



Transportation Science

Publication details, including instructions for authors and subscription information:
<http://pubsonline.informs.org>

ULTRA: Unlimited Transfers for Efficient Multimodal Journey Planning

Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, Tobias Zündorf

To cite this article:

Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, Tobias Zündorf (2023) ULTRA: Unlimited Transfers for Efficient Multimodal Journey Planning. Transportation Science

Published online in Articles in Advance 30 Aug 2023

. <https://doi.org/10.1287/trsc.2022.0198>

Full terms and conditions of use: <https://pubsonline.informs.org/Publications/Librarians-Portal/PubsOnLine-Terms-and-Conditions>

This article may be used only for the purposes of research, teaching, and/or private study. Commercial use or systematic downloading (by robots or other automatic processes) is prohibited without explicit Publisher approval, unless otherwise noted. For more information, contact permissions@informs.org.

The Publisher does not warrant or guarantee the article's accuracy, completeness, merchantability, fitness for a particular purpose, or non-infringement. Descriptions of, or references to, products or publications, or inclusion of an advertisement in this article, neither constitutes nor implies a guarantee, endorsement, or support of claims made of that product, publication, or service.

Copyright © 2023 The Author(s)

Please scroll down for article—it is on subsequent pages



With 12,500 members from nearly 90 countries, INFORMS is the largest international association of operations research (O.R.) and analytics professionals and students. INFORMS provides unique networking and learning opportunities for individual professionals, and organizations of all types and sizes, to better understand and use O.R. and analytics tools and methods to transform strategic visions and achieve better outcomes.

For more information on INFORMS, its publications, membership, or meetings visit <http://www.informs.org>

ULTRA: Unlimited Transfers for Efficient Multimodal Journey Planning

Moritz Baum,^a Valentin Buchhold,^a Jonas Sauer,^{a,*} Dorothea Wagner,^a Tobias Zündorf^a

^aKarlsruhe Institute of Technology, 76131 Karlsruhe, Germany

*Corresponding author

Contact: moritz@ira.uka.de (MB); buchhold@kit.edu,  <https://orcid.org/0000-0002-1506-3544> (VB); jonas.sauer2@kit.edu,

 <https://orcid.org/0000-0002-7196-7468> (JS); dorothea.wagner@kit.edu (DW); tobias.zuendorf@kit.edu (TZ)

Received: July 14, 2022

Revised: November 28, 2022


Accepted: April 26, 2023; May 15, 2023

Published Online in Articles in Advance:
August 30, 2023

<https://doi.org/10.1287/trsc.2022.0198>

Copyright: © 2023 The Author(s)

Abstract. We study a multimodal journey planning scenario consisting of a public transit network and a transfer graph that represents a secondary transportation mode (e.g., walking, cycling, e-scooter). The objective is to compute Pareto-optimal journeys with respect to arrival time and the number of used public transit trips. Whereas 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. Existing approaches, therefore, typically only support limited walking between stops by either imposing a maximum transfer distance or requiring the transfer graph to be transitively closed. To overcome these shortcomings, we propose a novel preprocessing technique called unlimited transfers (ULTRA): given an unlimited transfer graph, which may represent any non-schedule based transportation mode, ULTRA computes a small number of transfer shortcuts that are provably sufficient for computing a Pareto set of optimal journeys. 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 improves these algorithms from limited to unlimited transfers without sacrificing query speed. This is true not just for walking, but also for faster transfer modes, such as bicycle or car. Compared with the state of the art for multimodal journey planning, the fastest ULTRA-based algorithm achieves a speedup of an order of magnitude.

 **Open Access Statement:** This work is licensed under a Creative Commons Attribution 4.0 International License. You are free to copy, distribute, transmit and adapt this work, but you must attribute this work as “*Transportation Science*. Copyright © 2023 The Author(s). <https://doi.org/10.1287/trsc.2022.0198>, used under a Creative Commons Attribution License: <https://creativecommons.org/licenses/by/4.0/>.”

Funding: This work was supported by Deutsche Forschungsgemeinschaft [Grant WA 654/23-2].

Supplemental Material: The online appendix is available at <https://doi.org/10.1287/trsc.2022.0198>.

Keywords: journey planning • public transit • algorithm engineering

1. Introduction

Research on efficient route-planning algorithms has seen remarkable advances in the past two decades. On road networks, queries can be answered in less than a millisecond with moderate preprocessing effort even for continental-scale graphs. Similar results are currently out of reach for public transit networks, but state-of-the-art algorithms nevertheless achieve query times of a few milliseconds on metropolitan and mid-sized country networks (Bast et al. 2016). Even more challenging is the multimodal journey planning problem, which combines schedule-based (i.e., public transit) and non-schedule based (e.g., walking, cycling, driving) modes of transportation. Whereas this covers a greater variety of possible journeys, solving it efficiently remains difficult (Wagner and Zündorf 2017).

In this work, we consider a multimodal problem that augments public transit with a transfer graph, which represents one arbitrary, non-schedule based transportation mode. This transfer mode can be used at the start and end of a journey to enter and exit the public transit network and for transferring between public transit vehicles in the middle of the journey. Given a source and target vertex in the transfer graph and a departure time, the objective is to compute Pareto-optimal journeys with respect to arrival time and the number of used public transit trips.

1.1. Related Work

Journey planning algorithms for public transit networks can be divided into graph-based approaches and algorithms that operate directly on the timetable, exploiting

its schedule-based structure (Bast et al. 2016). Graph-based approaches model the public transit network as a graph and then answer queries with Dijkstra's (1959) algorithm, which can be sped up by applying preprocessing techniques (Delling, Pajor, and Wagner 2009; Bast et al. 2010; Bauer, Delling, and Wagner 2011; Delling et al. 2015; Bast, Hertel, and Storandt 2016). The two main modeling approaches are the time-dependent (Disser, Müller-Hannemann, and Schnee 2008; Pyrga et al. 2008) and time-expanded (Müller-Hannemann and Schnee 2007, Pyrga et al. 2008) models. The time-expanded model uses vertices to represent events in the timetable (e.g., a vehicle arriving at or departing from a stop) and edges to connect consecutive events. By contrast, the time-dependent model represents stops (e.g., a train station) of the network as vertices and connects two stops with an edge if they are served consecutively by at least one vehicle. Associated with each edge is a function that maps departure time to travel time. Both models can integrate footpaths (Disser, Müller-Hannemann, and Schnee 2008; Bast et al. 2010; Delling et al. 2015) but only as direct edges between public transit stops. This means that an unrestricted footpath network cannot be encoded efficiently because the number of edges would be quadratic in the number of stops. To ensure a reasonable graph size, footpaths are typically restricted to small, connected components of nearby stops (Delling, Katz, and Pajor 2012), for example, by limiting the maximal duration (e.g., five minutes of walking) or distance (e.g., 400 m) (Bast and Storandt 2014; Bast, Hertel, and Storandt 2016; Giannakopoulou, Paraskevopoulos, and Zaroliagis 2019) of footpaths.

Notable timetable-based approaches include the round-based public transit optimized router (RAPTOR) (Delling, Pajor, and Werneck 2015), connection scan algorithm (CSA) (Dibbelt et al. 2018), and the corresponding speedup techniques, HypRAPTOR (Delling et al. 2017) and ACSA (Dibbelt et al. 2018). Instead of exploring the public transit network with Dijkstra's (1959) algorithm, these algorithms rely on array-based scanning operations that improve cache locality. As with the graph-based approaches, footpaths can be integrated as transfer edges between pairs of stops. However, these are required to be one-hop transfers; that is, at most one transfer edge may be used when transferring between two public transit trips. This removes the need for Dijkstra (1959) searches within the transfer graph as every possible destination can be reached with a single edge. Additionally, both RAPTOR and CSA require that the transfer graph is transitively closed, which ensures that optimal journeys never require multiple transfer edges in succession. RAPTOR can be modified to lift this restriction (Delling, Dibbelt, and Pajor 2019), allowing for one-hop transfers without a transitive closure. In this case, journeys with multiple consecutive transfer edges are prohibited, and

the algorithm finds optimal journeys among those that remain. This can lead to counterintuitive journeys that take detours to avoid using two transfer edges in succession. On the other hand, computing the transitive closure significantly increases the size of the transfer graph. As shown by Wagner and Zündorf (2017), limiting the maximal transfer duration to 20 minutes before computing the transitive closure already leads to a graph that is too large for practical applications.

A special case among the timetable-based approaches is trip-based routing (TB) (Witt 2015), which requires a preprocessing phase that computes transfers between pairs of trips. This is done by enumerating all possible transfers and then applying a set of pruning rules to omit some but not all unnecessary transfers. TB requires a transitively closed transfer graph as input and was originally only evaluated for very sparse transfer graphs. Because it enumerates all transfers before pruning them, the preprocessing time is highly sensitive to the size of the transfer graph. Lehoux and Loiodice (2020) mitigate this by proposing an alternative transfer enumeration method that discards many unnecessary transfers before they are enumerated. However, neither version of the TB preprocessing supports unrestricted transfer graphs.

Using a restricted transfer graph is often justified with the argument that long transfers are rarely useful. However, experiments show that the availability of unrestricted walking significantly reduces travel times (Wagner and Zündorf 2017, Sauer 2018, Phan and Viennot 2019). Naturally, this effect is even stronger for faster transportation modes, such as bicycle or car. Handling unrestricted transfer graphs (which may represent any non-schedule based transportation mode) requires multimodal journey planning algorithms. These algorithms typically work by interleaving an existing public transit algorithm with Dijkstra (1959) searches on the transfer graph. Notable examples are user-constrained contraction hierarchies (UCCH) (Dibbelt, Pajor, and Wagner 2015) and multimodal multicriteria RAPTOR (MCR) (Delling et al. 2013), which are based on a time-dependent, graph-based approach and RAPTOR, respectively. Because the Dijkstra (1959) searches are expensive, these algorithms are slow compared with their pure public transit counterparts. More recently, HL-RAPTOR and HL-CSA (Phan and Viennot 2019) were proposed. Here, RAPTOR and CSA are interleaved with two-hop searches based on hub labeling (HL) (Abraham et al. 2011) instead of Dijkstra (1959). Whereas this requires a moderately expensive preprocessing phase, the authors report a speedup of 1.7 over the bicriteria variant of MCR for HL-RAPTOR.

1.2. Contribution

Preliminary experiments (Sauer 2018) show 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. By 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 unlimited transfers (ULTRA), 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 provably sufficient for answering all possible queries correctly.

ULTRA shortcuts can be used without adjustment by any algorithm that requires one-hop transfers between stops. In our experimental evaluation, we demonstrate this for RAPTOR and CSA. The resulting multimodal algorithms have roughly the same query performance as the original restricted algorithms regardless of the speed of the considered transfer mode. In particular, ULTRA-CSA is the first multimodal variant of CSA. For TB, we show that only minor changes are necessary to make ULTRA compute shortcuts between trips instead of stops. This allows ULTRA to replace the TB preprocessing phase, enabling unlimited transfers. We demonstrate that this significantly reduces the number of required shortcuts and the query time compared with a naive approach, that is, using the output of ULTRA as input for the TB preprocessing. Overall, ULTRA-TB outperforms the bicriteria version of MCR, which was previously the fastest multimodal algorithm, by about an order of magnitude. This yields query times of a few milliseconds on metropolitan networks and less than 100 ms on the much larger network of Germany.

1.3. Outline

The remainder of this work is structured as follows. Section 2 establishes basic notation and gives an overview of the algorithms on which ULTRA builds. We then describe the ULTRA shortcut computation in Section 3 and prove that it computes a sufficient set of transfer shortcuts. Section 4 explains how the transfer shortcuts can be integrated into query algorithms that require one-hop transfers. We also present modifications to the TB query algorithm to make it more efficient in a multimodal setting. We evaluate the performance of our preprocessing and query algorithms on real-world multimodal networks in Section 5. Finally, we summarize our results and give an outlook on potential future work in Section 6.

2. Preliminaries

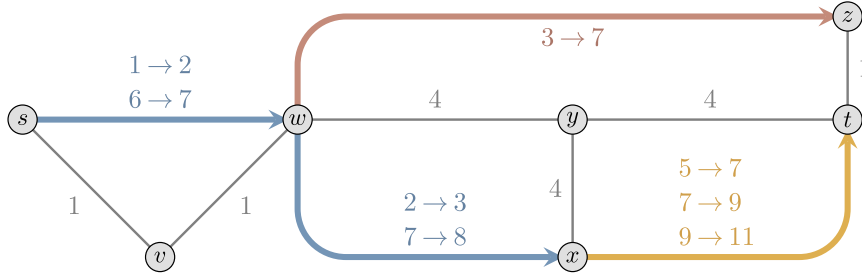
This section establishes basic terminology and introduces foundational algorithms.

2.1. Terminology

2.1.1. Network. A public transit network is a four-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 $G = (\mathcal{V}, \mathcal{E})$. A stop is a location in the network where passengers can board or disembark a vehicle (such as buses, trains, ferries, etc.). A trip $T = \langle \epsilon_0, \dots, \epsilon_k \rangle \in \mathcal{T}$ is a sequence of stop events performed by the same vehicle. A stop event $\epsilon = (\tau_{\text{arr}}(\epsilon), \tau_{\text{dep}}(\epsilon), v(\epsilon))$ represents the vehicle arriving at the stop $v(\epsilon)$ with the arrival time $\tau_{\text{arr}}(\epsilon)$ and subsequently departing from the same stop with the departure time $\tau_{\text{dep}}(\epsilon)$. The i th stop event in T is denoted as $T[i]$. The length $|T| := k$ is the number of stop events in T . A trip segment $T^{ij} := \langle \epsilon_i, \dots, \epsilon_j \rangle$ is a contiguous subsequence of T that begins at $T[i]$ and ends at $T[j]$. The set of routes \mathcal{R} is a partition of \mathcal{T} such that two trips that are part of the same route visit the same sequence of stops and do not overtake each other. A trip $T_a \in \mathcal{T}$ overtakes a trip $T_b \in \mathcal{T}$ if there exist two indices $i < j$ such that T_a arrives at or departs from $v(T_a[i])$ not before T_b but arrives at or departs from $v(T_a[j])$ not after T_b . Given a trip T , the route of T is denoted as $R(T)$. The length $|R|$ of a route R is the length of any trip belonging to the route.

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 $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Traveling along an edge $e = (v, w) \in \mathcal{E}$ requires the transfer time $\tau_{\text{tra}}(e)$. The notion of transfer time carries over to paths $P = \langle v_1, \dots, v_k \rangle$ in G , using the definition $\tau_{\text{tra}}(P) := \sum_{i=1}^{k-1} \tau_{\text{tra}}((v_i, v_{i+1}))$. Unlike in scenarios with limited footpaths, we impose no restrictions on 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. An example of a public transit network with an unrestricted transfer graph is shown in Figure 1.

2.1.2. Journeys. A journey describes the movement of a passenger through the network from a source vertex $s \in \mathcal{V}$ to a target vertex $t \in \mathcal{V}$. Each ride of the passenger in a public transit vehicle can be described by a trip segment, whereas the transfers between the rides are represented by paths in the transfer graph. An intermediate transfer between two trip segments T_a^{ij} and T_b^{mn} is a path P in G such that (1) the path begins with the last stop of T_a^{ij} , that is, $v(T_a[j])$; (2) the path ends with the first stop of T_b^{mn} , that is, $v(T_b[m])$; and (3) the transfer time of the path is sufficient to reach T_b^{mn} after vacating T_a^{ij} . This can be expressed formally as $\tau_{\text{arr}}(T_a[j]) + \tau_{\text{tra}}(P) \leq \tau_{\text{dep}}(T_b[m])$. An initial transfer before a trip segment T^{ij} is a path in G from the source s to the first stop

Figure 1. (Color online) An Example of a Public Transit Network with an Unrestricted Transfer Graph

Notes. Edges in the transfer graph (thin lines) are labeled with their travel time. Routes are displayed as sequences of thick lines. The lines (s, w) and (w, x) belong to the same route. Each edge is labeled with the departure and arrival times of the associated in trips in the format $\tau_{\text{dep}} \rightarrow \tau_{\text{arr}}$. For a query from s to t with departure time 0, a Pareto set with respect to arrival time and number of trips consists of the journeys $J_0 = \langle \langle s, v, w, t \rangle \rangle$ with arrival time 10, $J_1 = \langle \langle s, v, w \rangle, \langle 3 \rightarrow 7 \rangle, \langle z, t \rangle \rangle$ with arrival time 8, and $J_2 = \langle \langle s \rangle, \langle 1 \rightarrow 2, 2 \rightarrow 3 \rangle, \langle x \rangle, \langle 5 \rightarrow 7 \rangle, \langle t \rangle \rangle$ with arrival time 7.

of T^{ij} . Correspondingly, a final transfer after a trip segment T^{ij} is a path in G from the last stop of T^{ij} to the target t .

