T-REX: Fast and Dynamic Journey Planning for Continental-Scale Public Transit Networks

Jonas Sauer
Karlsruhe Institute of Technology, Germany

Patrick Steil
Karlsruhe Institute of Technology, Germany

Sascha Witt
Karlsruhe Institute of Technology, Germany

Abstract

We present T-REX (Transfer-Ranked EXploration), a new algorithm for journey planning in public transit networks on the country and continental scale. Our algorithm applies the principles of multi-level overlays to Trip-Based Public Transit Routing (TB). Using a multi-level partition of the network, T-REX identifies transfers between trips that are relevant for long-distance travel in a short precomputation phase. This information is then used to prune irrelevant local transfers during a query. Like other state-of-the-art algorithms, T-REX Pareto-optimizes arrival time and the number of used trips. T-REX dramatically outperforms previous overlay-based algorithms for three key reasons: (1) a better partition, (2) reducing the search space by focusing on transfers rather than trips, and (3) a redesigned query algorithm with improved memory efficiency and throughput. As a result, T-REX answers queries in less than 10 ms on a network of Europe, including local and long-distance transit. This constitutes a speedup of 20 compared to TB and 80 compared to algorithms without preprocessing. The memory footprint is moderate and the precomputation takes only two minutes, while real-time schedule updates can be incorporated in a few seconds. These properties make T-REX the first public transit journey planning algorithm that fulfills the requirements of interactive real-time applications on the continental scale.

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

Keywords and phrases Public transit routing, graph algorithms, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.9

Related Version This work is partially based on the master's thesis by Steil [58].
Full Version: `https://arxiv.org/abs/2605.18778`

Supplementary Material *Software (Source Code)*: `https://github.com/PatrickSteil/TREX` [52] archived at `swh:1:dir:098c3810dedbdf093d9ae741f0b3183c5fbd65e8`
Dataset (Dataset): `https://doi.org/10.5281/zenodo.19697732` [53]

Acknowledgements We thank Christian Schulz for providing us with the original implementation of KaHIP, Lars Gottesbüren and Nikolai Maas for advice on configuring Mt-KaHyPar, Ben Strasser for providing us with the original implementations of CSA and ACSA, and Patrick Brosi and Transitous [59] for supplying the Germany and Europe datasets used in our experiments.

1 Introduction

We encounter journey planning in public transport in our daily lives, whether on the morning commute to work or on vacation across countries. Interactive server-based applications, such as Google Maps, receive thousands of requests per second. Handling them in a timely fashion requires algorithms that produce accurate results within milliseconds. Whereas on road networks it is often sufficient to minimize travel time as a single objective, travelers using

public transport are typically also interested in comfort criteria, most importantly the number of different vehicles (trips) used. Therefore, it is common practice to compute Pareto-optimal journeys in terms of arrival time and number of trips, minimizing both criteria [4].

Because Dijkstra's algorithm [22] is too slow for interactive applications, *speedup techniques* precompute auxiliary information that is used to accelerate queries. Their performance is evaluated according to several metrics, including query time, memory consumption, precomputation time and the ability to handle real-time information (e.g., traffic or delays). For road networks, several techniques perform well in all four metrics, even on the continental scale [16, 21, 10]. On public transit networks, the situation is less satisfactory. Here, classic speedup techniques appear to be much less effective [7, 9, 2]. On the other hand, progress has been made by designing new query algorithms that are tailored to the special structure of public transit networks and modern CPU architectures [20, 18, 60]. On metropolitan networks, these algorithms are fast enough for interactive applications and support real-time updates, but their query speed is not sufficient for the continental scale. Moreover, integrating them with speedup techniques is challenging. Previous attempts have either caused the already low speedups to degrade further [15, 20, 1] or led to a blowup in the preprocessing effort, such that processing real-time updates becomes infeasible [3, 5, 61, 37].

Contribution. We present T-REX (Transfer-Ranked EXploration), a public transit journey planning algorithm that achieves interactive query times even on continent-sized networks, with moderate preprocessing time and space. T-REX extends Trip-Based Public Transit Routing (TB) [60], a cache-efficient modern query algorithm, by adapting the *multi-level overlays* (MLO) speedup technique [43, 16]. It uses a hierarchical nested partition of the network to determine which transfers between trips are required for long-distance travel. During a query, unimportant transfers are ignored. Three key improvements allow T-REX to achieve much faster query times than previous partitioning-based algorithms [57, 15, 20, 1]: (1) It obtains a better multi-level partition from a state-of-the-art graph partitioner. (2) The auxiliary information is computed for transfers rather than trips, which is more effective. (3) The query algorithm is carefully redesigned to exploit the pruning information in a cache-efficient manner.

