1} of the candidate journey to the shortcut graph. In the second case, the candidate journey J^c can be replaced by the journey J' corresponding to the target's arrival label, leading to a journey that is not worse than the original journey and does not require the missing transfer. Therefore both cases contradict our assumption.

4 Query Algorithms

The shortcuts obtained by ULTRA can in principle be combined with any public transit query algorithm that normally requires a transitively closed transfer graph, such as RAPTOR [13], CSA [14, 15], or Trip-Based Routing [27]. The basic idea of the query algorithm is to simply use one of the above algorithms together with our precomputed shortcut graph instead of the original transfer graph. However, our shortcut graph only represents transfers between two trips, and does not provide any information for transferring from the source to the first trip or from the last trip to the target. In this section we describe how the public transit algorithms can be modified in order to handle initial and final transfers efficiently.

Basic Query Algorithm. Our approach is based on the observation that for initial and final transfers one endpoint of the transfer is fixed. All initial transfers start at the source vertex, and all final transfers end at the target vertex. Therefore, we can use two additional one-to-many queries (one of them performed in reverse) to cover initial and final transfers. These queries have to be performed on the original transfer graph, where they compute the distances from the source to all stops and from all stops to the target. While any one-to-many algorithm might be used to perform this task, we decided to use Bucket-CH, as it is one of the fastest known one-to-many algorithms and allows for optimization of local queries. Pseudocode for the resulting query algorithm using Bucket-CH and our transfer shortcuts is shown in Algorithm 2.

Our algorithm begins with performing the two Bucket-CH queries from the source and target stop in lines 1 and 2. Afterwards a temporary copy of the shortcut graph G'' is initialized. In lines 5 and 6, this temporary graph is complemented with edges from the source to all other stops and edges from all stops to the target, using the distances obtained from the Bucket-CH queries. Finally, a public transit algorithm is invoked as a black box on the public transit network with the temporary graph instead of the shortcut graph in line 7. The temporary graph is sufficient for the query to yield correct results, as it contains edges from the source to any possible first stop, all edges required to transfer between trips, and edges from any possible last stop to the target. Since there are no additional requirements on the black box public transit algorithm, it is easy to see that any existing public transit algorithm can be used with our shortcuts.

Running Time Optimizations. We can further improve the performance of this query algorithm in practice by introducing some adjustments. First, we observe that we actually do not need edges from the source to every other stop. If the distance $d(s,v)$ from s to a stop v is greater than the distance $d(s,t)$ from s to t, every journey that requires a transfer from s to v is dominated by simply transferring directly from s to t. Thus, we do not need to add the edge (s,v) to the temporary graph. The same argument can be made for edges from some stop u to the target t if the distance $d(u,t)$ is greater than $d(s,t)$. Moreover, if we know that a stop v is further away from the source than the target, then we do not even need to compute the actual distance $d(s,v)$. We can use this fact to prune the search space of the Bucket-CH queries in lines 1 and 2. For this purpose, we first perform a standard bidirectional CH query from source to target that stops settling vertices from the forward (respectively backward) queue if the corresponding key is greater than the tentative distance from the source to the target. As a result we obtain the distance $d(s,t)$, as well as the partial forward (backward) CH search space from s (t), containing no vertices that have a greater distance from s (to t) than $d(s,t)$. We then perform the second phase of the Bucket-CH query (i.e., scanning the buckets) only for the vertices in the partial search spaces of the CH query. Furthermore, we store the entries in each bucket sorted by the distance to their target. Thus we can stop scanning through the bucket of a vertex u once we reach a stop v within the bucket with $d(s,u) + d(u,v) \geq d(s,t)$. Doing so can drastically improve local queries, as we do not need to look at all stops, but only at stops that are close to the source or target.

If we do not treat the underlying public transit algorithm as a black box, we can further improve practical performance by omitting the construction of the temporary graph G''. Instead of adding edges from s to stops v, we can directly initialize the tentative arrival times used by most public transit algorithms with $\tau_{\text{dep}} + d(s,v)$. Instead of adding edges to t, we try to update the tentative arrival time at the target with the arrival time at v plus $d(v,t)$ whenever the arrival time at v is updated.

Table 1 Sizes of the used public transit networks and their transfer graphs (full and transitive).