A journey $J = \langle P_0, T_0^{ij}, \dots, T_{k-1}^{mn}, P_k \rangle$ is an alternating sequence of transfers and trip segments. Note that some or all of the transfers may be empty, that is, consist of a single stop only. Given source and target vertices $s, t \in \mathcal{V}$, we call journey J an s - t journey if P_0 begins with s and P_k ends with t . The departure time of the journey is defined as $\tau_{\text{dep}}(J) := \tau_{\text{dep}}(T_0[i]) - \tau_{\text{tra}}(P_0)$ and the arrival time as $\tau_{\text{arr}}(J) := \tau_{\text{arr}}(T_{k-1}[n]) + \tau_{\text{tra}}(P_k)$. The number of trips used by the journey is denoted as $|J| := k$. An important special case is a journey $J = \langle P_0 \rangle$ that consists solely of a path in the transfer graph. Because such a journey does not use any trips, it can be traversed at any time. Thus, its departure time $\tau_{\text{dep}}(J)$ has to be stated separately, and its arrival time is then given by $\tau_{\text{arr}}(J) := \tau_{\text{dep}}(J) + \tau_{\text{tra}}(P_0)$. The vertex sequence of J is the concatenation of its transfers: $\mathcal{V}(J) = P_0 \circ P_1 \circ \dots \circ P_k$. A subjourney of J is a journey $J_s = \langle P'_x, T_x^{gh}, \dots, T_{y-1}^{pq}, P'_y \rangle$ such that $\langle T_x^{gh}, \dots, T_{y-1}^{pq} \rangle$ is a contiguous subsequence of J , P'_x is a suffix of P_x , and P'_y is a prefix of P_y . If $x = 0$ and $P'_x = P_0$, we call J_s a prefix of J . Conversely, if $y = k$ and $P'_y = P_k$, we call J_s a suffix of J . Note that a subjourney may start or end in the middle of a transfer but never in the middle of a trip segment. Given two vertices $v, w \in \mathcal{V}(J)$, the subjourney of J from v to w is denoted as J_{vw} .

2.1.3. Problem Statement. To evaluate the usefulness of a journey J , we mainly consider the two criteria arrival time $\tau_{\text{arr}}(J)$ and number of trips $|J|$. Given a set of criteria, a journey J weakly dominates another journey J' if J is not worse than J' in any criterion. Moreover, J strongly dominates J' if J is strictly better than J' in at least one criterion and not worse in the others. Given a query consisting of source and target vertices $s, t \in \mathcal{V}$ and an earliest departure time τ_{dep} , a journey is called feasible if it is an s - t journey that does not depart earlier than τ_{dep} . A feasible journey J is called Pareto-

optimal if no other feasible journey exists that strongly dominates J . A Pareto set is a set \mathcal{J} containing a minimal number of Pareto-optimal journeys such that every feasible journey is weakly dominated by a journey in \mathcal{J} . For a given query, the objective is to compute a Pareto set with respect to the two criteria: arrival time and number of trips. See Figure 1 for a Pareto set of journeys in the shown example network.

2.1.4. Departure Buffer Times. Many works on public transit routing (e.g., Pyrga et al. 2008, Delling et al. 2015) allow a minimum change time to be specified for each stop. It must be observed when transferring between two trips at the same stop but not when entering a trip after arriving via a path in the transfer graph or when entering the first trip at the start of the journey. The minimum change time is useful for modeling stops that represent larger stations with multiple platforms. Here, the minimum change time represents the time needed to change between platforms. This modeling choice is reasonable for settings with direct transfers between stops. However, when allowing an unrestricted transfer graph, it can lead to inconsistencies. Given a stop with minimum change time τ , if a path starting and ending at this stop with a transfer time less than τ exists, then taking that path allows passengers to circumvent the minimum change time.

To prevent this, we introduce departure buffer times as an alternative modeling approach. Each stop $v \in \mathcal{S}$ has a nonnegative departure buffer time $\tau_{\text{buf}}(v)$, which is the minimum amount of time that has to pass after arriving at the stop before a vehicle can be boarded. Unlike the minimum change time, the departure buffer time always has to be observed when a trip is entered even if the stop was reached via a transfer or if the trip is the first one in the journey. Departure buffer times can be integrated into the network implicitly by reducing the departure times of the stop events accordingly. For each stop event $\epsilon = (\tau_{\text{arr}}(\epsilon), \tau_{\text{dep}}(\epsilon), v(\epsilon))$, we obtain the reduced stop event $\epsilon' = (\tau_{\text{arr}}(\epsilon), \tau_{\text{dep}}(\epsilon) - \tau_{\text{buf}}(v(\epsilon)),$

$v(\epsilon)$). Note that this may cause stop events to appear as if they depart before they arrive. However, because the departure time is only relevant when entering the trip at the current stop and not when remaining seated in the trip, this does not lead to trips that travel backward in time. In the following, we do not discuss departure buffer times explicitly and instead assume that they are integrated into the departure times as described here.

2.2. Algorithms

We now give an overview of the algorithms on which ULTRA is based.

2.2.1. Dijkstra’s (1959) Algorithm. Given a graph $G = (\mathcal{V}, \mathcal{E})$ with edge length function $\ell : \mathcal{E} \rightarrow \mathbb{R}_0^+$ and a source vertex $s \in \mathcal{V}$, Dijkstra’s (1959) algorithm computes for each vertex v the length of the shortest s – v path. It maintains for each vertex v a tentative distance $\text{dist}[v]$, which is initialized with ∞ . Additionally, it maintains a priority queue Q of vertices ordered by their key, which is the tentative distance. Initially, s is inserted into Q with key $\text{dist}[s] = 0$. Then, vertices are extracted from Q in increasing order of key. Each extracted vertex v is settled by relaxing its outgoing edges. An edge $e = (v, w) \in \mathcal{E}$ is relaxed by comparing the tentative distance $\text{dist}[w]$ to the distance $\text{dist}[v] + \ell(e)$ that is achieved by traversing e . If the latter is smaller, $\text{dist}[w]$ is updated accordingly, and w is inserted into Q with key $\text{dist}[w]$.

2.2.2. Contraction Hierarchies. To explore the transfer graph, ULTRA utilizes algorithms based on contraction hierarchies (CH) (Geisberger et al. 2012), a preprocessing technique originally developed to speed up one-to-one queries in road networks. The basic building block of CH is vertex contraction: a vertex is contracted by removing it from the graph and inserting shortcut edges between its neighbors such that shortest path distances in the graph are preserved. The CH preprocessing phase for a graph $G = (\mathcal{V}, \mathcal{E})$ iteratively contracts the vertices of G in a heuristically determined order. The position of a vertex in this contraction order is called its rank. The output of this preprocessing phase is an augmented graph $G^+ = (\mathcal{V}, \mathcal{E}^+)$ that contains all original edges and all inserted shortcut edges. The augmented graph can be split into an upward graph $G^\uparrow = (\mathcal{V}, \mathcal{E}^\uparrow)$ containing only edges from lower to higher ranked vertices and a corresponding downward graph $G^\downarrow = (\mathcal{V}, \mathcal{E}^\downarrow)$. Queries are answered with a bidirectional variant of Dijkstra’s (1959) algorithm, in which the forward search explores G^\uparrow and the backward search explores G^\downarrow .

Bucket-CH (Knopp et al. 2007, Geisberger et al. 2012) is an extension of CH for one-to-many queries. It operates in three phases. First, given the graph $G = (\mathcal{V}, \mathcal{E})$, the CH precomputation is performed. Second, given

the set $\mathcal{V}_t \subseteq \mathcal{V}$ of targets, a bucket containing distances to the targets is computed for every vertex. This is done by performing a backward search on G^\downarrow from every target vertex $t \in \mathcal{V}_t$. For each vertex v settled by this search with distance $\text{dist}(v, t)$, the entry $(t, \text{dist}(v, t))$ is added to the bucket of v . Finally, given a query with source vertex s , the algorithm performs a forward search on G^\uparrow . For each vertex v settled by this search with distance $\text{dist}(s, v)$, the bucket of v is evaluated. For each bucket entry $(t, \text{dist}(v, t))$, the shortest distance to t found so far is compared with $\text{dist}(s, v) + \text{dist}(v, t)$ and updated if it is improved.

Multimodal algorithms, such as UCCH and MCR, employ a special variant of the CH precomputation that we call core-CH (Bauer et al. 2010; Delling et al. 2013; Dibbelt, Pajor, and Wagner 2015). Here, the precomputation is not allowed to contract vertices that coincide with stops. Thus, a set of core vertices \mathcal{V}^c with $\mathcal{S} \subseteq \mathcal{V}^c \subseteq \mathcal{V}$ is left uncontracted. In addition to the (partially) augmented graph, this yields a core graph $G^c = (\mathcal{V}^c, \mathcal{E}^c)$, which consists of \mathcal{V}^c and all shortcuts that were inserted between core vertices. If only stops are allowed as core vertices, the number of core edges is quadratic in the number of stops. This slows down both the precomputation and query algorithms to the point at which they become impractical. In practice, the contraction process is, therefore, stopped once the average vertex degree in the core graph surpasses a specified limit.

2.2.3. RAPTOR. To explore the public transit network, ULTRA employs algorithms from the RAPTOR family. RAPTOR (Delling, Pajor, and Werneck 2015) answers one-to-one and one-to-all queries in a public transit network with one-hop transfers. It operates in rounds, in which the i th round finds journeys with i trips by appending an additional trip to journeys found in the previous round. For each stop $v \in \mathcal{S}$ and each round i , the algorithm maintains a tentative arrival time $\tau_{\text{arr}}(v, i)$, which is the earliest arrival time among all journeys to v with at most i trips found so far. Each round consists of a route scanning phase followed by a transfer relaxation phase. Round i assumes that every stop v for which $\tau_{\text{arr}}(v, i - 1)$ is improved in round $i - 1$ has been marked. Before either phase is performed, $\tau_{\text{arr}}(v, i)$ is initialized with $\tau_{\text{arr}}(v, i - 1)$ for each stop $v \in \mathcal{S}$. Then, the route scanning phase collects all routes that visit marked stops and scans them. A route R is scanned by iterating across all visited stops, starting at the first marked stop. During the scan, the algorithm maintains an active trip T_{min} , which is the earliest trip of R that can be entered at any of the already processed stops. Let v be the j th stop of R . If T_{min} has already been set, the algorithm checks whether exiting T_{min} at v with arrival time $\tau_{\text{arr}}(T_{\text{min}}[j])$ improves $\tau_{\text{arr}}(v, i)$. If so, $\tau_{\text{arr}}(v, i)$ is updated accordingly, and v is marked.

Afterward, the algorithm checks whether there is an earlier trip than T_{\min} that can be entered when arriving at v with arrival time $\tau_{\text{arr}}(v, i - 1)$. If so, T_{\min} is updated accordingly. After all collected routes have been scanned, the transfer relaxation phase is performed. For every marked stop v , each outgoing transfer edge $e = (v, w) \in \mathcal{E}$ is relaxed. If $\tau_{\text{arr}}(v, i) + \tau_{\text{tra}}(e)$ is smaller than $\tau_{\text{arr}}(w, i)$, the latter is updated, and w is marked as well. For a query with source stop $s \in \mathcal{S}$ and departure time τ_{dep} , the algorithm initializes $\tau_{\text{arr}}(s, 0)$ with 0 and all other arrival times in round 0 with ∞ . Then, round 0 is performed, which marks s and relaxes its outgoing transfers. Afterward, new rounds are performed until no more stops have been marked.

An extension of RAPTOR called McRAPTOR (Delling, Pajor, and Werneck 2015) is able to Pareto-optimize additional criteria besides arrival time and number of trips. In turn, McRAPTOR can be extended to support multimodal scenarios with unlimited transfers. The resulting algorithm, MCR (Delling et al. 2013), replaces the transfer relaxation phase of (Mc)RAPTOR with a Dijkstra (1959) search on a core graph computed with core-CH. ULTRA employs the bicriteria variant of MCR, which was originally proposed under the name MR- ∞ , but which we call MR for the sake of simplicity. MR maintains the tentative arrival time $\tau_{\text{arr}}(v, i)$ for every core vertex $v \in \mathcal{V}^c$, not just for stops. The transfer relaxation phase runs Dijkstra’s (1959) algorithm on the core graph, using $\tau_{\text{arr}}(\cdot, i)$ as the tentative distances. The priority queue is initialized with all marked stops, and all stops that are settled by the search are themselves marked. Note that the Dijkstra (1959) search on the core graph can only guarantee to find shortest paths between pairs of stops. However, the source and target vertices $s, t \in \mathcal{V}$ may not necessarily be stops. Initial and final transfers are, therefore, explored with searches on the upward and downward graph produced by core-CH, respectively.

Another RAPTOR extension, rRAPTOR (Delling, Pajor, and Werneck 2015), answers range queries, which ask for a Pareto set of journeys for every departure time within a given interval. rRAPTOR exploits the observation that every Pareto-optimal journey (except for a direct transfer from s to t) starts by entering a trip at s or a stop reachable via a transfer from s . This limits the number of possible departure times to a small set \mathcal{DT} of discrete values. For each of these departure times, rRAPTOR performs a run of the basic RAPTOR algorithm. The departure times are processed in descending order, and the arrival times $\tau_{\text{arr}}(\cdot, \cdot)$ are not reset between runs. As a result, journeys found during the current run are implicitly pruned by journeys that depart later and neither arrive later nor have more trips. This property of rRAPTOR is called self-pruning.

2.2.4. Trip-Based Routing. A faster alternative to RAPTOR for one-to-one queries is TB (Witt 2015). It includes a preprocessing phase that computes transfers between pairs of stop events by first generating all possible transfers and then removing unnecessary ones in a transfer-reduction phase. The query algorithm resembles a breadth-first search on the set of stop events. Instead of tentative arrival times at stops, TB maintains a reached index $r(T)$ for each trip T . This is the index k of the first stop event $T[k]$ that has already been reached by the search. Initially, it is set to $|T|$. As with RAPTOR, TB operates in rounds, in which each round scans trip segments collected in a first in, first out (FIFO) queue. When the algorithm reaches a stop event $T[j]$, it calls the Enqueue operation: if $j < r(T)$, the trip segment T^{jk} with $k = r(T) - 1$ is added to the queue of the next round. Then, the reached index is updated: for each trip T' of the route $R(T)$ that does not depart before T , the reached index $r(T')$ is set to $\min(r(T'), j)$. Initially, the algorithm processes stops that are reachable from the source stop s with a transfer. For each stop v and each route R visiting v , the algorithm finds the earliest trip of R that can be entered at v and calls the Enqueue operation for the corresponding stop event. Then, the algorithm performs rounds until the next queue is empty. A trip segment T^{jk} is scanned by iterating over the stop events from $T[j]$ to $T[k]$. For each stop event $T[i]$, the outgoing precomputed transfers are relaxed. A transfer $(T[i], T'[i'])$ is relaxed by calling the Enqueue operation for $T'[i']$. Additionally, TB maintains a Pareto set of journeys at the target stop t . If t is reachable from $v(T[i])$ via a transfer, the algorithm adds the produced journey to the Pareto set and removes dominated journeys.