We benchmark T-REX against state-of-the-art algorithms on networks ranging from metropolitan areas to a network covering most of Europe. With over 1.3 million stops and over 100 million events, our Europe instance is, to our knowledge, the largest network used in a scientific evaluation of journey planning algorithms so far. On this instance, T-REX answers queries in less than 10 ms, requiring two minutes of precomputation time and 28 GB of data (including 1.8 GB for the original timetable). This is a speedup of 20 over TB and 80 over RAPTOR [18], currently the fastest Pareto-optimal algorithm without substantial preprocessing. Moreover, T-REX can incorporate real-time updates quickly without re-running the entire preprocessing. Hence, it is the first public transit journey planning algorithm that satisfies all the criteria for an interactive application on continental-scale networks.

Outline. Section 2 reviews related work in journey planning. In Section 3, we introduce notation and outline the TB algorithm. We then present T-REX in Section 4 and conduct an experimental evaluation in Section 5. Finally, Section 6 summarizes our findings.

2 Related Work

We mainly consider algorithms that Pareto-optimize arrival time and the number of used trips. For a survey of routing techniques published prior to 2016, we refer the reader to [5].

Basic Algorithms. Traditional approaches model the timetable as a graph and apply variants of Dijkstra's algorithm [22, 11, 48, 50]. More recent algorithms achieve faster query times via cache-friendly scan operations. Connection Scan Algorithm (CSA) [20] optimizes the arrival time with a single sweep over all elementary connections, but does not support Pareto optimization. RAPTOR [18] is a modified breadth-first search that scans each discovered line in a single sweep. Trip-Based Public Transit Routing (TB) [60] speeds up this approach by precalculating a set of relevant transfers between trips. This allows the search to bypass the step of finding the earliest reachable trip of a line.

Speedup Techniques. Classic speedup techniques for road networks include goal-directed approaches, such as Arc-Flags [42, 45, 47] and A* search [41, 32], and hierarchical techniques. Among the latter, *multi-level overlays* (MLO) [43] recursively select a set of "important" vertices and add distance-preserving *shortcuts* between them to create a hierarchy of *overlays*. Intuitively, when the search is "far away" from the source and the target, it moves upwards in the hierarchy and bypasses "unimportant" vertices. MLOs can be built via vertex contraction [43] or a multi-level partition [44, 16]. In the latter case, the "important" vertices are the boundary vertices of the cells in the partition. An evolution of MLO are Contraction Hierarchies (CH) [25], which rely on a complete ranking of the vertices by "importance".

Because road networks have small and balanced separators [55, 35, 40], multilevel partitions can be found without considering the edge weights. This enables *customizable* techniques, in which the preprocessing is mostly metric-independent and the edge weights are incorporated in a second, much faster step. Thus, metric updates, such as real-time traffic, can be processed within seconds. Customizable Route Planning (CRP) [16] and Customizable Contraction Hierarchies (CCH) [21, 10] extend MLO and CH, respectively, to the customizable setting and speed up Dijkstra's algorithm by up to four orders of magnitude.

Applications. Traditionally, speedup techniques have seen limited success in public transit [7, 9, 2], with speedups often in the low single digits [23, 9, 17]. A recent breakthrough is FLASH-TB [37], which adapts Arc-Flags to TB. The network is partitioned into k cells, and each transfer is labeled with a k-bit vector, where bit i indicates whether the transfer is required to reach cell i. The query only relaxes transfers that are flagged for the target cell. Previous adaptations of Arc-Flags [9, 17] suffered from the presence of many equivalent journeys and the inability to break ties between them in a consistent manner. To resolve this issue, FLASH-TB is forced to use a preprocessing step with a quadratic runtime in the network size, which is infeasible for continental-sized networks. However, it achieves speedups of up to 500 over TB, demonstrating that classic speedup techniques can be effective.

A challenge in applying hierarchical techniques to public transit networks is that they exhibit a weaker hierarchy than road networks [2, 20], particularly at the local transit level, which makes it harder to find small separators. Node contraction is also less successful due to the presence of high-degree nodes (e.g., large train stations) [9]. An early application of MLO to graph-based models that optimized only arrival time achieved a speedup of 10 [57]. Connection Scan Accelerated (ACSA)[20], which applies MLO to CSA, also only optimizes the arrival time. For each cell in a multi-level partition, ACSA identifies connections that are required to traverse it. During a query, it collects, sorts and scans only the relevant connections.

On country-scale networks, this achieves a speedup of 7 over CSA. HypRAPTOR [15] and HypTB [1] apply overlay-based ideas to RAPTOR and TB, respectively, using a hypergraph representation of the network with lines as vertices and stops as hyperedges. Sadly, these algorithms achieve a speedup of at most two over their respective baselines.

Expensive Preprocessing. A family of algorithms with extremely fast query times answers every possible query in advance and compresses the obtained information. This requires quadratic precomputation effort, which is infeasible for continental-sized networks. The first such algorithm is Transfer Patterns (TP) [3], which builds search graphs called transfer patterns. These require enormous space, so Scalable Transfer Patterns [5] split them into "local" and "global" patterns. The query algorithm must re-assemble these, making it much slower than TP and only slightly faster than TB. Heuristic variants of (Scalable) TP reduce the precomputation time at the expense of provable optimality. Newer representatives include TB using Condensed Search Trees [61] and FLASH-TB. Unlike the other algorithms, the memory consumption of FLASH-TB is configurable, and query times remain extremely fast even with moderate space. A different technique with sub-millisecond query times is Public Transit Labeling (PTL) [14], which adapts Hub Labeling [13]. However, for Pareto-optimal queries, the required space is already in the tens of gigabytes on metropolitan networks.