Network	Stops	Routes	Trips	Stop events	Vertices	Full edges	Tran. edges
Switzerland	25 426	13 934	369 534	4 740 929	604 167	1 847 140	4 687 016
Germany	244 055	231 089	2 387 297	48 495 169	6 872 105	21 372 360	22 645 480

5 Experiments

All algorithms were implemented in C++17 compiled with GCC version 7.3.1 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. The shortcut preprocessing was performed in parallel on all 16 cores. The transfer graph contraction and the queries were performed on a single core.

Networks. We evaluated our technique on the public transit networks of Switzerland and Germany, which were previously used in [26]. The Switzerland network was extracted from a publicly available GTFS feed[1] and consists of two successive business days (30th and 31st of May 2017). The Germany network is based on data from `bahn.de` for Winter 2011/2012, comprising two successive identical days. For both networks, stops and connections outside of the country borders were removed. As unrestricted transfer graphs, we used the road networks of Switzerland and Germany, including pedestrian zones and stairs, which were obtained from OpenStreetMap[2] data. Vertices with degree one and two were contracted unless they coincided with stops. Unless stated otherwise, we used walking as the transfer mode, assuming a walking speed of 4.5 km/h on each edge. To obtain transitively closed transfer graphs (for comparison with standard RAPTOR and CSA), we inserted an edge between all stops for which the distance in the transfer graph lies below a certain threshold (15 minutes for Switzerland, 8 minutes for Germany) and then computed the transitive closure. An overview of the networks is given in Table 1.

5.1 Preprocessing

In this section we evaluate the performance of the ULTRA preprocessing phase, including the transfer graph contraction and the shortcut computation.

Core Degree and Witness Limit. The two main parameters influencing the performance of the ULTRA preprocessing are the average vertex degree of the contracted transfer graph and the witness limit $\bar{\tau}$. Figure 1 shows the impact of these two parameters on the Switzerland network. The lowest preprocessing times are achieved with a core degree of 14. While the actual shortcut computation still becomes slightly faster for higher core degrees, this is offset by the increased time required to contract the transfer graph. Contracting up to a core degree of 14 took 1:29 minutes and yielded a graph with 32 683 vertices and 466 331 edges. By contrast, the witness limit $\bar{\tau}$ only has a minor impact on the number of computed shortcuts, with a difference of fewer than 600 shortcuts between $\bar{\tau} = 0$ and $\bar{\tau} = \infty$. For all following experiments, we chose a witness limit of 15 minutes, yielding 139 669 shortcuts for Switzerland.

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

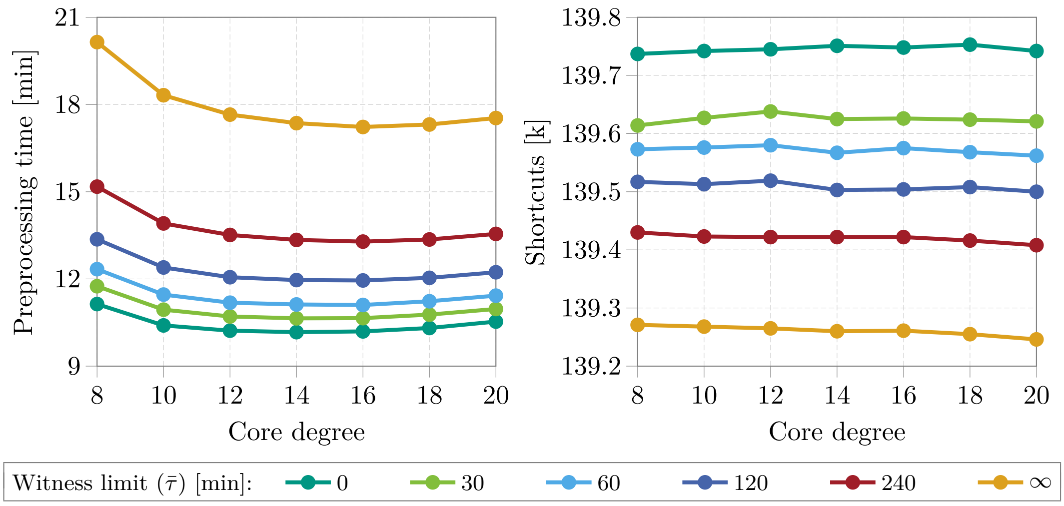

Figure 1 Impact of core degree and witness limit on the running time of the preprocessing algorithm and the number of computed shortcuts, measured on the Switzerland network. Preprocessing time includes both contracting the transfer graph and computing the shortcuts.