3. Shortcut Computation

We now present the ULTRA preprocessing phase, which computes shortcut edges that represent intermediate transfers between trips. These shortcuts must be sufficient for answering every point-to-point query correctly. This is achieved if every query can be answered with a Pareto set of journeys whose intermediate transfers are all represented by shortcuts. On the other hand, the number of shortcuts should be as small as possible to allow for fast queries.

We present two variants of the ULTRA preprocessing, which differ in the granularity of the computed shortcuts: In the stop-to-stop variant, shortcuts connect pairs of stops. This is sufficient for most public transit algorithms, including RAPTOR and CSA. The event-to-event variant computes shortcuts between stop events, which are required by TB. Unlike stop-to-stop shortcuts, these also provide information about the specific trips between which a transfer is necessary. Both variants are

identical except for a few crucial details, which are discussed explicitly as appropriate.

ULTRA works by enumerating a set of journeys \mathcal{J}^c with exactly two trips such that all required shortcuts occur as intermediate transfers in \mathcal{J}^c . For each enumerated journey, the intermediate transfer is unpacked, and a shortcut is generated for it. Before we describe the algorithm, we first establish a definition for \mathcal{J}^c that is sufficient for answering all queries but keeps the number of shortcuts as low as possible. We then provide a high-level overview of the ULTRA shortcut computation and prove that it enumerates \mathcal{J}^c . Afterward, we discuss running time optimizations to make the algorithm efficient in practice. Finally, we compare event-to-event ULTRA to the TB preprocessing and show that it is more effective at discarding unnecessary transfers.

3.1. Enumerating a Sufficient Set of Journeys

Consider the subproblem in which only queries between fixed source and target vertices $s, t \in \mathcal{V}$ must be answered. Then, the following naive algorithm computes a sufficient set of shortcuts: Enumerate the set \mathcal{J}^{opt} of all s - t journeys J that are Pareto-optimal for the departure time $\tau_{\text{dep}}(J)$ and generate a shortcut for every intermediate transfer that occurs in \mathcal{J}^{opt} . This produces more shortcuts than necessary: if there are multiple Pareto-optimal journeys that are equivalent in both criteria, only one of them is required to answer a query. The goal is, therefore, to find a set $\mathcal{J}^{\text{canon}} \subseteq \mathcal{J}^{\text{opt}}$ of journeys that excludes such duplicates but is still sufficient for answering all queries correctly. We observe that every journey in $\mathcal{J}^{\text{canon}}$ with more than two trips can be decomposed into subjourneys with two trips each. Every shortcut that occurs in $\mathcal{J}^{\text{canon}}$ also occurs in the much smaller set containing only these subjourneys. To exploit this algorithmically, we require that $\mathcal{J}^{\text{canon}}$ is closed under subjourney decomposition, that is, every subjourney of a journey in $\mathcal{J}^{\text{canon}}$ is itself contained in $\mathcal{J}^{\text{canon}}$.

3.1.1. Tiebreaking Sequences. In order to achieve closure under subjourney decomposition, ties between equivalent journeys must be broken in a consistent manner. For this purpose, we define total orderings on the sets of routes and vertices with a route index function $\text{id}_{\mathcal{R}} : \mathcal{R} \rightarrow \mathbb{N}$ and a vertex index function $\text{id}_{\mathcal{V}} : \mathcal{V} \rightarrow \mathbb{N}$. Then, ties between equivalent journeys are broken as follows: journeys that end with trip segments are preferred over journeys that end with (nonempty) transfers. For journeys that end with a trip segment T^{ij} , the index of the route $R(T)$ and the index i at which the trip segment starts are used as tiebreakers in this order. For journeys that end with an edge (w, v) , ties are broken first by considering the arrival time at w and then by considering the vertex index $\text{id}_{\mathcal{V}}(w)$. If two journeys

share a nonempty suffix, this suffix is ignored, and the respective prefixes of the journeys are compared instead. To formalize these tiebreaking rules, we associate with each s - t journey J a unique tiebreaking sequence. The tiebreaking sequence $X(v, J)$ of a vertex $v \in \mathcal{V}(J)$ with $v \neq s$ is defined as

$$X(v, J) := \begin{cases} \langle \tau_{\text{arr}}(J_{sv}), \text{id}_{\mathcal{R}}(R(T)), i, \infty, \infty \rangle & \text{if } J_{sv} \text{ ends with a trip segment } T^{ij} \\ \langle \tau_{\text{arr}}(J_{sv}), \infty, \infty, \tau_{\text{arr}}(J_{sv}), \text{id}_{\mathcal{V}}(w) \rangle & \text{if } J_{sv} \text{ ends with an edge } (w, v). \end{cases}$$

The tiebreaking sequence of an s - t journey J with vertex sequence $\mathcal{V}(J) = \langle s = v_1, \dots, v_k = t \rangle$ is defined as $X(J) := X(v_k, J) \circ \dots \circ X(v_2, J)$. This sequence is unique among all s - t journeys. In particular, if two journeys J and J' end with trip segments $T_a^{ij} \neq T_b^{mn}$, then their tiebreaking sequences are different. If $\tau_{\text{arr}}(J) = \tau_{\text{arr}}(J')$ and $R(T_a) = R(T_b)$, then $T_a = T_b$, and $j = n$ must hold because the trips cannot overtake each other. Then, the tiebreaking sequences are different because of $i \neq m$. Sequences are ordered lexicographically: for sequences $A = \langle a_1, a_2, \dots, a_k \rangle$ and $B = \langle b_1, b_2, \dots, b_k \rangle$ of equal length, $A < B$ if $a_1 < b_1$, or $a_1 = b_1$ and $\langle a_2, \dots, a_k \rangle < \langle b_2, \dots, b_k \rangle$. For sequences of different length, the shorter one is padded with $-\infty$ on the right side before they are compared.

3.1.2. Canonical Journeys. Because tiebreaking sequences are strictly ordered, ambiguities between equivalent journeys can be resolved by replacing the criterion arrival time with the tiebreaking sequence. We say that an s - t journey J canonically dominates another s - t journey J' if $X(J) < X(J')$ and $|J| \leq |J'|$. Because the two tiebreaking sequences cannot be equal, there is no need to distinguish between strong and weak canonical dominance. An s - t journey J is called canonical if it is Pareto-optimal with respect to the tiebreaking sequence and number of trips for the departure time $\tau_{\text{dep}}(J)$, that is, if no other s - t journey exists that is feasible for $\tau_{\text{dep}}(J)$ and canonically dominates J . Because no two journeys can be equivalent in both criteria, the set that consists of all feasible canonical journeys is the only Pareto set for any given query. We call this the canonical Pareto set. The set $\mathcal{J}^{\text{canon}}$ is the union of the canonical Pareto sets for all possible s - t queries. This set is closed under subjourney decomposition.

Lemma 1. *For every canonical s - t journey J and every pair $v, w \in \mathcal{V}(J)$ of vertices visited by J , the subjourney J_{vw} is canonical.*

Proof. Assume that J_{vw} is not canonical. Then, there is a journey J'_{vw} such that J'_{vw} is feasible for $\tau_{\text{dep}}(J_{vw})$, $X(J'_{vw}) < X(J_{vw})$ and $|J'_{vw}| \leq |J_{vw}|$. Because J'_{vw} does not depart earlier or arrive later than J_{vw} , replacing J_{vw} with J'_{vw} in J yields a feasible journey J' with $|J'| \leq |J|$. Adding the prefix J_{sv} to J'_{vw} and J_{vw} adds identical

suffixes to both tiebreaking sequences. This does not change their relative order, so $X(J'_{sw}) < X(J_{sw})$. Similarly, adding the suffix J_{wt} to J'_{sw} and J_{sw} adds identical prefixes to both tiebreaking sequences, which does not change their relative order. Therefore, $X(J') < X(J)$ and J is not canonical. \square

3.1.3. Candidate Journeys. We exploit the closure of $\mathcal{J}^{\text{canon}}$ under subjourney decomposition by defining a suitable set of subjourneys that need to be enumerated. A candidate is a journey that consists of two trips connected by an intermediate transfer but with empty initial and final transfers. Every canonical journey that uses at least two trips can be decomposed into candidate subjourneys. By Lemma 1, these subjourneys are themselves canonical. Accordingly, every shortcut that occurs in $\mathcal{J}^{\text{canon}}$ also occurs in the set $\mathcal{J}^c \subseteq \mathcal{J}^{\text{canon}}$ of canonical candidate journeys. A sufficient set of shortcuts can, therefore, be computed by enumerating \mathcal{J}^c .

3.1.4. Canonical MR. Canonical Pareto sets can be computed by making slight modifications to MR in order to ensure proper tiebreaking: first, at the start of each round, the collected routes are sorted according to $\text{id}_{\mathcal{R}}$ before they are scanned. The second change concerns the keys of vertices in the Dijkstra (1959) priority queue. In standard MR, the key of a vertex v in round i is the tentative arrival time $\tau_{\text{arr}}(v, i)$ at v with i trips. This is now replaced with $\langle \tau_{\text{arr}}(v, i), \text{id}_{\mathcal{V}}(v) \rangle$. The resulting implementation of MR, which we call canonical MR, finds equivalent journeys in increasing order of tiebreaking sequence. Hence, canonical journeys are found first, and all other equivalent journeys are discarded because they are weakly dominated by them. This is proven by the following lemma.

Lemma 2. *Canonical MR returns the canonical Pareto set for every query.*

Proof. See Online Appendix A.

The journeys returned by a straightforward (noncanonical) implementation of MR are not closed under subjourney decomposition. An example demonstrating this is given in Online Appendix B.

3.2. Algorithm Overview

We now describe how \mathcal{J}^c can be enumerated efficiently. Directly applying the definition of \mathcal{J}^c yields a simple but wasteful approach: for every possible source stop and every possible departure time, a one-to-all canonical MR search restricted to the first two rounds is performed. A candidate J^c is canonical if there is no feasible journey J^w with at most two trips that canonically dominates J^c (and is, therefore, found before J^c by the respective canonical MR search). If such a journey J^w exists, we call it a witness because its existence proves that J^c is not canonical. Unlike candidates, witnesses

may have nonempty initial or final transfers, and they may use fewer than two trips. If there is no witness for a candidate J^c , the corresponding canonical MR search includes J^c in its Pareto set. A shortcut representing the intermediate transfer of J^c is then generated.

3.2.1. Adapting rRAPTOR. The reason this approach is wasteful is that it does not exploit the self-pruning property of rRAPTOR: if journeys with later departure times are explored first, they can be used to dominate worse journeys with an earlier departure time. We, therefore, adapt rRAPTOR to the ULTRA setting: the RAPTOR search that is performed in each run is replaced with a canonical two-round MR search. This version of rRAPTOR is then invoked for each possible source stop $s \in \mathcal{S}$ with a departure time interval that covers the entire duration of the timetable.

We can make further improvements by carefully choosing the departure times for which runs are performed. rRAPTOR performs a run for every possible departure time τ_{dep} at s . A departure time τ_{dep} is possible if there is a stop v (which may be s itself) that is reachable from s via an initial transfer of length $\tau_{\text{tra}}(s, v)$ and a trip that departs from v at $\tau_{\text{dep}} + \tau_{\text{tra}}(s, v)$. If transfers are unrestricted, the number of possible departure times is very high because, typically, most stops in the network are reachable from s . Accordingly, a straightforward multimodal adaptation of rRAPTOR performs many runs and is, therefore, slow. In the context of ULTRA, however, most possible departure times require a nonempty initial transfer, which means that the corresponding runs would not find any candidates. Because the goal is to enumerate candidates, ULTRA only performs the runs for departure events that occur directly at s . Let $\mathcal{DT} = \{\tau_{\text{dep}}^0, \dots, \tau_{\text{dep}}^k\}$ be the set of possible departure times directly at s , sorted in ascending order. The run for τ_{dep}^i explores candidates departing at τ_{dep}^i and witnesses with departure times in the interval $[\tau_{\text{dep}}^i, \tau_{\text{dep}}^{i+1})$. We define $\tau_{\text{dep}}^{k+1} := \infty$ to ensure that the run for τ_{dep}^k explores all witnesses that depart after τ_{dep}^k . By integrating the witness search into the candidate runs, the algorithm skips many witnesses that would be required to answer a range query but are irrelevant for dominating candidates. Thus, the ULTRA preprocessing is much faster than a straightforward multimodal adaptation of rRAPTOR.

Algorithm 1 (ULTRA Transfer Shortcut Computation)

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R}, G)$, core graph $G^c = (\mathcal{V}^c, \mathcal{E}^c)$

Output: Shortcut graph $G^s = (\mathcal{S}, \mathcal{E}^s)$

```

1 for each  $s \in \mathcal{S}$  do
2   Clear all arrival labels and Dijkstra (1959)
   queues
3    $\tau_{\text{tra}}(s, \cdot) \leftarrow$  Compute transfer times in  $G^c$  from
    $s$  to all stops
```

```

4   DT ← Collect departure times of trips at s
5   for each  $\tau_{\text{dep}}^i \in DT$  in descending order do
6       //canonical MR run
7       Collect and sort routes reachable within
8       [ $\tau_{\text{dep}}^i, \tau_{\text{dep}}^{i+1}$ )
9       //first round
10      Scan routes
11      Relax transfers
12      Collect and sort routes serving updated
13      stops
14      //second round
15      Scan routes
16       $\mathcal{E}_{\text{canon}} \leftarrow$  Relax transfers, thereby collecting
17      shortcuts
18       $\mathcal{E}^s \leftarrow \mathcal{E}^s \cup \mathcal{E}_{\text{canon}}$ 
    
```

3.2.2. Pseudocode. High-level pseudocode for the ULTRA shortcut computation scheme is given by Algorithm 1. For each source stop $s \in S$, the algorithm performs the modified multimodal rRAPTOR search described in Section 3.2.1. To avoid redundant Dijkstra (1959) searches, initial transfers to all other stops are explored only once per source stop (line 3), and the results are then reused for each run in line 6. The departure times at s for which runs need to be performed are collected in line 4. The runs are performed in lines 6–12. Each run consists of two canonical MR rounds, which are subdivided into three phases: collecting routes and sorting them according to $\text{id}_{\mathcal{R}}$ (lines 4 and 6), scanning routes (lines 7 and 10), and relaxing transfers with a Dijkstra (1959) search (lines 8 and 11). After the final transfer relaxation phase in line 11, the remaining candidates that have not been dominated by witnesses are canonical, so shortcuts representing their intermediate transfers are added to the shortcut graph in line 12.

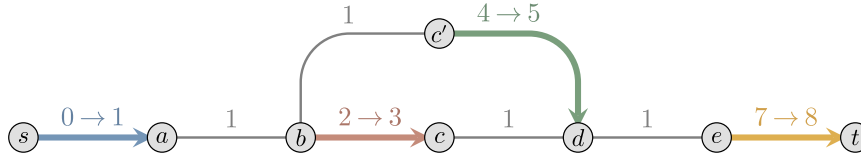
3.2.3. Extracting Shortcuts. The final transfer relaxation phase in line 11 identifies canonical candidates and extracts their shortcuts. Whenever a stop is settled during the Dijkstra (1959) search, the algorithm checks whether the corresponding journey J is a candidate, that is, has an empty initial and final transfer. If so, we know that J is canonical because any witness that canonically dominates it would have been found already. Therefore, an edge representing the intermediate transfer of J is added to the shortcut graph G^s . In order to extract the intermediate transfer, each vertex v maintains two parent pointers $p_1[v]$ and $p_2[v]$, where $p_k[v]$ is the parent for reaching v using k trips (i.e., within the k th MR round). If the journey to v ends with a trip, $p_k[v]$ points to the stop at which this trip was entered. If the journey ends with a transfer, it points to the stop at which the transfer starts. For a candidate ending at a stop t , the shortcut representing its intermediate transfer is given by $(p_1[p_2[t]], p_2[t])$. Because intermediate transfers only need to be extracted for

candidates, the parent pointer is set to a special value \perp if the corresponding journey has a nonempty initial or final transfer. Then, the final Dijkstra (1959) search in line 11 can check whether the journey ending at a stop v is a candidate or a witness by inspecting $p_2[v]$.