Dynamic Settings. Updates such as delays and trip cancellations are easy to incorporate into algorithms without heavy preprocessing, including graph-based models [49, 12, 26], RAPTOR and CSA. For TB, the precomputed transfer set can be updated quickly without recomputing the entire set [62]. Algorithms with heavy preprocessing generally do not support real-time updates. Experimentally, TP has been shown to be fairly robust with respect to delays [6], but optimality cannot be guaranteed. Moreover, this robustness may not extend to blocked tracks or closed stations, which are likely to require substantial changes to the stored patterns. For PTL, a single delayed connection can require minutes to process [19].

3 Fundamentals

This section introduces notation and definitions and gives an overview of the TB algorithm.

Network. A *public transit network* is a 5-tuple $(\mathcal{S}, \mathcal{F}, \mathcal{E}, \mathcal{T}, \mathcal{L})$ consisting of a set of stops $\mathcal{S}$, footpaths $\mathcal{F} \subseteq \mathcal{S} \times \mathcal{S}$, events $\mathcal{E}$, trips $\mathcal{T}$ and lines $\mathcal{L}$. A *stop* $p \in \mathcal{S}$ is a location at which passengers can enter and exit vehicles. A *trip* is a sequence $T = \langle \varepsilon_1, \ldots, \varepsilon_k \rangle$ of *stop events* performed by a single vehicle, with $|T| = k$. The i-th stop event of T is denoted by $T[i]$, its visited stop by $p(T[i])$, its arrival time by $\tau_{\text{arr}}(T[i])$, and its departure time by $\tau_{\text{dep}}(T[i])$, measured in a resolution of seconds. For $1 \leq i < j \leq |T|$, we denote by $T[i, j]$ the trip segment from $T[i]$ to $T[j]$. If $j = i + 1$, we call the segment a *connection*. Trips are grouped into *lines* $\mathcal{L}$ such that all trips of the same line visit the same stop sequence and admit a total ordering $\prec$, where $T \prec T'$ if $\tau_{\text{arr}}(T[i]) < \tau_{\text{arr}}(T'[i])$ and $\tau_{\text{dep}}(T[i]) < \tau_{\text{dep}}(T'[i])$ for every $1 \leq i \leq |T|$. We write $T \preceq T'$ as shorthand for $T \prec T' \vee T = T'$. A *footpath* $(p, q) \in \mathcal{F}$ connects two stops $p, q \in \mathcal{S}$ and can be traversed in time $\tau_{\text{f}}(p, q)$. We assume that the footpath (p, p) exists for every stop p with $\tau_{\text{f}}(p, p) = 0$, and that $\tau_{\text{f}}(p, q) = \infty$ if $(p, q) \notin \mathcal{F}$. Furthermore, we require that $\mathcal{F}$ is transitively closed and fulfills the triangle inequality.

For simplicity of exposition, we do not explicitly model a minimum change time between trips at the same stop. Following [54], a *buffer time* can instead be incorporated implicitly by subtracting it from the departure time of the respective stop events.

Journeys. A *transfer* is a tuple $\mathfrak{t} = (T_a[j], T_b[i]) \in \mathcal{E} \times \mathcal{E}$ such that $\mathcal{F}$ contains the footpath $(p(T_a[j]), p(T_b[i]))$ and the traversal time is short enough to reach $T_b[i]$ from $T_a[j]$, i.e., $\tau_{\text{arr}}(T_a[j]) + \tau_{\text{f}}(p(T_a[j]), p(T_b[i])) \leq \tau_{\text{dep}}(T_b[i])$. Given a source event $\varepsilon_{\text{s}} \in \mathcal{E}$ and a target event $\varepsilon_{\text{t}} \in \mathcal{E}$, an ε_{s}-ε_{t}-*journey* is a sequence $J = \langle T_1[i_1, j_1], \ldots, T_k[i_k, j_k] \rangle$ of trip segments that describes how a passenger travels from $\varepsilon_{\text{s}} = T_1[i_1]$ to $\varepsilon_{\text{t}} = T_k[j_k]$, such that the transfer $(T_n[j_n], T_{n+1}[i_{n+1}])$ exists for $1 \leq n < k$. Given a source stop p_s and a target stop p_t, a p_s-p_t-journey is a sequence $J = \langle f_0, T_1[i_1, j_1], \ldots, T_k[i_k, j_k], f_{k+1} \rangle$ with initial footpath $f_0 = (p_s, p(T_1[i_1])) \in \mathcal{F}$, final footpath $f_{k+1} = (p(T_k[j_k]), p_t) \in \mathcal{F}$ and transfers as above. We omit a footpath from the notation if it has the form (p, p), i.e., connects a stop to itself. We denote the number of trip segments in J by $|J|$. The departure time of J is $\tau_{\text{dep}}(J) \coloneqq \tau_{\text{dep}}(T_1[i_1]) - \tau_{\text{f}}(p_s, p(T_1[i_1]))$ and the arrival time is $\tau_{\text{arr}}(J) \coloneqq \tau_{\text{arr}}(T_k[j_k]) + \tau_{\text{f}}(p(T_k[j_k]), p_t)$.