For the Germany network, we chose to contract up to a core degree of 20, since the share of the core computation in the overall preprocessing time decreases as the network size increases. Contraction took 24:56 minutes and produced a core graph with 314 021 vertices and 6 280 440 edges. As before, we used a witness limit of 15 minutes for the shortcut computation, which yielded 2 077 374 shortcuts.

Parallelization. We used all 16 cores of our machine in parallel to accelerate the shortcut computation. On the Switzerland network this reduced the shortcut computation time from 2:02:55 hours sequentially to 9:35 minutes, which corresponds to a speedup of 12.8. Thus, we obtain a total preprocessing time of 11:05 minutes, including the time for the contraction, which was not parallelized. This yields an overall speedup for the preprocessing phase of 11.2. For the Germany network the sequential shortcut computation would take several days, while computing the shortcuts in parallel using all 16 cores took 10:53:35 hours.

Transfer Speed. In order to test the impact of the used transfer mode on the shortcut computation, we changed the transfer speed in the Switzerland network from 4.5 km/h to different values between 1 km/h and 140 km/h. We considered two ways of applying the transfer speed: In the first version, we did not allow the transfer speed on an edge to exceed the speed limit given in the road network. This allowed us to model fast transfer modes such as cars fairly realistically. In the second version, we ignored speed limits and assumed a constant speed on every edge. Thus, we can analyze to which extend the effects observed in the first version are caused by the speed limit data. Figure 2 (left side) reports the number of computed shortcuts measured for each configuration. In all measurements, the preprocessing time remained below 15 minutes. A peak in the number of shortcuts is reached between 10 and 20 km/h, which roughly corresponds to the speed of a bicycle. If speed limits are ignored, the number of shortcuts then starts decreasing again for higher transfer speeds and reaches a plateau at around 188 000 shortcuts. If speed limits are obeyed, the number of shortcuts eventually rises again and reaches the overall peak at 140 km/h, which was the highest speed limit observed in the network.

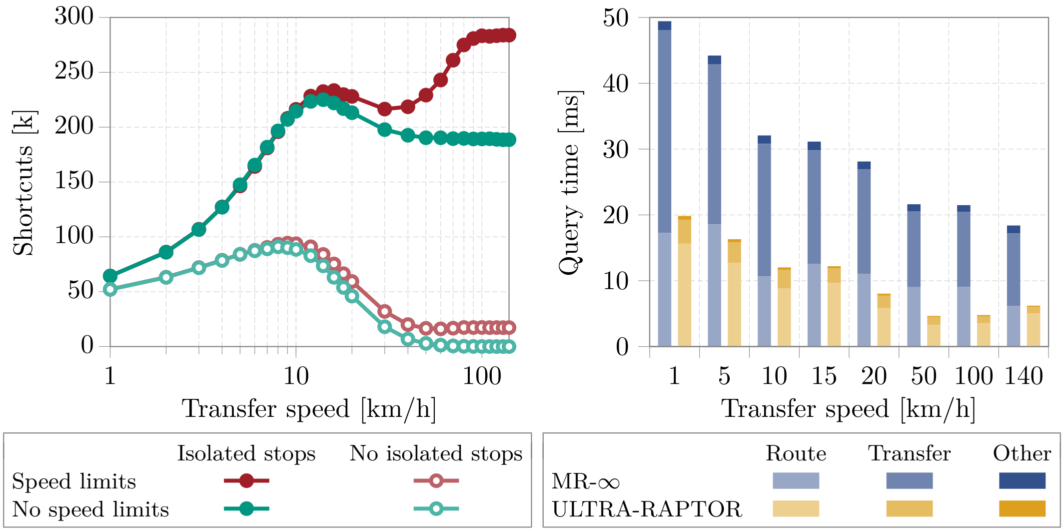

Figure 2 Impact of transfer speed, measured on the Switzerland network with a core degree of 14 and a witness limit of 15 minutes. *Left:* Number of computed shortcuts. Speed limits in the network were obeyed for the red lines and ignored for the green lines. For the two lines at the bottom, shortcuts were only added to the result if the source and target stop for which they were found were connected by a path in the transfer graph. *Right:* Query performance of MR-∞ and ULTRA-RAPTOR, averaged over 10 000 random queries. Speed limits were obeyed. Query times are divided into route collecting/scanning, transfer relaxation, and remaining time.