The event-to-event variant of ULTRA generates shortcuts not between stops, but between stop events. The parent pointer definitions are changed accordingly: if the journey to a vertex v ends with a trip, $p_k[v]$ now points to the stop event at which this trip was entered. If the journey ends with a transfer, it points to the stop event at which the preceding trip was exited. Because only candidates have valid parent pointers and candidates have empty initial transfers, this preceding trip always exists. For a candidate that ends at a stop t , the corresponding shortcut is now given by $(p_1[v(p_2[t])], p_2[t])$.

3.2.4. Repairing Self-Pruning. Using a rRAPTOR-based approach with self-pruning allows ULTRA to discard many irrelevant candidates early on. However, self-pruning can also cause the algorithm to discard canonical journeys. By exploring journeys with later departure times first, rRAPTOR implicitly maximizes departure time as a third criterion. With this additional criterion, there may be queries for which all Pareto-optimal journeys include suboptimal subjourneys. An example of this is shown in Figure 2. In this case, some canonical candidates are suboptimal for three criteria and, therefore, not found by the rRAPTOR-based scheme. Moreover, in the depicted network, there is no Pareto set for two criteria that is closed under subjourney decomposition and only includes journeys that are Pareto-optimal for three criteria. Hence, the problem cannot be avoided by defining $\mathcal{J}^{\text{canon}}$ in a different manner. Instead, we modify the dominance criterion to ensure that canonical journeys are not discarded.

For a journey J , let $\text{run}(J)$ be the highest i with $\tau_{\text{dep}}^i \in DT$ such that $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}^i$. This is the run in which our modified rRAPTOR finds J . For each vertex v and round i , the algorithm maintains a label $\ell(v, i) = (\tau_{\text{arr}}(v, i), p_i[v], \text{run}(v, i))$, where $\tau_{\text{arr}}(v, i)$ is the tentative arrival time, $p_i[v]$ is the parent pointer, and $\text{run}(v, i)$ is the run of the journey corresponding to this label, which we denote as $J(v, i)$. Let $\ell = (\tau_{\text{arr}}, p, j)$ be the label of a new journey J that is found by the algorithm at v in round i . Normally, rRAPTOR discards J if it is weakly dominated by $J(v, i)$, that is, $\tau_{\text{arr}}(v, i) \leq \tau_{\text{arr}}$. Otherwise, it replaces $\ell(v, i)$ with ℓ . Our modified algorithm discards J if it is weakly dominated by $J(v, i)$ and one of the following conditions is fulfilled: (1) J is not a prefix of a candidate, that is, $p = \perp$; (2) J is strongly dominated by $J(v, i)$, that is, $\tau_{\text{arr}}(v, i) < \tau_{\text{arr}}$ or $\tau_{\text{arr}}(v, i - 1) \leq \tau_{\text{arr}}$; or (3) $J(v, i)$ is found in the current run, that is, $\text{run}(v, i) = j$. With this modified dominance condition, we can prove

Figure 2. (Color online) An Example Network Showing the Conflict Between Canonicity and Self-Pruning

Notes. Every s-t journey that is Pareto-optimal with respect to the three criteria, arrival time, number of trips, and departure time, includes a sub-optimal subjourney. Transfer edges (thin lines) are labeled with their travel time, whereas trips (thick lines) are labeled with $\tau_{\text{dep}} \rightarrow \tau_{\text{arr}}$. The two Pareto-optimal journeys are $J = \langle \langle s \rangle, \langle 0 \rightarrow 1 \rangle, \langle a, b \rangle, \langle 2 \rightarrow 3 \rangle, \langle c, d, e \rangle, \langle 7 \rightarrow 8 \rangle, \langle t \rangle \rangle$ and $J' = \langle \langle s \rangle, \langle 0 \rightarrow 1 \rangle, \langle a, b, c' \rangle, \langle 4 \rightarrow 5 \rangle, \langle d, e \rangle, \langle 7 \rightarrow 8 \rangle, \langle t \rangle \rangle$. The subjourney J_{bt} of J is not Pareto-optimal because it has an earlier departure time than J'_{bt} and is otherwise equivalent. Likewise, the subjourney J'_{sd} is suboptimal because it has a later arrival time than J_{sd} .

that the ULTRA preprocessing computes a sufficient shortcut graph.

Theorem 1. For every canonical journey $J = \langle P_0, T_0^{ij}, \dots, T_{k-1}^{mn}, P_k \rangle$, every intermediate transfer in J is represented by an edge in the shortcut graph computed by ULTRA.

Proof. Consider an intermediate transfer P_{x+1} of J and the corresponding candidate subjourney $J^c = \langle T_x^{gh}, P_{x+1}, T_{x+1}^{pq} \rangle$. We show that the modified rRAPTOR search for the source stop $v(T_x[g])$ finds this candidate in the run for $\tau_{\text{dep}}(J^c)$ and inserts a shortcut for it. Assume J^c is not found. Then, some prefix J' of J^c is discarded by the search in favor of a witness J^w . By Lemma 1, J' is canonical and, therefore, not strongly dominated by J^w . Then, by our modified dominance criterion, J^w must have been found in the same canonical MR run as J' . However, by Lemma 2, canonical MR discards J^w in favor of J' , a contradiction. \square

3.3. Optimizations

We now discuss running time optimizations that are not mentioned in the high-level overview given by Algorithm 1. These optimizations are crucial for achieving fast preprocessing times.

3.3.1. Initial Route Collection. An rRAPTOR run with departure time τ_{dep}^i explores journeys that depart at s within the interval $[\tau_{\text{dep}}^i, \tau_{\text{dep}}^{i+1})$. Line 6 collects the set $\mathcal{R}(\tau_{\text{dep}}^i)$ of routes that must be scanned in the first round of this run. This set consists of all routes R for which there is a stop v visited by R and a trip T of R such that $\tau_{\text{dep}}(T, v) - \tau_{\text{tra}}(s, v) \in [\tau_{\text{dep}}^i, \tau_{\text{dep}}^{i+1})$. In order to speed up this step, the set $\mathcal{R}(\tau_{\text{dep}}^i)$ is precomputed when τ_{dep}^i is added to the set \mathcal{DT} of candidate departure times in line 4. This leads to the following procedure for calculating \mathcal{DT} and $\mathcal{R}(\cdot)$: first, the algorithm collects all departure triplets $(v, \tau_{\text{dep}}, R)$ of departure stop v , departure time τ_{dep} , and route R that occur in the network. They are then sorted by their departure time at s , which is $\tau_{\text{dep}} - \tau_{\text{tra}}(s, v)$, and processed in descending order. The algorithm maintains a tentative set \mathcal{R}' of routes for the next candidate departure time that is added to \mathcal{DT} . For each triplet $(v, \tau_{\text{dep}}, R)$, the

algorithm checks whether $v = s$. If $v \neq s$, R is added to \mathcal{R}' . Otherwise, τ_{dep} is a candidate departure time. If τ_{dep} is already contained in \mathcal{DT} , the algorithm already found another route departing from s at τ_{dep} , so R is added to $\mathcal{R}(\tau_{\text{dep}})$. Otherwise, τ_{dep} is added to \mathcal{DT} , $\mathcal{R}(\tau_{\text{dep}})$ is set to $\mathcal{R}' \cup \{R\}$, and \mathcal{R}' is cleared.

3.3.2. Limited Dijkstra (1959) Searches. The algorithm can be sped up by introducing a stopping criterion to the Dijkstra (1959) search for final transfers in line 11. For this purpose, the preceding route scanning phase in line 10 counts the number of stops that are marked because their tentative arrival time is improved by a candidate. Whenever such a stop is settled in line 11, the counter is decreased. Once the counter reaches zero, we know that the Dijkstra (1959) search has processed all candidates that have been found in this run, so it is stopped.

A similar stopping criterion is applied to the intermediate Dijkstra (1959) search in line 8. Here, the first route scanning phase in line 7 counts the stops whose tentative arrival time is improved by a candidate prefix, that is, a journey with an empty initial transfer. As in line 11, the Dijkstra (1959) search is stopped as soon as no such stops are left in the queue. This does not affect the correctness of the computed shortcut graph G^s because all candidates are still processed. However, some of the witnesses that are pruned might be required to dominate noncanonical candidates. In this case, superfluous shortcuts are added to G^s . This can be counteracted by continuing the Dijkstra (1959) search for some time after the last candidate prefix has been extracted. We introduce a parameter $\bar{\tau}_{\text{wit}}$ called the witness limit that determines how long the search continues. Let τ_{arr} be the arrival time of the last extracted candidate prefix. Instead of stopping the Dijkstra (1959) search immediately, it continues until the smallest element in the queue has an arrival time greater than $\tau_{\text{arr}} + \bar{\tau}_{\text{wit}}$.

Once a Dijkstra (1959) search is stopped, the remaining witness labels are kept in the queue because they may dominate candidates in later runs. This requires that the two Dijkstra (1959) searches in lines 8 and 11 use separate queues so that labels from the final Dijkstra (1959) search of a previous run do not interfere

with the intermediate Dijkstra (1959) search of the current run. As a consequence, if a label is discarded because it is dominated, it must be explicitly removed from any queues that still contain it. Moreover, the run in which a label is settled may no longer be the same one in which it was enqueued. Accordingly, the run in which a journey J is found may no longer equal $\text{run}(J)$. To ensure that the dominance condition is applied correctly, the run of a newly created label is carried over from its parent label rather than setting it to the currently performed run.

With these changes, the only remaining part of the algorithm that performs an unlimited Dijkstra (1959) search on the core graph is the initial transfer relaxation in line 3. Unlike the searches for the intermediate and final transfers, this search is only performed once for every source stop instead of once per run, so its impact on the overall running time is small.

3.3.3. Pruning with Found Shortcuts. Once a shortcut is found and added to the shortcut graph G^s , it is no longer necessary to find candidates that produce the same shortcut. We exploit this by further restricting the definition of candidates: a journey is only classified as a candidate if its intermediate transfer is not contained in the set of already computed shortcuts. Because this reduces the number of candidates, the stopping criterion for the Dijkstra (1959) searches in lines 8 and 11 may be applied earlier, further saving preprocessing time.

Whenever a potential candidate is found during the second route scanning phase in line 10, the stop-to-stop variant of ULTRA checks if the corresponding shortcut is already contained in G^s . If so, the journey is classified as a witness by setting its parent pointer to \perp . In the event-to-event variant, this check is more expensive because the number of shortcuts is much larger. Furthermore, because an event-to-event shortcut typically occurs in many fewer candidate journeys than its stop-to-stop counterpart, it is much less likely that the shortcut is already contained in G^s . Our preliminary experiments show that the benefit of potentially dismissing a candidate no longer outweighs the work required to look up the shortcut. Therefore, the check is skipped in the event-to-event variant.

When a candidate is extracted from the Dijkstra (1959) queue in line 11 and a shortcut is inserted for it, there may be other candidates remaining in the queue that use the same intermediate transfer. These must be turned into witnesses by setting the respective parent pointers to \perp . This requires keeping track of all candidates belonging to a particular shortcut. Within a single canonical MR run, the search can find at most one intermediate transfer ending at a particular stop or stop event. In stop-to-stop ULTRA, each stop v , therefore, maintains a list of all nondominated candidates

whose intermediate transfer ends at v . The event-to-event variant does the same for each stop event. When a shortcut is inserted, all candidates in the corresponding list are turned into witnesses.

3.3.4. Transfer Graph Contraction. As with MCR (Delling et al. 2013), the Dijkstra (1959) searches are performed on a core graph, which is constructed with core-CH in advance. Because ULTRA only needs to compute journeys between pairs of stops rather than arbitrary vertices in the transfer graph, only transfers that start and end at stops are relevant. Accordingly, the initial and final transfer searches that MCR performs on the upward and downward CH graphs can be omitted.

Another type of contraction is performed for cliques of stops that have a pairwise distance of zero in the transfer graph. These cliques typically occur when different platforms of a larger station are modeled as individual stops. Each such clique is contracted into a single stop in order to decrease the number of canonical MR runs that need to be performed. The number of runs for a source stop s is equal to the number of unique departure times at s . If a departure time occurs at multiple stops within a clique with transfer distance zero, then the algorithm performs one run for this departure time at each stop. The journeys found by these runs are identical save for initial transfers of length zero. By contracting the clique into a single stop, these redundant runs are merged into one. This does not affect the correctness of the algorithm because it is conceptually equivalent to allowing candidates to begin with an initial transfer of length zero.

3.3.5. Parallelization. Finally, we observe that ULTRA allows for trivial parallelization. The preprocessing algorithm searches for candidates once for every possible source stop (line 1 of Algorithm 1). As these searches are mostly independent of each other, they can be distributed to parallel threads, and the results are then combined in a final sequential step. The only aspect of the algorithm that introduces a dependency between the searches for different source stops is the restricted candidate definition: a journey is only considered a candidate if no shortcut has yet been added for its intermediate transfer. If a shortcut was added by a different thread, the algorithm does not notice this. However, because this is merely a performance optimization, the algorithm remains correct if only shortcuts added by the current thread are considered.

3.4. Integration with Trip-Based Routing

Unlike other public transit algorithms, TB on its own already requires a preprocessing step even when used without ULTRA. One possible approach for enabling unlimited transfers in TB is with a sequential three-

phase algorithm: First, shortcuts between stops are generated with the stop-to-stop variant of the ULTRA preprocessing. These are then used as input for the TB preprocessing, which generates event-to-event shortcuts that can be used by the ULTRA-TB query. However, we show that an integrated two-phase approach is superior. Here, the TB preprocessing is replaced entirely by the event-to-event variant of the ULTRA preprocessing. The resulting shortcuts between stop events are then used as input for the ULTRA-TB query. The advantage of the integrated approach is that it produces fewer shortcuts because ULTRA applies stricter pruning rules than the TB preprocessing. Both algorithms enumerate journeys with at most two trips in order to find witnesses that prove that a potential shortcut is not necessary. The TB preprocessing does this in a transfer-reduction step after all potential shortcuts have been generated. Because the latter is no longer feasible with unlimited transfers, ULTRA interleaves the generation and pruning of shortcuts. Furthermore, ULTRA examines a larger set of witnesses. In the TB preprocessing, witnesses must start with the same stop event as the candidate, whereas ULTRA also considers witnesses that start with a nonempty initial transfer or a different initial trip. Furthermore, because the TB preprocessing explores intermediate transfers by iterating along the stop sequence of the initial trip in reverse, a candidate cannot be pruned by witnesses that exit the initial trip before the candidate. Overall, ULTRA has more options for pruning candidates and, thus, produces fewer shortcuts.

4. Query Algorithms

ULTRA shortcuts can be combined with any public transit query algorithm that normally requires one-hop transfers. The idea is to replace the original transfer graph with the precomputed shortcuts and run the algorithm on the resulting network. Some algorithms, including RAPTOR, CSA, and TB, normally require that the transfer graph is transitively closed. Whereas this is not the case for the ULTRA shortcut graph, this is not a problem: Theorem 1 proves that journeys with two consecutive shortcut edges are never required to answer a query correctly. Accordingly, if a transitive edge is missing in the shortcut graph, we know that it is never required as part of an optimal journey.

Whereas the shortcut graph covers intermediate transfers between two trips, it does not provide any information for transferring from the source to the first trip or transferring from the last trip to the target. In this section, we describe how initial and final transfers can be integrated into the query algorithms efficiently. Additionally, we describe optimizations for the TB query

algorithm that make it more efficient in a scenario with unlimited transfers.

4.1. Query Algorithm Framework

The ULTRA query algorithm exploits the fact that, for initial and final transfers, one endpoint of the transfer is fixed. All initial transfers start at the source vertex s of the query, whereas all final transfers end at the target vertex t . Therefore, initial and final transfers can be explored with two additional one-to-many queries on the original transfer graph: a forward query to compute distances from s to all stops and a backward query for the distances from all stops to t . ULTRA uses bucket-CH for this task as it is one of the fastest known one-to-many algorithms and allows for optimization of local queries. Thus, ULTRA requires three preprocessing steps in total: First, a core graph is constructed with core-CH. This is then used as input for the transfer shortcut computation outlined in Section 3. The third step is the bucket-CH preprocessing for the original transfer graph G . The query algorithm then takes as input the public transit network, transfer shortcut graph, and bucket-CH data. Pseudocode for the query algorithm is shown in Algorithm 2.