Given a journey $J = \langle T_1[i_1, j_1], \ldots, T_k[i_k, j_k] \rangle$, a *proper subjourney* is a subsequence of J, i.e., it includes trip segments in their entirety, whereas an *event subjourney* may start and end in the middle of a trip segment. For indices $1 \leq m < n \leq k$, we denote the proper subjourney from the m-th to the n-th trip segment by $J[m, n] = \langle T_m[i_m, j_m], \ldots, T_n[i_n, j_n] \rangle$. For indices $i_m \leq \ell_m < j_m$ and $i_n < \ell_n \leq j_n$, we denote the event subjourney from $T_m[\ell_m]$ to $T_n[\ell_n]$ by $J[T_m[\ell_m], T_n[\ell_n]] = \langle T_m[\ell_m, j_m], \ldots, T_n[i_n, \ell_n] \rangle$. A (proper or event) subjourney of J is a *prefix* if it starts with $T_1[i_1]$ and a *suffix* if it ends with $T_k[j_k]$.

Queries. A *query* $Q = (p_s, p_t, \tau_{\text{dep}})$ consists of source and target stops p_s, p_t and a departure time τ_{dep}. A p_s-p_t-journey J is *feasible* for Q if $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}$. A journey J *dominates* another journey J' if its *cost vector* $c(J) = (\tau_{\text{arr}}(J), |J|)$ is strictly smaller than $c(J')$ in at least one criterion and not larger in the other. A feasible journey J and its cost vector are *Pareto-optimal* if no other feasible journey dominates J. The *Pareto front* is the set of all Pareto-optimal cost vectors. A *representative set* consists of one journey for each cost vector in the Pareto front. The objective of Q is to report the Pareto front and a representative set.

Partitioning. For $k \in \mathbb{N}^+$, a *k-partition* of a set O is a family $\mathcal{C}(O) = \{C_0, C_1, \ldots, C_{k-1}\}$ of sets such that $\bigcup_{0 \leq i < k} C_i = O$ and $C_i \cap C_j = \emptyset$ holds for all $0 \leq i, j < k$. Each set $C \in \mathcal{C}(O)$ is called a *cell*. If $k = 2$, we refer to $\mathcal{C}(O)$ as a *bipartition* of O.

▶ **Definition 1.** *A **nested bipartition** $\mathcal{C}_K(G)$ of a $G = (V, E)$ into $K \in \mathbb{N}^+$ levels is a family $\{\mathcal{C}^\ell(V)\}_{\ell=0}^{K}$, where each $\mathcal{C}^\ell(V)$ is a $2^{K-\ell}$-partition of V, i.e., $\mathcal{C}^\ell(V) = \{C_0^\ell, C_1^\ell, \ldots, C_{2^{K-\ell}-1}^\ell\}$, and for each $1 \leq \ell \leq K$ and $0 \leq i < 2^{K-\ell}$, the set $\{C_{2i}^{\ell-1}, C_{2i+1}^{\ell-1}\}$ is a bipartition of C_i^ℓ.*

For $u, v \in V$, the *lowest common level* $\text{LCL}(u, v)$ is the lowest level ℓ on which u and v are in the same cell. Note that all vertices are in the same cell on level K. For a vertex u contained in cell C_i^ℓ on level ℓ, we denote the *cell ID* of u on level ℓ by $c_{\text{ID}}(u, \ell) = i$.

TB

We give a description of the TB algorithm (see [60] for further details).

Transfer Precomputation. This step computes a set $\mathfrak{T} \subseteq \mathcal{E} \times \mathcal{E}$ of transfers between pairs of stop events that is sufficient for answering all queries optimally. This is done in three steps: First, a superset of the required transfers is generated. Then, two pruning rules are applied to remove irrelevant transfers. All three steps are trivial to parallelize, as trips are processed independently.