For low to medium transfer speeds, the results conformed with our expectations. As the transfer speed increases, it becomes increasingly feasible to cover large distances in the transfer graph quickly, making it possible to transfer between trips that are further away from each other. Accordingly, new shortcuts appear between these trips. However, once the transfer speed becomes competitive with the public transit vehicles, we would expect the number of shortcuts to decrease sharply as it eventually becomes preferable to avoid the public transit network altogether and transfer directly from source to target. The reason why this decrease is not observed in our measurements is that not all stops in our network instances are connected to the transfer graph. Consider what happens in the shortcut computation for journeys between stops s and t that are isolated from each other and the rest of the transfer graph. In this case, a direct transfer is not possible, regardless of the transfer speed. In fact, unless there is a route that serves both s and t, any optimal journey from s to t will include at least two trips. If a transfer is necessary between these two trips, then the journey is an nondominated candidate journey and a shortcut is added for the corresponding transfer. In our Switzerland network, 625 stops are isolated from the transfer graph, usually as a result of incomplete or imperfect data. To assess the impact of these stops on the number of computed shortcuts, we repeated our experiments, this time not adding shortcuts to the result if the source and target stop of the corresponding candidate journey were not connected in the transfer graph. This resulted in much fewer shortcuts, especially for high transfer speeds. If speed limits are ignored, the amount of necessary shortcuts becomes negligible at around 60 km/h and eventually reaches 0. If speed limits are obeyed, the number of shortcuts stagnates at 17 000.

Overall, these experiments show that our shortcut computation remains feasible regardless of the speed of the used transfer mode. Moreover, if the network does not include many stops that are isolated from the transfer graph, transferring between stops is most useful for transfer speeds between 10 and 20 km/h.

Table 2 Query performance for RAPTOR, MR-∞, and ULTRA-RAPTOR. Query times are divided into phases: initial transfers, collecting routes, scanning routes, and relaxing transfers. All results are averaged over 10 000 random queries. RAPTOR (marked with *) only supports stop-to-stop queries with transitive transfers, instead of vertex-to-vertex queries on the full graph.

Network	Algorithm	Full graph	Scans [k] Routes	Scans [k] Edges	Time [ms] Init.	Time [ms] Collect	Time [ms] Scan	Time [ms] Relax	Time [ms] Total
Switzerland	RAPTOR*	○	27.2	3 527	0.0	3.7	6.4	7.8	18.4
	MR-∞	●	34.9	769	11.6	5.9	8.2	12.3	39.3
	ULTRA-RAPTOR	●	37.7	148	1.6	4.9	7.9	1.9	16.7
Germany	RAPTOR*	○	480.4	25 798	0.0	166.9	178.0	85.1	436.5
	MR-∞	●	555.8	12 571	191.1	250.7	202.2	272.2	944.1
	ULTRA-RAPTOR	●	610.6	2 224	26.8	204.5	202.9	37.0	477.8

5.2 ULTRA Queries

To evaluate the impact of our shortcuts on the query performance, we tested them with two public transit algorithms, RAPTOR and CSA. For each algorithm, we compared three query variants: one using our ULTRA approach, one using a transitively closed transfer graph, and one using a multi-modal variant of the algorithm on an unrestricted transfer graph.

RAPTOR Queries. In the case of RAPTOR, we used the MR-∞ variant of MCR as the multi-modal algorithm, employing the same core graph that was used by the ULTRA preprocessing. The results of our comparison are shown in Table 2. Using ULTRA-RAPTOR drastically reduces the time consumption for exploring the transfer graph compared to MR-∞, from 50–60% of the overall running time to 10–20%. The reason for this is that both scanning the initial/final transfers and relaxing the intermediate transfers are an order of magnitude faster in ULTRA-RAPTOR compared to MR-∞. For the initial and final transfers, the Core-CH search of MR-∞ is replaced by a Bucket-CH query in ULTRA-RAPTOR. Similarly, ULTRA-RAPTOR uses shortcuts for relaxing the intermediate transfers whereas MR-∞ performs a Dijkstra search in the core graph. Overall, ULTRA-RAPTOR is twice as fast as MR-∞ and has a similar running time to RAPTOR with transitive transfers. Note that comparing the running times of RAPTOR and ULTRA-RAPTOR has to be done with caution, as they were measured for a different set of queries. Hence, our shortcut technique enables RAPTOR to use unrestricted transfers without incurring the performance loss that is associated with MCR.