A query begins with a bidirectional CH search from s to t in line 1. This yields the travel time $\tau_{\text{tra}}(s, t)$ for a direct transfer from s to t (which may be ∞ if no direct transfer is possible). A naive approach would then perform a forward bucket-CH query from s and a reverse bucket-CH query from t , yielding for every stop v the initial transfer distance $\tau_{\text{tra}}(s, v)$ and the final transfer distance $\tau_{\text{tra}}(v, t)$. However, not all of these distances are actually needed. An initial transfer to a stop v cannot be part of an optimal journey if $\tau_{\text{tra}}(s, v) \geq \tau_{\text{tra}}(s, t)$ because any journey containing the initial transfer is dominated by the direct transfer from s to t . Likewise, no optimal journey can include a final transfer to a stop v with $\tau_{\text{tra}}(v, t) \geq \tau_{\text{tra}}(s, t)$. The algorithm exploits this by using the forward and backward search spaces \mathcal{V}_s and \mathcal{V}_t of the bidirectional CH query. Because the CH search is stopped once the shortest s - t path is found, these contain no vertices whose distance from s and to t , respectively, is greater than $\tau_{\text{tra}}(s, t)$. Therefore, it is sufficient to scan the forward buckets of all vertices in \mathcal{V}_s (line 2) and the backward buckets of all vertices in \mathcal{V}_t (line 2). Additional query time can be saved by sorting the entries of each bucket in ascending order of distance during the preprocessing phase. Then, the scan for the forward bucket of a vertex v can be stopped once it reaches a stop w within the bucket with $\tau_{\text{tra}}(s, v) + \tau_{\text{tra}}(v, w) \geq \tau_{\text{tra}}(s, t)$ (and analogously for backward buckets). Doing so drastically improves local queries as they do

not need to evaluate all stops, but only stops that are close to the source or target.

Algorithm 2 (ULTRA Query Algorithm Framework)

Input: Public transit network (S, T, R, G) , transfer shortcut graph $G^s = (S, \mathcal{E}^s)$, bucket-CH data for G , source vertex s , departure time τ_{dep} , and target vertex t

Output: Pareto set \mathcal{J} of s - t journeys for departure time τ_{dep}

- 1 $(\tau_{\text{tra}}(s, t), \mathcal{V}_s, \mathcal{V}_t) \leftarrow$ Run a CH query from s to t with departure time τ_{dep}
- 2 $\tau_{\text{tra}}(s, \cdot) \leftarrow$ Evaluate the vertex-to-stop buckets for vertices in \mathcal{V}_s
- 3 $\tau_{\text{tra}}(\cdot, t) \leftarrow$ Evaluate the stop-to-vertex buckets for vertices in \mathcal{V}_t
- 4 $\tilde{G}^s \leftarrow (S \cup \{s, t\}, \mathcal{E}^s)$
- 5 Add edge (s, t) with travel time $\tau_{\text{tra}}(s, t)$
- 6 **for each** $v \in S \setminus \{s, t\}$ with $\tau_{\text{tra}}(s, v) < \tau_{\text{tra}}(s, t)$ **do**
- 7 \perp Add edge (s, v) to \tilde{G}^s with travel time $\tau_{\text{tra}}(s, v)$
- 8 **for each** $v \in S \setminus \{s, t\}$ with $\tau_{\text{tra}}(v, t) < \tau_{\text{tra}}(s, t)$ **do**
- 9 \perp Add edge (v, t) to \tilde{G}^s with travel time $\tau_{\text{tra}}(v, t)$
- 10 Run black-box public transit algorithm on $(S \cup \{s, t\}, T, R, \tilde{G}^s)$.

After the distances for the initial and final transfers are computed, the algorithm creates a temporary copy \tilde{G}^s of the shortcut graph G^s , which contains s and t as additional vertices. In lines 5–9, this temporary graph is complemented with edges for the initial and final transfers and the direct transfer from s and t , 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 \tilde{G}^s in line 10. The temporary graph is sufficient for obtaining correct results as it contains edges for all necessary initial, intermediate, and final transfers and an edge for a direct transfer from source to target. Because there are no additional requirements on the black-box public transit algorithm, it is easy to see that any existing algorithm can be used with ULTRA shortcuts.

If the public transit algorithm is not treated as a black box, the performance can be improved further by omitting the construction of \tilde{G}^s . Most public transit algorithms, including RAPTOR and CSA, maintain a tentative arrival time at each stop, which is improved as new journeys are found. Instead of adding an edge from s to a stop v , the tentative arrival time of v can be initialized with $\tau_{\text{dep}} + \tau_{\text{tra}}(s, v)$. To incorporate final transfers, whenever the tentative arrival time at a stop v is set to some value τ , the algorithm can try to improve the tentative arrival time at t with $\tau + \tau_{\text{tra}}(v, t)$.

4.2. Improved TB Query

Unlike most algorithms, TB already distinguishes between initial/final and intermediate transfers, exploring different graphs for both. The original transfer graph G is only used for the initial and final transfers, whereas intermediate transfers are explored using the precomputed

event-to-event transfers. In the context of ULTRA, this requires a modification to the query framework shown in Algorithm 2: the temporary graph \tilde{G}^s now only contains the edges added for the initial and final transfers and not the ULTRA shortcuts. The query then uses \tilde{G}^s for the initial and final transfers and the unmodified event-to-event shortcut graph $G^e = (\mathcal{V}^e, \mathcal{E}^e)$ for the intermediate transfers.

Additionally, the TB query algorithm can be optimized further for networks with unlimited transfers. The original query, as introduced by Witt (2015), is optimized for a use case in which only a few stops are reachable with an initial or final transfer. However, with unlimited transfers, it is typical for almost all stops to be reachable. Therefore, we restructure the query to allow the huge number of possible initial and final transfers to be processed more efficiently. Pseudocode for the modified query is given by Algorithm 3. In the following, we describe this algorithm in detail.

4.2.1. Initial Transfer Evaluation. As in the generic ULTRA query, the algorithm begins with the bucket-CH search (lines 1–3). This yields a minimum arrival time $\tau_{\text{arr}}(s, v)$ for every reached stop v as well as the minimum arrival time τ_{min} at t via a direct transfer. If $\tau_{\text{min}} < \infty$, a label representing the s - t journey with zero trips is added to the result set \mathcal{L} in line 5. The algorithm then identifies trips that are reachable via an initial transfer (lines 8–19). In the original TB query (Witt 2015), this is done by iterating over all stops that are reachable via an initial transfer. For each such stop v and each route R visiting v , the algorithm identifies the earliest trip of R that can be entered at v after taking the initial transfer. This approach is efficient as long as the number of stops reachable via an initial transfer is small. However, in a scenario with unlimited transfers in which almost all stops are reachable, consecutive stops of a route often share the same earliest reachable trip. This can cause the same trip to be found multiple times, leading to redundant work. To avoid this, we propose a new approach for evaluating the initial transfers, which is based on two steps of the RAPTOR algorithm: collecting updated routes and scanning routes.

Lines 8 and 9 collect all routes that visit a stop that is reachable via an initial transfer. This is analogous to collecting routes that visit marked stops at the beginning of a RAPTOR round. Then, all collected routes are scanned. As in RAPTOR, a route R is scanned by processing its stops in the order in which they are visited by R . The algorithm maintains an active trip T_{min} , which is the earliest trip of R that is reachable from any of the already processed stops. Initially, T_{min} is set to a dummy value \perp (line 11). Let v be the next stop to be processed during the scan of R . To check if T_{min} can be improved, the algorithm finds the earliest trip T'_{min} of R that can be boarded when arriving at v with the arrival time $\tau_{\text{arr}}(s, v)$. If no

reachable trip is found for any of the previous stops in R (i.e., $T_{\min} = \perp$), then T'_{\min} is found with a binary search. Otherwise, the algorithm starts a linear search from T_{\min} and looks backward for earlier trips. Because T'_{\min} is often not much earlier than T_{\min} , this is faster than a binary search in practice. Note that T'_{\min} is not found if it is later than T_{\min} , but in this case, entering T'_{\min} at v does not produce an optimal journey, so it can be discarded. If T'_{\min} is earlier than T_{\min} , then T_{\min} is updated, and the Enqueue operation is called for the corresponding stop event in line 18. The Enqueue operation itself is unchanged from the original TB query. If T'_{\min} is the earliest trip in R , the remainder of the route scan can be skipped.

Algorithm 3 (ULTRA-Trip-Based Query)

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R}, G)$, transfer shortcut graph $G^e = (\mathcal{V}^e, \mathcal{E}^e)$, bucket-CH data for G , source vertex s , departure time τ_{dep} , and target vertex t

Output: Labels \mathcal{L} representing Pareto set of s - t journeys for departure time τ_{dep}

```

1  $(\tau_{\text{tra}}(s, t), \mathcal{V}_s, \mathcal{V}_t) \leftarrow$  Run a CH query from  $s$  to  $t$ 
  with departure time  $\tau_{\text{dep}}$ 
2  $\tau_{\text{tra}}(s, \cdot) \leftarrow$  Evaluate the vertex-to-stop buckets for
  vertices in  $\mathcal{V}_s$ 
3  $\tau_{\text{tra}}(\cdot, t) \leftarrow$  Evaluate the stop-to-vertex buckets for
  vertices in  $\mathcal{V}_t$ 
4  $\tau_{\min} \leftarrow \tau_{\text{dep}} + \tau_{\text{tra}}(s, t)$ 
5 if  $\tau_{\min} < \infty$  then  $\mathcal{L} \leftarrow \{(\tau_{\min}, 0)\}$ 
6 for each  $v \in \mathcal{S}$  do  $\tau_{\text{arr}}(s, v) \leftarrow \tau_{\text{dep}} + \tau_{\text{tra}}(s, v)$ 
7  $\mathcal{R}', Q_1 \leftarrow \emptyset$ 
8 for each  $v \in \mathcal{S}$  with  $\tau_{\text{tra}}(s, v) < \tau_{\text{tra}}(s, t)$  do
9    $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{\text{Routes from } \mathcal{R} \text{ that contain } v\}$ 
10 for each  $R \in \mathcal{R}'$  do
11    $T_{\min} \leftarrow \perp$ 
12   for  $i$  from 0 to  $|R| - 1$  do
13      $v \leftarrow i$ -th stop of  $R$ 
14     if  $\tau_{\text{arr}}(s, v) \geq \tau_{\min}$  then continue
15      $T'_{\min} \leftarrow$  earliest  $T \in R$  departing from  $v$ 
16     if  $T'_{\min}$  is earlier than  $T_{\min}$  then
17        $T_{\min} \leftarrow T'_{\min}$ 
18       Enqueue( $T_{\min}[i], Q_1$ )
19     if  $T_{\min}$  is the first trip in  $R$  then break
20    $n \leftarrow 1$ 
21 while  $Q_n$  is not empty do
22   for each  $T^{jk} \in Q_n$  do
23     for  $i$  from  $j$  to  $k$  do
24       if  $\tau_{\text{arr}}(T[i]) \geq \tau_{\min}$  then break
25       if  $\tau_{\text{arr}}(T[i]) + \tau_{\text{tra}}(v(T[i]), t) < \tau_{\min}$  then
26          $\tau_{\min} \leftarrow \tau_{\text{arr}}(T[i]) + \tau_{\text{tra}}(v(T[i]), t)$ 
27          $\mathcal{L} \leftarrow \mathcal{L} \cup \{(\tau_{\min}, n)\}$ , removing domi-
          nated labels
28    $Q_{n+1} \leftarrow \emptyset$ 
29   for each  $T^{jk} \in Q_n$  do
30     for  $i$  from  $j$  to  $k$  do
31       if  $\tau_{\text{arr}}(T[i]) \geq \tau_{\min}$  then break
32       for each  $(T[i], T'[i']) \in \mathcal{E}^e$  do
33         Enqueue( $T'[i'], Q_{n+1}$ )
34    $n \leftarrow n + 1$ .

```

4.2.2. Trip Scanning. The trip-scanning phase (lines 20–34) is identical to the original TB query algorithm except for the evaluation of final transfers. It is organized in rounds, in which the n th round scans the trip segments that were previously collected in the FIFO queue Q_n . A trip segment T^{jk} is scanned by iterating over all stop events from $T[j]$ to $T[k]$. When scanning a stop event $T[i]$, the algorithm checks whether a final transfer from the i th stop of the trip T to the target exists in line 24. If such a transfer exists and improves the earliest known arrival time τ_{\min} at the target, then the algorithm has found a new Pareto-optimal journey. In this case, τ_{\min} is updated, and a label representing the newly found journey is added to the result set \mathcal{L} . If \mathcal{L} already contains a label with n trips (note that a Pareto set can only contain one such label), this label is replaced. After the final transfers are evaluated, the algorithm relaxes the outgoing shortcuts from $T[i]$. For each shortcut $(T[i], T'[i']) \in \mathcal{E}^e$, the Enqueue operation is called for $T'[i']$. This adds the relevant segment of T' to the queue Q_{n+1} of trips that are scanned in the next round.

Note that the trips in Q_n are scanned twice: once to evaluate the final transfers and then again to relax transfer shortcuts. This is done for two reasons: First, separating the two scans improves memory locality as $\tau_{\text{tra}}(\cdot, t)$ is only accessed by the first scan and \mathcal{E}^e is only accessed by the second scan. Second, τ_{\min} is improved throughout the first scan, which enables stricter pruning of trips that cannot contribute to Pareto-optimal journeys in line 31 of the second scan.

4.2.3. Data Structures and Memory Layout. In order to achieve optimal performance, the query algorithm needs to use a streamlined memory layout. To this end, the FIFO queues Q_n are implemented using dynamic arrays. This enables an efficient Enqueue operation and efficient scanning of the entries in Q_n . The shortcuts \mathcal{E}^e are stored in an array such that all outgoing shortcuts of a stop event $T[i]$ are consecutive in memory and the outgoing shortcuts of the next stop event $T[i + 1]$ follow directly afterward. Finally, note that the trip scanning step only needs access to the arrival time $\tau_{\text{arr}}(T[i])$ and the stop $v(T[i])$ of a stop event $T[i]$. Therefore, these values are stored separately from the departure time $\tau_{\text{dep}}(T[i])$ of the stop event, which improves memory locality.

5. Experiments

All algorithms were implemented in C++17 and compiled with GCC version 10.3.0 and optimization flag -O3. Experiments were performed on the following machines:

- Xeon: A machine with two eight-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.50 GHz with a

boost frequency of 4.2 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache.

- Eycp: A machine with two 64-core AMD Eycp Rome 7742 CPUs clocked at 2.25 GHz with a boost frequency of 3.4 GHz, 1,024 GiB of DDR4-3200 RAM, and 256 MiB of L3 cache.

Source code for ULTRA is available at <https://github.com/kit-algo/ULTRA>.

5.1. Networks

We evaluated our algorithms on the transportation networks of Stuttgart, London, Switzerland, and Germany. The Stuttgart network represents the greater region of Stuttgart and comprises two identical business days. It was previously used by Mallig, Kagerbauer, and Vortisch (2013) and Briem et al. (2017) and is not publicly available. The public transit timetable of London was obtained from Transport for London (<https://data.london.gov.uk>) and covers a single Tuesday in the periodic summer schedule of 2011. It was previously used to evaluate RAPTOR (Delling, Pajor, and Werneck 2015), MCR (Delling et al. 2013), and TB (Witt 2015). The Switzerland network was extracted from a publicly available general transit feed specification (<http://gtfs.geops.ch/>) and consists of two successive business days (May 30 and 31, 2017). Finally, the Germany network was provided by Deutsche Bahn for research purposes and is not publicly available. It is based on data from bahn.de for winter 2011/2012, comprising two successive identical days, and was previously used to evaluate CSA (Dibbelt et al. 2018) and TB (Witt 2015). Both the Switzerland and Germany networks were previously used by Wagner and Zündorf (2017). For each network, we computed the set \mathcal{R} of routes greedily by iterating across the set \mathcal{T} of trips: for each trip T , we checked if a route R with the same stop sequence as T was already generated such that T does not overtake any trips in R and is not overtaken by any of them itself. If so, we added T to R . Otherwise, we generated a new route for T .