Algorithm 1 TB `Scan` method for round n and target stop p_t.

```
1  for each T[j,k] ∈ Q_n do
2      for i from j to k do
3          if τ_arr(T[i]) ≥ τ_min then break                    // target pruning
4          if τ_arr(T[i]) + τ_f(p(T[i]), p_t) < τ_min then      // create cost vector
5              τ_min ← τ_arr(T[i]) + τ_f(p(T[i]), p_t)
6              L ← L ∪ {(τ_min, n)}, removing dominated cost vectors
7  for each T[j,k] ∈ Q_n do
8      for i from j to k do
9          if τ_arr(T[i]) ≥ τ_min then k ← i − 1            // target pruning
10 for each T[j,k] ∈ Q_n do                                  // relax transfers
11     for each t = (T[i], T'[i']) ∈ 𝔗 with j ≤ i ≤ k do Enqueue(t, Q_{n+1})
```

For the initial generation step, Witt [60] originally proposed a simple scheme. It generates all transfers $(T_a[j], T_b[i])$ such that $1 < j \leq |T_a|$, $1 \leq u < |T_b|$, and T_b is the earliest trip of its line that can be entered at $p(T_b[i])$, i.e., the earliest trip that fulfills

$$\tau_{\text{arr}}\left(T_a[j]\right) + \tau_{\text{f}}(p(T_a[j]), p(T_b[i])) \leq \tau_{\text{dep}}\left(T_b[i]\right).$$

The only situation in which the transfer is directly discarded is if both trips belong to the same line and staying seated in T_a is preferable, i.e., if $T_a \preceq T_b$ and $j \leq i$. In our implementation, we use the alternative transfer generation scheme proposed by Lehoux and Loiodice [46], which improves the preprocessing time by discarding some superfluous transfers before they are generated. If a transfer $(T_a[i], T_b[j])$ exists, then all other transfers of the form $(T_a[k], T_c[\ell])$ with $k \leq i$, $\ell \geq j$ and $T_c \succeq T_b$ are not generated.

After the initial transfer set has been generated, the *U-turn rule* prunes transfers $\mathfrak{t} = (T_a[i], T_b[i])$ with $p(T_a[j-1]) = p(T_b[i+1])$ and $(T_a[j-1], T_b[i+1]) \in \mathfrak{T}$. Finally, the *latest-exit* rule [37] removes transfers that are "dominated" by others starting from the same trip. For a transfer $\mathfrak{t} = (T_a[j], T_b[i])$, consider the journey $J = \langle T_a[k, j], T_b[i, l] \rangle$. The transfer can be pruned if there exists another journey $J' = \langle T_a[k, j'], T_c[i', l'], f \rangle$, with $j < j'$ and $f = (p\left(T_c[l']\right), p\left(T_b[l]\right))$ which has a better (or equal) arrival time at $p\left(T_b[l]\right)$.

Query. The query algorithm operates in rounds, where round n finds journeys with exactly n trips by scanning trip segments collected in a FIFO queue Q_n during the previous round. For each trip T, the *reached index* $R\left(T\right)$ is the smallest i such that for each stop event $T[j]$ with $i \leq j \leq |T|$, an event $T'[j]$ with $T' \preceq T$ has already been scanned. The algorithm also maintains the tentative Pareto front $\mathcal{L}$ and the earliest known arrival time τ_{min} at p_t.

The main `Scan` operation (see Algorithm 1) iterates over all trip segments $T[j, k]$ in the queue Q_n three times. For each event $T[i]$ with $j \leq i \leq k$, the first loop checks whether it is possible to reach p_t by exiting the trip at $T[i]$ and taking a final footpath. If this improves τ_{min} (line 4), a corresponding cost vector is added to $\mathcal{L}$. The found journeys are represented implicitly using parent pointers, which are unpacked at the end of the query. *Target pruning* is applied during the loop: If $\tau_{\text{arr}}(T[i]) \geq \tau_{\text{min}}$ (line 3), then the rest of the trip segment is skipped because all subsequent events have even later arrival times. The second loop applies target pruning again because the first loop may have reduced τ_{min} further. If the arrival time of a stop event $T[i]$ exceeds τ_{min}, then the subsegment $T[i, k]$ is cut off.

The third loop relaxes the outgoing transfers of all events $T[i]$ with $j \leq i \leq k$. These transfers are stored contiguously in memory, which allows them to be scanned in a single for-loop. For each relaxed transfer $(T_a[i], T_b[j])$, the `Enqueue` method (pseudocode in Algorithm 2)

Algorithm 2 TB enqueuing operation.

```
1 Procedure Enqueue((T_a[i], T_b[j]), Q_{n+1})
2     if R(T_b) ≤ j + 1 then return                    // trip segment already reached
3     Q_{n+1} ← Q_{n+1} ∪ {T_b[j + 1, R(T_b) − 1]}
4     for each T'_b ⪰ T_b do                           // update reached index
5         if R(T'_b) ≤ j + 1 then break
6         R(T'_b) ← j + 1
```

is called. The first index at which T_b can be exited is $j+1$. If the reached index $R(T_b)$ is greater than $j+1$, then the newly reached trip segment $T_b[j+1, R(T_b)]$ is enqueued. Afterwards, the reached index of T_b and all later trips of the same line is updated.

To fill the first queue Q_1, the query algorithm collects all lines that depart from p_s or any stop reachable from p_s via a footpath. For each of these lines, the algorithm finds the earliest reachable trip and adds the corresponding trip segment to Q_1 via the `Enqueue` operation.