CSA Queries. For CSA, incorporating unrestricted transfers efficiently is more challenging. Since no multi-modal variant of CSA has been published thus far, we implemented a naive multi-modal version of CSA, which we call MCSA, as a baseline for our comparison. This algorithm alternates connection scans with Dijkstra searches on the contracted core graph, in a similar manner to MCR. Query times for all three CSA variants are reported in Table 3. Note that CSA solves an easier problem than RAPTOR, since it only minimizes the arrival time and not the number of transfers. When using a transitive transfer graph, it is thus approximately three times as fast as RAPTOR. With unrestricted transfers, we observe that MCSA is slower than MR-∞. This is because alternating between connection scans and Dijkstra searches causes MCSA to lose the main performance advantage of CSA,

Table 3 Query performance for CSA, MCSA, and ULTRA-CSA. Query times are divided into two phases: initialization including initial transfers (Init.), and connection scans including intermediate transfers (Scan). All results are averaged over 10 000 random queries. CSA (marked with *) only supports stop-to-stop queries with transitive transfers, instead of vertex-to-vertex queries on the full graph.

Network	Algorithm	Full graph	Scans [k]		Time [ms]		
			Connections	Edges	Init.	Scan	Total
Switzerland	CSA*	○	126.7	1 307	0.2	5.0	5.1
	MCSA	•	88.0	5 337	12.9	48.4	61.3
	ULTRA-CSA	•	87.3	52	1.8	3.1	4.9
Germany	CSA*	○	2 620.3	6 216	2.9	162.1	165.1
	MCSA	•	1 568.2	118 026	233.6	1462.5	1696.1
	ULTRA-CSA	•	1 562.5	665	25.7	116.8	142.5

which is its high memory locality. When using ULTRA-CSA, however, this advantage is restored because only a few shortcut edges have to be relaxed after scanning each connection. Overall, ULTRA-CSA is only slightly slower than transitive CSA and about three times as fast as RAPTOR with shortcuts, making it the only efficient multi-modal variant of CSA known so far. Moreover, our query times for ULTRA-RAPTOR and ULTRA-CSA are significantly faster than those reported for the state-of-the-art techniques HLRaptor and HLCSA [22], by a factor of 3.6 and 11.1, respectively. With respect to preprocessing time and space consumption, HL-based techniques are also outperformed by ULTRA.

In addition to overall performance, we also measured how query times for RAPTOR are impacted by the transfer speed. Results are shown in Figure 2 (right side). The performance gains for ULTRA-RAPTOR compared to MR-∞ are similar for all transfer speeds, and in fact slightly better for higher speeds. In all cases, the entire query time for ULTRA-RAPTOR is similar to or lower than the time that MR-∞ takes for the route scanning phases only.

6 Conclusion

We developed a technique which significantly speeds up the computation of Pareto-optimal journeys in a public transit network with an unrestricted transfer graph. We achieved this by efficiently computing shortcuts that provably represent all necessary transfers. Parallelization enables fast precomputation, taking a few minutes on the network of Switzerland. Our evaluation showed that the number of computed shortcuts is low, regardless of the underlying transfer mode. The shortcuts can be used without adjustments by any public transit algorithm that previously required a transitively closed transfer graph. For RAPTOR and CSA, we showed that using shortcuts leads to similar query times as using a transitively closed transfer graph. Consequently, shortcuts enable the computation of unrestricted multi-modal journeys without incurring the performance losses of existing multi-modal algorithms. In particular, combining shortcuts with CSA yields the first efficient multi-modal variant of CSA.

For future work, we would like to develop a shortcut-based query algorithm that can answer many-to-many queries. It would also be interesting to adapt our shortcut precomputation to scenarios with additional Pareto criteria, such as walking distance or cost. Furthermore, it should be possible to extend the ULTRA approach to more complicated transfer modes, including bike or car sharing.

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 Hannah Bast and Sabine Storandt. Frequency-Based Search for Public Transit. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22. ACM Press, November 2014.

6 Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3:2.1–2.3:2.31, March 2010.

7 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.

8 Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '15, pages 44:1–44:10, New York, NY, USA, 2015. ACM.

9 Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 35–46. Springer, May 2010.

10 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.

11 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato Werneck. Public Transit Labeling. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, Lecture Notes in Computer Science, pages 273–285. Springer, 2015.

12 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zuendorf. Faster Transit Routing by Hyper Partitioning. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:14, Dagstuhl, Germany, 2017.

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

14 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.

15 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithmics*, 23:1.7:1–1.7:56, October 2018.

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

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

18 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)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.

19 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)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

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

21 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.

22 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)*, Lecture Notes in Computer Science. Springer, 2019.

23 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.

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

25 Ben Strasser and Dorothea Wagner. Connection Scan Accelerated. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 125–137. SIAM, 2014.

26 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)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.

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