We constructed unrestricted transfer graphs by extracting road graphs, including pedestrian zones and staircases, from OpenStreetMap (<https://download.geofabrik.de/>). Unless stated otherwise, we used walking as the transfer mode, assuming a constant speed of 4.5 km/h. The transfer graph was connected to the public transit network using the procedure outlined by Wagner and Zündorf (2017). For each stop $v \in \mathcal{S}$, we located its (geographically) nearest neighbor $w \in \mathcal{V}$ in the transfer graph. If v and w were less than five meters apart and v was also the nearest neighbor of w , we identified v with w . Otherwise, we added a new vertex for v and connected it to w if the distance was less than 100 meters. Afterward, vertices with degrees one and two were contracted unless they coincided with stops. Remote and isolated parts of the networks were removed

Table 1. Sizes of the Public Transit Networks and the Accompanying Transfer Graphs

	Stuttgart	London	Switzerland	Germany
Stops	13,584	19,682	25,125	243,167
Routes	12,351	1,955	13,786	230,255
Trips	91,304	114,508	350,006	2,381,394
Stop events	1,561,972	4,508,644	4,686,865	48,380,936
Vertices	1,166,604	181,642	603,691	6,870,496
Edges	3,682,232	575,364	1,853,260	21,367,044
Transitive edges	1,369,928	3,212,206	2,639,402	22,571,280

Note. Also reported is the number of edges in the transitively closed transfer graph used to compare ULTRA to unimodal algorithms.

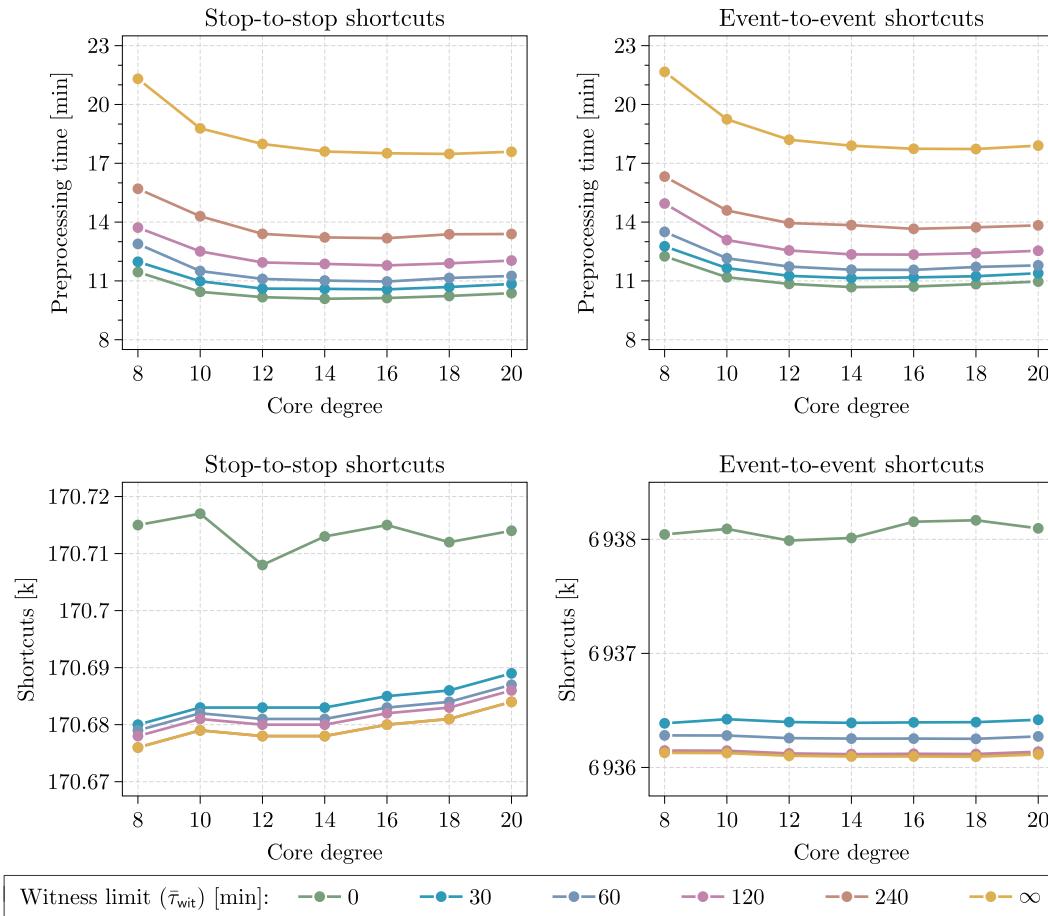
by applying a bounding box and removing everything except the largest connected component.

To obtain transitively closed transfer graphs (for comparison with standard RAPTOR, CSA, and TB), we inserted edges between all stops whose distance in the transfer graph lies below a certain threshold (nine minutes for Stuttgart and Switzerland, eight minutes for Germany, four minutes for London) and then computed the transitive closure. Following Wagner and Zündorf (2017), the thresholds were chosen so that the resulting graph has an average vertex degree of about 100. An overview of the networks is given in Table 1.

5.2. Preprocessing

In this section, we evaluate the performance of the ULTRA preprocessing phase, which includes the core-CH transfer graph contraction, the shortcut computation, and the bucket-CH computation. We analyze the effects of the parameters core degree, witness limit, and transfer speed in detail for the Switzerland network and then discuss more general results for all four networks.

5.2.1. Core Degree and Witness Limit. The two main parameters influencing the performance of the ULTRA preprocessing are the average vertex degree of the contracted core graph and the witness limit $\bar{\tau}_{\text{wit}}$. Figure 3 shows the impact of these parameters on the Switzerland network. The lowest preprocessing times are achieved with a core degree of 14. Although the actual shortcut computation is slightly faster for higher core degrees, this is offset by the increased time required to contract the transfer graph. The witness limit $\bar{\tau}_{\text{wit}}$ has a larger impact on the preprocessing time. Choosing a witness limit of zero instead of ∞ nearly cuts the preprocessing time in half. Regardless of core degree or witness limit, the event-to-event variant takes about one minute longer than the stop-to-stop variant. Both parameters have a negligible effect on the number of computed shortcuts. For all following experiments, we therefore choose a core degree of 14 and a witness limit of zero to minimize the

Figure 3. (Color online) Impact of Core Degree and Witness Limit on ULTRA Preprocessing

Notes. Measured are the running time of the ULTRA preprocessing and the number of shortcuts for the Switzerland network on the Xeon machine. Preprocessing time includes both contracting the transfer graph and computing the shortcuts. The time required for the bucket-CH computation, which is independent of both parameters, is excluded.

preprocessing time. The only exception is the Germany network, for which we use a core degree of 20. This is because the share of the core-CH computation in the overall running time is significantly lower for this network because of its much larger size. Preprocessing results for the stop-to-stop variant on all four networks are listed in Table 2.

Table 2. Stop-to-Stop ULTRA Preprocessing Results

	Stuttgart	London	Switzerland	Germany
Core-CH time	1:45	0:19	1:09	20:16
Number of core vertices	25,631	23,860	33,219	313,351
Number of core edges	358,842	334,112	465,067	6,267,050
Shortcut computation time	4:27	18:01	8:54	8:01:25
Number of shortcuts	83,086	190,388	170,713	2,907,691
Bucket-CH time	2:13	0:11	0:43	14:49

Notes. All running times were measured on the Xeon machine and are displayed as (hh):mm:ss. The core-CH and bucket-CH computations were run sequentially, whereas the shortcut computation used all 16 cores.

5.2.2. ULTRA-TB Preprocessing. To evaluate the effectiveness of the event-to-event ULTRA shortcut computation, we compare it to the original TB preprocessing, using the transitively closed transfer graphs as input, and to a naive sequential approach, that is, using stop-to-stop ULTRA shortcuts as input for the TB preprocessing. An overview of the results is given in Table 3. The integrated ULTRA preprocessing drastically reduces the number of shortcuts compared with the sequential approach. This reduction ranges from a factor of 6 for the London network to more than 15 for Germany. Regarding computation time, the sequential approach using the optimized TB preprocessing proposed by Lehoux and Liodice (2020) is only marginally faster than the integrated approach. Overall, the integrated preprocessing is clearly preferable because it produces many fewer shortcuts with only a minor overhead in running time.

Remarkably, event-to-event ULTRA significantly outperforms the original TB preprocessing in both number of shortcuts and computation time despite operating on

Table 3. Number of Shortcuts and Preprocessing Times for Different TB Preprocessing Variants

	Stuttgart	London	Switzerland	Germany
Shortcuts (transitive)	7,387,445	50,242,519	31,507,264	458,826,534
Shortcuts (transitive, optimized)	7,387,586	50,240,558	31,507,543	458,763,050
Shortcuts (sequential)	19,361,708	53,179,082	65,485,696	1,195,573,925
Shortcuts (sequential, optimized)	19,361,129	53,181,238	65,484,976	1,195,509,797
Shortcuts (integrated)	1,973,321	8,576,120	6,938,012	77,515,291
Time (transitive)	9:30	1:42:35	1:01:54	73:43:07
Time (transitive, optimized)	0:37	13:12	4:41	2:55:06
Time (sequential)	4:41	18:43	9:40	8:57:46
Time (sequential, optimized)	4:37	18:28	9:24	8:22:37
Time (integrated)	4:42	20:43	9:40	8:37:49

Notes. “Transitive” refers to the original TB preprocessing on the transitively closed transfer graph. “Sequential” uses stop-to-stop ULTRA shortcuts as input for the TB preprocessing, whereas “integrated” uses event-to-event ULTRA shortcuts directly. “Optimized” refers to the improved TB preprocessing algorithm of Lehoux and Loiodice (2020). Running times were measured on the Xeon machine with 16 cores and are displayed as (hh):mm:ss.

an unrestricted transfer graph instead of a transitively closed one. This underscores that the original TB preprocessing was only designed for very limited transfer graphs and confirms the findings of Lehoux and Loiodice (2020) that it does not scale well for larger graphs. Compared with the optimized TB preprocessing, ULTRA is slower by a factor of about two to three on most networks. On the Stuttgart network, the slowdown is about eight. The difference is explained by the fact that Stuttgart is the only network in which the transitively closed transfer graph has fewer edges than the full transfer graph. Overall, the preprocessing results show that ULTRA is much more effective than the TB preprocessing at identifying necessary transfers at the cost of a somewhat higher preprocessing time.

5.2.3. Parallelization. The previous experiments used all 16 cores of the Xeon machine for the shortcut computation. To assess the impact of parallelization on the preprocessing time, we evaluate the running time of the stop-to-stop shortcut computation for different numbers of threads. Additionally, we compare running times of the Epyc machine, which has worse single-core performance but contains more cores. Running times on both machines are listed in Table 4. Overall, the parallelized shortcut computation achieves good speedups for all networks on both machines. For the Switzerland network, the maximal speedup is 13.5 on the Xeon machine and 74.6 on the Epyc machine. The speedup for the entire preprocessing phase, including the sequential core-CH and bucket-CH computation times on the Xeon machine, drops to 11.4 and 38.6, respectively. Independently of the network, we observe the smallest speedup when switching from 64 to 128 threads on the Epyc machine. In this case the speedup is most likely limited by the memory bandwidth.

The results are similar for the event-to-event variant. On the Switzerland network, the single-threaded performance on the Xeon machine is 2:07:00 for the sequential

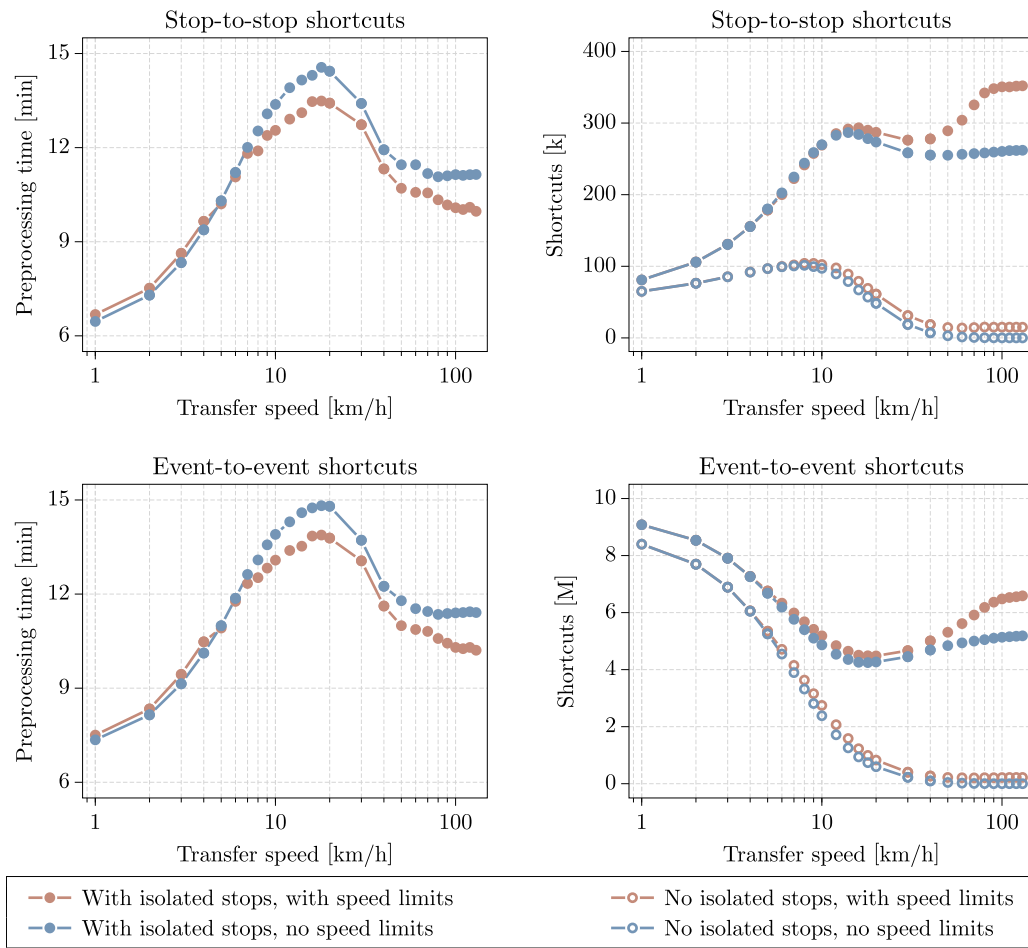
approach and 2:10:10 for the integrated approach. This corresponds to speedup factors of 13.1 and 13.5, respectively, which matches the speedups observed for the stop-to-stop variant and the TB preprocessing.

5.2.4. Transfer Speed. To test the impact of the transfer mode on the shortcut computation, we changed the transfer speed in the Switzerland network from 4.5 km/h to values between 1 and 140 km/h. We considered two ways of applying the transfer speed: in the first version, the speed on an edge is not allowed to exceed the speed limit given in the road network. This models fast transfer modes such as cars fairly realistically. In the second version, speed limits are ignored, and the same constant speed is assumed for every edge. This allows us to analyze to what extent the effects observed in the first version are caused by the speed limit data. Figure 4 reports the preprocessing times and number of shortcuts (both stop-to-stop and event-to-event) measured for each configuration. In all measurements, the preprocessing time remained below 15 minutes. The number of stop-to-stop shortcuts initially increases with the transfer speed

Table 4. Impact of Parallelization on the Running Time of the Stop-to-Stop ULTRA Shortcut Computation

Machine	Cores	Stuttgart	London	Switzerland	Germany
Xeon	1	59:28	4:00:31	2:00:29	100:02:46
	2	30:42	2:05:06	1:02:24	54:12:12
	4	15:49	1:06:24	32:17	29:02:18
	8	8:28	34:52	17:13	15:26:13
	16	4:27	18:01	8:54	8:01:25
Epyc	1	1:14:37	4:53:01	2:25:26	122:35:42
	2	40:38	2:43:33	1:21:57	72:42:27
	4	20:10	1:19:21	40:39	37:56:49
	8	10:03	39:54	20:23	19:11:35
	16	5:05	19:54	10:08	9:49:56
	32	2:37	10:08	5:11	4:57:06
	64	1:29	5:52	2:55	2:56:49
128	0:54	3:44	1:57	2:53:57	

Note. Running times are displayed as (hh):mm:ss.