Journey Unpacking. The found journeys are represented implicitly using parent pointers. Let $T_{n-1}[j_{n-1}, k_{n-1}] \in Q_{n-1}$ be a trip segment that is scanned in round $n-1$, and let $(T_{n-1}[i_{n-1}], T_n[j_n])$ be an outgoing transfer that is relaxed, and let $T_n[j_n, k_n]$ be the trip segment that is added to Q_n as a result. Then $T_n[j_n, k_n]$ stores a pointer to $T_{n-1}[j_{n-1}, k_{n-1}] \in Q_{n-1}$. The index i_{n-1} at which T_{n-1} is not stored explicitly but reconstructed during journey unpacking by rescanning the outgoing transfers of $T_{n-1}[j_{n-1}, k_{n-1}]$ and choosing the first one that leads to $T_n[j_n]$. This is faster than tracking this value explicitly during the query. Along with each cost vector in $\mathcal{L}$, the algorithm also stores a pointer to the trip segment from which the final footpath was relaxed. Using this information, the journey can be reconstructed by iteratively following the chain of parent pointers to the first trip segment in Q_1.

4 T-REX (Transfer-Ranked EXploration)

We now describe T-REX in detail. The first step is to compute a nested bipartition $\mathcal{C}_K$ of the timetable and, using the TB preprocessing, the set $\mathfrak{T}$ of transfers. Then, the algorithm computes the *rank* $r(\mathfrak{t})$ of each transfer $\mathfrak{t}$, which is the lowest level ℓ such that $\mathfrak{t}$ is not required to traverse the cell containing $\mathfrak{t}$ on level ℓ. Following the terminology of three-phase customizable techniques for road networks (e.g., CRP, CCH), we call this phase the *customization*. The ranks are used for pruning during a query: Let p be the current stop and ℓ the highest level such that the cell of p does not contain the source or target stop. Then the cell must be traversed, so the search can ignore transfers with rank ℓ and below.

T-REX offers two conceptual improvements over previous partition-based approaches. Firstly, it computes pruning information for transfers rather than stop events or connections because this leads to strictly stronger pruning [37]: if a transfer $\mathfrak{t} = (T_a[j], T_b[i])$ is relevant, then the involved events $T_a[j]$ and $T_b[i]$ are both relevant, but the converse is not necessarily true. Secondly, we carefully redesign the query algorithm to ensure that the search space reduction translates into faster query times. This is challenging in scan-based algorithms (such as TB) because they rely on branch prediction and cache locality, which are weakened by extra pruning steps. This was one of the issues faced by ACSA [20] and HypRAPTOR [15].

Partitioning. As in ACSA [20], we represent the network as a *compact layout graph* $G_L = (V_L, E_L)$, in which vertices correspond to stops, and an edge (u, v) exists if there is at least one connection from an event at u to an event at v. Stops that are connected by a footpath are contracted into a single vertex to ensure that transfers do not span multiple cells. Note that this contraction step imposes a practical limit on the size of the footpath set; this is discussed further in Section 6. Each vertex is weighted with the number of represented stops, and each edge with the number of represented connections. The nested bipartition $\mathcal{C}_K$ is then computed heuristically on G_L, aiming to minimize the total weight of the cut edges, i.e., edges that cross cell boundaries, while enforcing a maximum imbalance between the total vertex weights of the cells.

Query. In its most basic form, the T-REX query algorithm is identical to the TB query algorithm, but with one additional pruning step: Let p_s be the source and p_t the target stop. A transfer $\mathfrak{t}$ starting from a stop p is only relaxed if it passes the *LCL test*, which is

$$r(\mathfrak{t}) \geq \min\{\mathrm{LCL}(p, p_s), \mathrm{LCL}(p, p_t)\}. \tag{1}$$

This test is performed at the start of the `Enqueue` method.

This basic approach spends most of its time performing unsuccessful LCL tests for irrelevant transfers. We therefore redesign the query algorithm to avoid as many LCL tests as possible. Consider the scan of a stop event $T[i]$ at the stop $p = p(T[i])$. The relevant level for the LCL test of all outgoing transfers of $T[i]$ is $\ell = \min\{\mathrm{LCL}(p, p_s), \mathrm{LCL}(p, p_t)\}$. Obviously, recalculating ℓ for every transfer is wasteful, but we can go one step further. On level $\ell - 1$, the stop p is in a different cell than both p_s and p_t. If the next stop event $T[i+1]$ is in the same cell as p on level $\ell - 1$, then the relevant level for the LCL test remains the same. To exploit this, we add a new `Split` procedure at the start of each round, which splits the enqueued trip segments into subsegments along cell boundaries (see Algorithm 3). Then the relevant level for the LCL test remains the same within each subsegment. We do this in advance rather than on-the-fly to reduce the cache load during the `Scan` operation.

For every stop event $T[i]$ and every level ℓ, we precalculate a value $\mathrm{succ}(T[i], \ell)$, which is the smallest index j with $i \leq j \leq |T|$ such that $c_{\mathrm{ID}}(p(T[i]), \ell) \neq c_{\mathrm{ID}}(p(T[j]), \ell)$, or $|T| + 1$ if there is no such index. For each round n, we maintain an additional FIFO queue $\tilde{Q}_n$, which stores the enqueued trip segments before they are split. Each queue element is now a tuple $(T[j, k], \ell)$, where ℓ is the relevant level for the LCL tests of $T[j]$. To split this segment, the algorithm looks up $i = \mathrm{succ}(T[j], \max\{\ell - 1, 0\})$. Then the first subsegment is $T[j, k']$ with $k' = \min\{i - 1, k\}$, which is added to Q_n with level ℓ. If $i \leq k$, then the algorithm performs the LCL test for $T[i]$ and recurses on the remaining trip segment $T[i, k]$.

To skip irrelevant transfers during the `Scan` operation (see Algorithm 4), we precalculate for each level ℓ a *transfer overlay* $\mathfrak{T}_\ell$, which contains all transfers with rank at least ℓ. For a queue element $(T[j, k], \ell)$, the third loop of the `Scan` operation now relaxes the outgoing transfers in $\mathfrak{T}_\ell$. No LCL test is performed because all relaxed transfers would pass it. Instead, the relevant level ℓ is passed to `Enqueue` and added to trip segments that are inserted into $\tilde{Q}_{n+1}$. The first loop of the `Scan` operation, which relaxes outgoing footpaths to p_t, only needs to be performed for trip segments that are in the same cell as p_t on level 0. To exploit this, we maintain a third queue Q_n^t and only run the loop for the elements in this queue. When a subsegment $(T[j', k'], \ell)$ is created during the `Split` operation, we check whether the stop of $T[j']$ is in the same level-0 cell as p_t. If so, then this implies $\ell = 0$ and therefore the entire subsegment is in the same level-0 cell. Hence, the subsegment is added to Q_n^t.

Outside of the level-0 target cell, the only part of the `Scan` operation that scans the individual stop events of a trip segment $T[j, k]$ is now the second loop, which performs target pruning. Especially on higher levels, many trip segments have fewer outgoing transfers than

Algorithm 3 Efficient T-REX trip splitting operation.

```
Procedure Split(Q̃_n)
    Q_n, Q_n^t ← ∅
    for each (T[j,k], l) ∈ Q̃_n do
        j' ← j
        l' ← l
        while true do
            i ← succ(T[j'], max{l − 1, 0})
            k' ← min{i − 1, k}
            Q_n ← Q_n ∪ {(T[j', k'], l)}
            // Add to target cell queue
            if c_ID(p(T[j'])) = c_ID(p_t) then Q_n^t ← Q_n^t ∪ {T[j', k']}
            if i > k then break
            j' ← i
            p ← p(T[j'])
            l' ← min{LCL(p, p_s), LCL(p, p_t)}
```

Algorithm 4 T-REX `Scan` method for round n and target stop p_t. Changes in blue.

```
for each T[j,k] ∈ Q_n^t do
    for i from j to k do
        if τ_arr(T[i]) ≥ τ_min then break                    // target pruning
        if τ_arr(T[i]) + τ_f(p(T[i]), p_t) < τ_min then      // create cost vector
            τ_min ← τ_arr(T[i]) + τ_f(p(T[i]), p_t)
            L ← L ∪ {(τ_min, n)}, removing dominated cost vectors
for each (T[j,k], l) ∈ Q_n do
    if τ_arr(T[j]) ≥ τ_min then k ← j − 1                    // lazy target pruning
for each (T[j,k], l) ∈ Q_n do                                // relax transfers
    for each t = (T[i], T'[i']) ∈ 𝔗_l with j ≤ i ≤ k do Enqueue(t, Q̃_{n+1}, l)
```

stop events, so this step costs more time than it saves. We therefore make the target pruning check lazier: if the arrival time of the first stop event $T[j]$ exceeds τ_{min}, then the entire trip segment is skipped. As a result, the `Scan` operation now essentially treats each trip segment as a single node, whose outgoing transfers are relaxed in a single sweep.

Unlike Dijkstra-based MLO algorithms [44, 16], we do not add shortcuts other than the precomputed TB transfers. Although shortcuts that bypass trip segments might reduce the search space, preliminary experiments indicates that the potential for savings is low. This is because most journeys use only a few trips and the average number of outgoing transfers per trip is high (56 on our Europe instance). Moreover, the `Enqueue` method would become less efficient. When relaxing a shortcut that skips x trip segments, the discovered trip segment must be inserted into the queue Q_{n+x-1}. Hence, the correct queue must be fetched for each shortcut, which reduces cache locality.