Figure 4. (Color online) Impact of Transfer Speed on Preprocessing Time and Number of Shortcuts

Notes. All measurements for the Switzerland network with a core degree of 14 and a witness limit of zero. For the two lines at the bottom of the right plots, shortcuts were only added if the source and target of the candidate journey are connected by a path in the transfer graph.

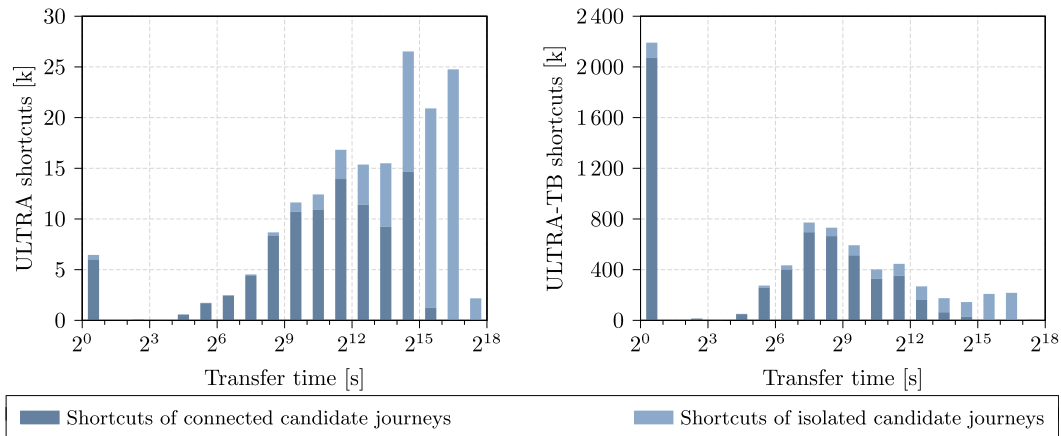
until it peaks at about 300,000 between 10 and 20 km/h (roughly the speed of a bicycle). In the event-to-event variant, the behavior is the opposite: the number of shortcuts is highest for 1 km/h and decreases from there. Above 20 km/h, both variants exhibit a slight increase in the number of shortcuts, which is more pronounced if speed limits are obeyed. Overall, the results show that ULTRA is practical for all transfer speeds in terms of both preprocessing time and the number of shortcuts.

To explain the difference in behavior between the two variants, consider how the transfer speed affects Pareto-optimal journeys. As the transfer mode becomes faster, it becomes increasingly feasible to cover large distances in the transfer graph quickly. This has two effects: on the one hand, more witnesses that require long initial or final transfers become feasible and start dominating slower candidates. Accordingly, the number of canonical candidates decreases from 409 million for 1 km/h to 114 million for 10 km/h. This explains the decrease in the number of event-to-event shortcuts. On the other hand, longer intermediate transfers between

trips also become feasible. This means that, although there are fewer canonical candidates for higher transfer speeds, the shortcuts that occur in them tend to cover larger distances in the transfer graph. The number of stop pairs within a certain distance of each other grows roughly quadratically with the distance. This explains why the number of stop-to-stop shortcuts rises for higher transfer speeds even as the number of event-to-event shortcuts declines.

Once the transfer speed becomes faster than public transit, the direct transfer from source to target dominates all other journeys, including all candidates. Accordingly, we should expect the number of shortcuts to eventually reach zero for very high transfer speeds. The reason this is not observed in our measurements is that not all stops in our network instances are reachable from each other in the transfer graph. Consider what happens in the shortcut computation for journeys between stops s and t that are isolated in the transfer graph. In this case, a direct transfer is not possible regardless of the transfer speed. In fact, unless there is

Figure 5. (Color online) Distribution of the ULTRA Shortcuts with Respect to Their Transfer Time for the Switzerland Network



Notes. The bar between 2^i and 2^{i-1} corresponds to the number of shortcuts with a transfer time in the interval $[2^i, 2^{i-1})$. An exception is the first bar, which also contains shortcuts with a transfer time of less than a second. The lower portion of each bar represents shortcuts for which the source and the target of the corresponding candidate journey are connected by a path in the transfer graph. Left: Shortcuts between stops as computed by the ULTRA preprocessing. Right: Shortcuts between stop events as computed by the ULTRA-TB preprocessing.

a route that serves both s and t , all $s-t$ journeys with at most two trips are candidates, and the shortcut computation adds shortcuts for the canonical ones. In our Switzerland network, 624 stops are isolated in the transfer graph, usually as a result of incomplete or imperfect data. If we omit shortcuts for candidates whose source and target stop are not connected in the transfer graph, the number of shortcuts behaves as expected: If speed limits are obeyed, a few shortcuts remain even for the highest transfer speed. If they are ignored, a direct transfer is always the fastest option, and thus, no shortcuts are required.

5.2.5. Shortcut Graph Structure. The stop-to-stop shortcut graph computed by ULTRA for Switzerland is structurally very different from the transitively closed transfer graph we created for comparison with pure public transit algorithms. This is already evidenced by the fact that the shortcut graph is much less dense, containing only 6% as many edges as the transitively closed graph. Furthermore, the transitive graph consists of many small, fully connected components with the largest one containing only 1,004 vertices. By contrast, the largest strongly connected component in the shortcut graph contains 10,891 vertices, which corresponds to 43% of all stops. Accordingly, a transitive closure of the shortcut graph would contain more than 100 million edges.

As Wagner and Zündorf (2017) observed when constructing a transitively closed transfer graph, preserving all transfers with a duration of up to a few minutes already leads to an average vertex degree of more than 100. This means algorithms that require a transitively closed transfer graph cannot be efficient and at the same time guarantee that long transfers are found.

Figure 5 (left side) shows the distribution of travel times for the ULTRA shortcuts. Note that the high number of shortcuts with travel time zero is caused by cases in which multiple stops model the same physical location. Most of the shortcuts have a travel time of more than nine minutes ($\approx 2^9$ seconds) and are, therefore, not contained in the transitive transfer graph. In fact, only 26,826 edges are shared between the two graphs, which constitute 1.0% of all transitive edges and 15.7% of all shortcuts. Altogether, this shows that the transitively closed graph fails to represent most of the relevant intermediate transfers at the expense of many superfluous ones.

As with the transfer speed experiment, Figure 5 distinguishes between shortcuts generated by candidates whose source and target stop are connected in the transfer graph and shortcuts in which source and target are isolated. We observe that most of the very long shortcuts are produced by candidates with isolated stops. To analyze how often longer shortcuts are required, we examine the distribution of the event-to-event shortcuts in Figure 5 (right side). Because stop events occur at a fixed point time, a stop-to-stop shortcut that is required at several times throughout the day corresponds to multiple event-to-event shortcuts. Thus, the number of event-to-event shortcuts with a certain travel time reflects more accurately how frequently these shortcuts are required. Approximately one third of all event-to-event shortcuts have a travel time of zero. Most of these connect pairs of trips at the same stop and, therefore, have no stop-to-stop counterpart. Among the remaining shortcuts, most have a travel time between 1 minute ($\approx 2^6$ s) and 34 minutes ($\approx 2^{11}$ s). This is in contrast to the stop-to-stop shortcuts, most of

Table 5. Query Performance for CSA, MCSA, and ULTRA-CSA

Network	Algorithm	Full graph	Scans, k		Time, ms		
			Connection	Edge	Init.	Scan	Total
Stuttgart	CSA*	○	52.6	281	0.0	1.4	1.4
	MCSA	●	113.7	238	10.1	6.4	16.5
	ULTRA-CSA	●	113.4	42	1.2	1.7	2.9
London	CSA*	○	83.9	663	0.0	3.0	3.0
	MCSA	●	58.2	182	4.6	4.5	9.1
	ULTRA-CSA	●	57.7	53	0.8	1.9	2.7
Switzerland	CSA*	○	135.2	787	0.1	4.9	4.9
	MCSA	●	88.2	241	8.4	8.1	16.4
	ULTRA-CSA	●	87.6	59	1.1	2.9	4.0
Germany	CSA*	○	2,587.8	6,351	1.3	144.3	145.5
	MCSA	●	1,662.1	3,191	142.8	195.2	338.0
	ULTRA-CSA	●	1,657.3	877	22.4	107.4	129.8

Notes. 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. Note that CSA (marked with *) only supports stop-to-stop queries with transitive transfers. The other two algorithms have been evaluated for vertex-to-vertex queries on the full graph.

which have a travel time of more than one hour ($\approx 2^{12}$ s). This shows that very long shortcuts are only rarely required. Furthermore, the fraction of shortcuts that are generated by candidates with isolated source and target is much lower in the event-to-event variant than in the stop-to-stop variant.

5.3. Queries

To evaluate the impact of ULTRA on the query performance, we test three public transit algorithms: CSA, RAPTOR, and TB. For CSA and RAPTOR, we compare our new ULTRA variant to the original algorithm on a transitively closed transfer graph and a multimodal variant with Dijkstra (1959) searches. For TB, no multimodal variants are proposed thus far. We, therefore, compare the original TB algorithm on a transitively closed transfer graph to ULTRA-TB with sequential and integrated preprocessing. Because we do not consider parallelized query algorithms, we use the Xeon machine (which has better single-core performance) for all following experiments.

Additional experiments evaluating the impact of the query distance on the running times can be found in Online Appendix C. Furthermore, a comparison with the HL-based approaches proposed by Phan and Viennot (2019) can be found in Online Appendix D. Because the original evaluation of the HL-based algorithms was based on a comparison of running times measured on different machines, we reimplemented all query algorithms and evaluated them on the same machine. In these experiments, we were only able to observe a marginal speedup of HL-RAPTOR compared with MR.

5.3.1. CSA Queries. Unlike the other algorithms we evaluate, CSA only supports optimizing arrival time as the sole criterion. Whereas profile CSA, a CSA variant for range queries, also supports optimizing the number

of trips as a second criterion, no bicriteria variant of basic CSA has been published thus far. We conducted preliminary experiments that showed a bicriteria variant of CSA is outperformed by RAPTOR. Therefore, we only consider single-criterion optimization for CSA. Unlike RAPTOR, no Dijkstra (1959) based multimodal variant of CSA has been proposed thus far. We, therefore, implemented a naive multimodal version of CSA, which we call multimodal CSA (MCSA), as a baseline for our comparison. This algorithm alternates connection scans with Dijkstra (1959) searches on the contracted core graph in a similar manner to MCR. Query times for all three CSA variants are reported in Table 5.

On all networks, ULTRA-CSA has a similar running time to CSA with transitively closed transfers. Caution has to be taken when comparing these running times because CSA does not support fully multimodal vertex-to-vertex queries and was, therefore, evaluated on a different set of stop-to-stop queries. Nonetheless, our experiments demonstrate that ULTRA enables CSA to use unrestricted transfers without performance loss. Compared with MCSA, the ULTRA approach is faster by about a factor of three to four on most networks and even more on the Stuttgart network, which has a particularly large transfer graph. By replacing the core-CH search of MCSA with a bucket-CH query, ULTRA speeds up the exploration of initial and final transfers by a factor of six to eight. The time required for the exploration of intermediate transfers is difficult to measure directly because it is interleaved with the individual connection scans. Nevertheless, we observe that using ULTRA shortcuts speeds up the connection scanning phase in its entirety by a factor of two to four compared with MCSA.

On all networks except Stuttgart, the multimodal variants scan significantly fewer connections than CSA on the transitively closed transfer graph. This is a direct

Table 6. Query Performance for RAPTOR, MR, and ULTRA-RAPTOR

Network	Algorithm	Full graph	Scans, k		Time, ms				
			Route	Edge	Init.	Coll.	Scan	Relax	Total
Stuttgart	RAPTOR*	○	19.8	756	0.2	1.6	2.1	2.1	5.9
	MR	●	35.6	687	12.3	5.2	5.2	11.1	33.5
	ULTRA-RAPTOR	●	37.9	105	1.4	3.5	3.5	1.0	9.6
London	RAPTOR*	○	4.4	2,573	0.3	1.1	2.2	5.4	8.9
	MR	●	5.0	500	6.4	1.9	2.7	7.0	18.0
	ULTRA-RAPTOR	●	5.4	179	1.2	1.5	2.3	1.2	6.2
Switzerland	RAPTOR*	○	26.2	2,115	0.4	2.4	5.0	5.0	12.8
	MR	●	33.0	731	10.6	4.8	7.2	11.7	34.1
	ULTRA-RAPTOR	●	35.9	177	1.6	3.3	6.2	1.4	12.5
Germany	RAPTOR*	○	472.9	26,420	7.0	102.6	120.4	74.2	304.2
	MR	●	541.4	12,359	154.2	187.5	153.5	236.2	731.4
	ULTRA-RAPTOR	●	599.7	3,165	33.0	144.0	151.7	33.3	362.1

Notes. Query times are divided into phases: initialization, including scanning initial transfers (Init.), collecting routes (Coll.), scanning routes (Scan), and relaxing transfers (Relax). All results are averaged over 10,000 random queries. Note that RAPTOR (marked with *) only supports stop-to-stop queries with transitive transfers, whereas the other three algorithms support vertex-to-vertex queries on the full graph and are evaluated accordingly.

result of the fact that fully multimodal journeys usually have a shorter travel time (Wagner and Zündorf 2017). Because CSA scans connections in chronological order, the number of scanned connections correlates directly with the earliest arrival time of the query. The Stuttgart network exhibits the opposite behavior because the transfer graph covers a much larger geographical area than the public transit network. Therefore, if the source and target are picked among all vertices instead of only stops, the average query distance increases and the search space becomes larger.

5.3.2. RAPTOR Queries. To evaluate RAPTOR, we used the MR variant of MCR as the multimodal baseline algorithm. The results of our comparison are shown in Table 6. The share of the overall running time spent exploring the transfer graph (i.e., the Init and Relax phases) is reduced from 50% to 75% for MR to 20% to

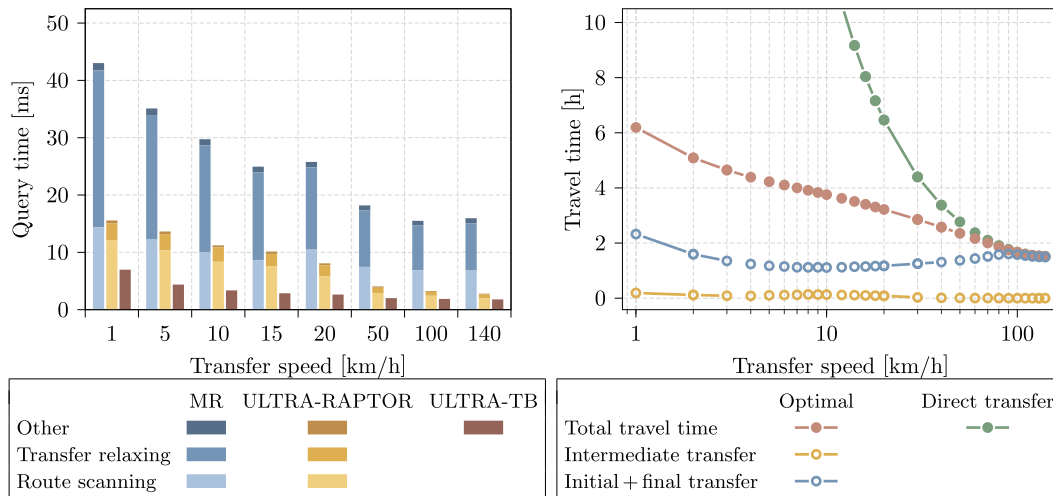
40% for ULTRA-RAPTOR. The Init phase exhibits the same speedup that was already observed for CSA. Because RAPTOR explores intermediate transfers in a separate phase, the impact of using ULTRA shortcuts can now be measured directly. Compared with the Dijkstra (1959) searches on the core graph performed by MR, exploring the transfer shortcuts is up to an order of magnitude faster. Overall, ULTRA-RAPTOR is two to three times as fast as MR and has a similar running time to RAPTOR with transitive transfers.

5.3.3. Trip-Based Queries. We continue with evaluating our improved ULTRA-TB query algorithm. Table 7 compares the query performance for ULTRA-TB with sequential and integrated preprocessing as well as the original TB query algorithm on the transitively closed transfer graph. ULTRA-TB with integrated preprocessing achieves significantly lower query times than the

Table 7. Query Performance for TB and ULTRA-TB (Sequential and Integrated)

Network	Algorithm	Full graph	Scans, k		Time, ms			
			Trip	Shortcut	B-CH	Initial	Scan	Total
Stuttgart	TB*	○	10.9	223	0.0	0.0	1.5	1.6
	ULTRA-TB (seq.)	●	25.1	1,417	1.2	1.0	5.8	7.9
	ULTRA-TB (int.)	●	15.3	112	1.1	0.8	1.7	3.6
London	TB*	○	15.3	830	0.0	0.0	3.7	3.7
	ULTRA-TB (seq.)	●	23.5	1,021	0.8	0.7	5.1	6.6
	ULTRA-TB (int.)	●	14.5	153	0.8	0.6	1.9	3.3
Switzerland	TB*	○	23.4	662	0.0	0.0	4.5	4.5
	ULTRA-TB (seq.)	●	34.9	1,620	1.0	1.2	7.1	9.3
	ULTRA-TB (int.)	●	19.5	138	1.0	1.0	2.2	4.3
Germany	TB*	○	389.1	16,331	0.0	0.0	106.6	106.9
	ULTRA-TB (seq.)	●	467.5	43,219	19.9	19.3	162.6	202.0
	ULTRA-TB (int.)	●	196.5	2,057	19.6	19.3	37.9	77.0

Notes. Query times are divided into phases: the bucket-CH query (B-CH), the initial transfer evaluation (Initial), and the scanning of trips (Scan). All results are averaged over 10,000 random queries. Note that TB (marked with *) only supports stop-to-stop queries with transitive transfers, whereas the other two algorithms support vertex-to-vertex queries on the full graph.

Figure 6. (Color online) Impact of Transfer Speed on Query and Travel Times

Notes. All measurements averaged over 10,000 random queries on the Switzerland network with a core degree of 14 and a witness limit of zero. Left: Query performance of MR, ULTRA-RAPTOR, and ULTRA-TB. Speed limits were obeyed during the construction of the transfer graph. For MR and ULTRA-RAPTOR, query times are divided into route collecting/scanning, transfer relaxation, and remaining time. Right: Total travel time and time spent on initial/final and intermediate transfers for the journey with minimal arrival time. The time required for a direct transfer from source to target is shown for reference. To allow for this comparison, we only chose random queries for which the source and target vertex are connected in the transfer graph.

state of the art. Depending on the network, it has a speedup of 2–5 over ULTRA-RAPTOR and 5–10 over MR, which was previously the fastest multimodal journey planning algorithm (cf. Table 6). As with RAPTOR and CSA, ULTRA-TB is able to match the query performance of the original TB algorithm despite solving a harder multimodal problem. Furthermore, ULTRA-TB achieves a similar performance to ULTRA-CSA despite optimizing an additional criterion.

Although ULTRA-TB with sequential preprocessing still outperforms other algorithms, it is slower than the integrated version by a factor of two. This is because the integrated preprocessing reduces the number of relaxed shortcuts by around an order of magnitude. This, in turn, reduces the overall search space and thereby the number of scanned trips. Overall, the trip scanning phase is sped up by a factor of three to four and only takes up around half of the overall query time. The remaining half is spent performing the bucket-CH searches and evaluating initial trips, both of which are unaffected by the number of transfer shortcuts.

5.3.4. Impact of Transfer Speed. In addition to overall query performance, we also measured how the query times of MR, ULTRA-RAPTOR, and ULTRA-TB are impacted by the transfer speed. Results are shown in Figure 6 (left side). The performance gains for ULTRA-RAPTOR compared with MR are similar for all transfer speeds and, in fact, slightly better for higher speeds. To explain this, we observe that the time required for the route scanning phase decreases as the transfer speed increases. This is because the total number of rounds

and, thus, the number of scanned routes decreases for higher transfer speeds. ULTRA-RAPTOR benefits more from this because the share of the route scanning phase in the overall running time is greater for ULTRA-RAPTOR than for MR. 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. ULTRA-TB achieves its highest speedup over the other two algorithms for medium transfer speeds, for which the number of event-to-event shortcuts is lowest. For very high transfer speeds, the bucket-CH search for the initial and final transfers starts to dominate the overall running time of both ULTRA-based algorithms. Accordingly, the speedup of ULTRA-TB over ULTRA-RAPTOR decreases.

The impact of the transfer speed on the travel time of the fastest journey is shown in Figure 6 (right side). As the transfer speed increases, the overall travel time decreases. The time that is spent on an initial or final transfer also decreases at first, but its share in the overall travel time becomes larger. From 10 km/h onward, transferring directly from source to target starts becoming the best option for more queries, and consequently, the time spent on initial and final transfers starts increasing. For very high transfer speeds, a direct transfer is almost always the fastest option. This matches our observation that intermediate transfers become useless for high transfer speeds unless the source and target are isolated from each other in the transfer graph. In contrast to initial and final transfers, intermediate transfers have a very small impact on the overall travel time, further demonstrating that long intermediate transfers are rarely needed.

6. Conclusion

We proposed ULTRA, a technique that accelerates the computation of Pareto-optimal journeys in a public transit network with an unrestricted transfer graph. The centerpiece of ULTRA is a preprocessing step that computes shortcuts that provably represent all necessary intermediate transfers. With parallelization, this step takes only a few minutes for metropolitan and midsized country networks and about three hours for Germany. The number of computed shortcuts is low regardless of the speed of the transfer mode. ULTRA shortcuts can be used without adjustments by any public transit algorithm that requires one-hop transfers. This enables the computation of unrestricted multimodal journeys without incurring the performance losses of existing multimodal algorithms. In particular, combining ULTRA with CSA yields the first efficient multimodal variant of CSA. To combine ULTRA with TB, we develop tailored versions of the ULTRA preprocessing and the TB query. The resulting ULTRA-TB algorithm outperforms MR, the fastest previously known multimodal algorithm for bicriteria optimization, by an order of magnitude.

Future work could involve extending ULTRA to support more optimization criteria, such as walking distance or cost, and multiple non-schedule based transportation modes. Furthermore, it would be interesting to adapt our approach to scenarios in which public transit vehicles can be delayed. Without adaptation, ULTRA can no longer guarantee optimal results in such a setting because journeys with delayed vehicles might require additional intermediate transfers that are not covered by the shortcut set. We suspect, however, that the underlying assumption of ULTRA (i.e., the set of required intermediate transfers is small) is still valid in a scenario with delays.

Acknowledgments

The authors thank Laurent Viennot and Tim Zeitz for helpful suggestions, and Sascha Witt for fruitful discussions about the trip-based routing algorithm. This manuscript is partially based on extended abstracts that appeared in the proceedings of the 27th Annual European Symposium on Algorithms (Baum et al. 2019) and the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (Sauer, Wagner, and Zündorf 2020) as well as the PhD thesis of one of its authors (Zündorf 2020).

References

Abraham I, Delling D, Goldberg AV, Werneck RF (2011) A hub-based labeling algorithm for shortest paths in road networks. Pardalos PM, Rebennack S, eds. *Proc. 10th Internat. Sympos. Experiment. Algorithms*, Lecture Notes in Computer Science, vol. 6630 (Springer, Berlin), 230–241.

Bast H, Storandt S (2014) Frequency-based search for public transit. Huang Y, Schneider M, Gertz M, Krumm J, Sankaranarayanan J, eds. *Proc. 22nd SIGSPATIAL Internat. Conf. Adv. Geographic*

Inform. Systems (Association for Computing Machinery, New York), 13–22.

Bast H, Hertel M, Storandt S (2016) Scalable transfer patterns. Goodrich M, Mitzenmacher M, eds. *Proc. 18th Workshop Algorithm Engrg. Experiments* (Society for Industrial and Applied Mathematics, Philadelphia), 15–29.

Bast H, Carlsson E, Eigenwillig A, Geisberger R, Harrelson C, Raychev V, Viger F (2010) Fast routing in very large public transportation networks using transfer patterns. de Berg M, Meyer U, eds. *Proc. 18th Annual Eur. Sympos. Algorithms*, Lecture Notes in Computer Science, vol. 6346 (Springer, Berlin), 290–301.

Bast H, Delling D, Goldberg A, Müller-Hannemann M, Pajor T, Sanders P, Wagner D, Werneck RF (2016) Route Planning in Transportation Networks. Kliemann L, Sanders P, eds. *Algorithm Engineering: Selected Results and Surveys*, Lecture Notes in Computer Science, vol. 9220 (Springer, Berlin), 19–80.

Bauer R, Delling D, Wagner D (2011) Experimental study of speed up techniques for timetable information systems. *Networks* 57(1): 38–52.

Bauer R, Delling D, Sanders P, Schieferdecker D, Schultes D, Wagner D (2010) Combining hierarchical and goal-directed speed-up techniques for Dijkstra’s algorithm. *J. Experiment. Algorithmics* 15:1–31.

Baum M, Buchhold V, Sauer J, Wagner D, Zündorf T (2019) Unlimited transfers for multi-modal route planning: An efficient solution. Bender MA, Svensson O, Herman G, eds. *Proc. 27th Annual Eur. Sympos. Algorithms*, Leibniz International Proceedings in Informatics, vol. 144 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl), 1–16.

Briem L, Buck S, Ebhart H, Mallig N, Strasser B, Vortisch P, Wagner D, Zündorf T (2017) Efficient traffic assignment for public transit networks. Iliopoulos CS, Pissis SP, Puglisi SJ, Raman R, eds. *Proc. 16th Internat. Sympos. Experiment. Algorithms*, Leibniz International Proceedings in Informatics, vol. 75 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl), 1–14.

Delling D, Dibbelt J, Pajor T (2019) Fast and exact public transit routing with restricted Pareto sets. Kobourov S, Meyerhenke H, eds. *Proc. 21st Workshop Algorithm Engrg. Experiments* (Society for Industrial and Applied Mathematics, Philadelphia), 54–65.

Delling D, Katz B, Pajor T (2012) Parallel computation of best connections in public transportation networks. *J. Experiment. Algorithmics* 17:1–26.

Delling D, Pajor T, Wagner D (2009) Engineering time-expanded graphs for faster timetable information. Ahuja RK, Möhring RH, Zorzi CD, eds. *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, Lecture Notes in Computer Science, vol. 5868 (Springer, Berlin), 182–206.

Delling D, Pajor T, Werneck RF (2015) Round-based public transit routing. *Transportation Sci.* 49(3):591–604.

Delling D, Dibbelt J, Pajor T, Werneck RF (2015) Public transit labeling. *Proc. 14th Internat. Sympos. Experiment. Algorithms*, Lecture Notes in Computer Science, vol. 9125 (Springer, Berlin), 273–285.

Delling D, Dibbelt J, Pajor T, Zündorf T (2017) Faster transit routing by hyper partitioning. *Proc. 17th Workshop Algorithmic Approaches Transportation Model. Optim. Systems*, OpenAccess Series in Informatics, vol. 59 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik), 1–14.

Delling D, Dibbelt J, Pajor T, Wagner D, Werneck RF (2013) Computing multimodal journeys in practice. Bonifaci V, Demetrescu C, Marchetti-Spaccamela A, eds. *Proc. 12th Internat. Sympos. Experiment. Algorithms*, Lecture Notes in Computer Science, vol. 7933 (Springer, Berlin), 260–271.

Dibbelt J, Pajor T, Wagner D (2015) User-constrained multimodal route planning. *J. Experiment. Algorithmics* 19:1–19.

Dibbelt J, Pajor T, Strasser B, Wagner D (2018) Connection scan algorithm. *J. Experiment. Algorithmics* 23:1–56.

- Dijkstra EW (1959) A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.
- Disser Y, Müller-Hannemann M, Schnee M (2008) Multi-criteria shortest paths in time-dependent train networks. McGeoch CC, ed. *Proc. Seventh Internat. Workshop Experiment. Efficient Algorithms*, Lecture Notes in Computer Science, vol. 5038 (Springer, Berlin), 347–361.
- Geisberger R, Sanders P, Schultes D, Vetter C (2012) Exact routing in large road networks using contraction hierarchies. *Transportation Sci.* 46(3):388–404.
- Giannakopoulou K, Paraskevopoulos A, Zaroliagis CD (2019) Multimodal dynamic journey-planning. *Algorithms* 12(10):1–16.
- Knopp S, Sanders P, Schultes D, Schulz F, Wagner D (2007) Computing many-to-many shortest paths using highway hierarchies. Applegate D, Brodal GS, eds. *Proc. Ninth Workshop Algorithm Engrg. Experiments* (Society for Industrial and Applied Mathematics, Philadelphia), 36–45.
- Lehoux V, Loiodice C (2020) Faster preprocessing for the trip-based public transit routing algorithm. Huisman D, Zaroliagis CD, eds. *Proc. 20th Sympos. Algorithmic Approaches Transportation Model. Optim. Systems*, OpenAccess Series in Informatics, vol. 85 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl), 1–12.
- Mallig N, Kagerbauer M, Vortisch P (2013) mobiTopp—A modular agent-based travel demand modelling framework. *Procedia Comput. Sci.* 19:854–859.
- Müller-Hannemann M, Schnee M (2007) Finding all attractive train connections by multi-criteria Pareto search. Geraets F, Kroon L, Schöbel A, Wagner D, Zaroliagis CD, eds. *Algorithmic Methods for Railway Optimization*, Lecture Notes in Computer Science, vol. 4359 (Springer, Berlin), 246–263.
- Phan DM, Viennot L (2019) Fast public transit routing with unrestricted walking through hub labeling. Kotsireas I, Pardalos P, Parsopoulos KE, Souravlias D, Tsokas A, eds. *Proc. Special Event Anal. Experiment. Algorithms*, Lecture Notes in Computer Science, vol. 11544 (Springer, Berlin), 237–247.
- Pyrga E, Schulz F, Wagner D, Zaroliagis CD (2008) Efficient models for timetable information in public transportation systems. *J. Experiment. Algorithmics* 12:1–39.
- Sauer J (2018) Faster public transit routing with unrestricted walking. Unpublished master's thesis, Karlsruhe Institute of Technology, Germany.
- Sauer J, Wagner D, Zündorf T (2020) Integrating ULTRA and trip-based routing. Huisman D, Zaroliagis CD, eds. *Proc. 20th Sympos. Algorithmic Approaches Transportation Model. Optim. Systems*, Open-Access Series in Informatics, vol. 85 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl), 1–15.
- Wagner D, Zündorf T (2017) Public transit routing with unrestricted walking. D'Angelo G, Dollevoet T, eds. *Proc. 17th Workshop Algorithmic Approaches Transportation Model. Optim. Systems*, OpenAccess Series in Informatics, vol. 59 (Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl), 1–14.
- Witt S (2015) Trip-based public transit routing. Bansal N, Finocchi I, eds. *Proc. 23rd Annual Eur. Sympos. Algorithms*, Lecture Notes in Computer Science, vol. 9294 (Springer, Berlin), 1025–1036.
- Zündorf T (2020) Multimodal journey planning and assignment in public transportation networks. Unpublished PhD thesis, Karlsruhe Institute of Technology, Germany.