Customization. For each journey J found by TB and each transfer $\mathfrak{t}$ in J, the customization must ensure that Equation (1) is fulfilled. We observe the following: At every level ℓ such that p_s and p_t are not in the same cell in the nested bipartition $\mathcal{C}_K$, we can decompose J into event subjourneys along cell boundaries. For each subjourney that traverses a cell, the ranks

of its transfers must be at least $\ell + 1$. For a cell C, a stop event $T[i]$ is an *incoming border event* (IBE) if $T[i]$ does not lie in C, but $T[i+1]$ does. Analogously, $T[i]$ is an *outgoing border event* (OBE) if $T[i]$ belongs to C, but $T[i+1]$ does not. We find journeys that traverse C by performing a search from each IBE $T[i]$ of cell C to all OBEs of C with a special variant of TB, which we call *Event-TB*. Because all journeys must start with $T_a[i]$, the initial queue Q_1 is initialized with only the trip segment $T_a[i, |T_a|]$. Target pruning is omitted, and instead of tracking a tentative Pareto front, the first loop over the queue Q_n collects all reached OBEs in a set $\mathcal{O}$. The search is restricted to C by cutting off each trip segment after the first encountered OBE. Using this search, the customization operates in a bottom-up fashion: For every level ℓ from 0 to $K-1$, the IBEs of all cells are processed in parallel. For each IBE $T[i]$ of a cell $C \in \mathcal{C}^\ell$, an Event-TB search is performed. During the `Enqueue` method, transfers with a rank smaller than ℓ are discarded because they were already found to be irrelevant for traversing the subcells of C. For each reached OBE $T'[k] \in \mathcal{O}$, the found $T[i]$-$T'[k]$-journey J is unpacked and $r(\mathfrak{t}) = \ell + 1$ is set for every transfer $\mathfrak{t}$ in J.

Correctness. The correctness of T-REX relies on the *subjourney closure property* of TB:

▶ **Theorem 2.** *Let J be a journey returned by TB for a query Q and let $J[\varepsilon_\mathrm{s}, \varepsilon_\mathrm{t}]$ be an event subjourney of J. An Event-TB search from ε_s finds the journey $J[\varepsilon_\mathrm{s}, \varepsilon_\mathrm{t}]$.*

We give a detailed proof of this claim in Appendix A. We stress that although it seems intuitively clear that TB has this property, the proof is far from trivial and requires careful analysis of the interaction between different pruning rules in the transfer generation and query phases. In fact, Baum et al. [8] showed that an analogous property does not hold for RAPTOR. Using this property, the correctness of T-REX is fairly straightforward.

▶ **Theorem 3.** *Let J be a journey returned by TB for a query $Q = (p_s, p_t, \tau_\mathrm{dep})$. After the T-REX customization, every transfer $\mathfrak{t}$ in J fulfills Equation (1).*

Proof. Let $\mathfrak{t} = (T_m[j_m], T_n[i_n])$, $p = p(T_m[j_m])$ and $r = \min\{\mathrm{LCL}(p, p_s), \mathrm{LCL}(p, p_t)\}$. We show by induction that for each level ℓ with $0 \le \ell < r$, the customization extracts $\mathfrak{t}$ and sets $r(\mathfrak{t}) \leftarrow \ell + 1$. This ensures that $r(\mathfrak{t}) \ge r$ holds after the customization, i.e., Equation (1) holds. Let $J_\mathcal{E} = J[T_a[i_a], T_b[j_b]]$ be the maximal subjourney of J such that $J_\mathcal{E}$ uses $\mathfrak{t}$ and all stop events of $J_\mathcal{E}$ except (possibly) for $T_a[i_a]$ lie inside the cell $c_\mathrm{ID}(p, \ell)$. Because $\ell < r$, neither p_s nor p_t is located in this cell, so it follows that $T_a[i_a]$ is an IBE of the cell and $T_b[j_b]$ is an OBE. We show that the Event-TB search from $T_a[i_a]$ during the customization of cell $c_\mathrm{ID}(p, l)$ finds $J_\mathcal{E}$. Without the rank-based pruning rule in the modified `Enqueue` method, this follows from Theorem 2. This rule prunes the search at any transfer $\mathfrak{t}^*$ with $r(\mathfrak{t}^*) < \ell$. In the base case $\ell = 0$, this is trivially false for all transfers in $J_\mathcal{E}$, and otherwise it follows from the induction hypothesis. Hence, the search finds $J_\mathcal{E}$ and sets the rank of $\mathfrak{t} \in J_\mathcal{E}$ to $\ell + 1$. ◀

Updates. Real-time updates can affect all three components of the T-REX preprocessing: the partition, the transfer set, and the ranks. Changes to the compact layout graph (e.g., rerouting a line) are fairly rare and can often be incorporated without recomputing the partition. Updating the transfer set is discussed in [62]. The ranks can be updated in a bottom-up fashion by re-running the customization in the affected cells. Moreover, a cell C on level ℓ does not need to be processed if the customization on level $\ell - 1$ did not change the ranks of any transfers. In the full version, we present more sophisticated update algorithms that exploit the fact that updates usually only affect a short time window in practice.

5 Experiments

All algorithms are implemented in C++ and compiled using GCC 14.2.0 with `-march=native -O3`. The code is publicly available [52, 27], except for CSA and ACSA, which were provided to us by the authors [20]. All precomputations besides partitioning were run on a machine with two 64-core AMD Epyc 7742 CPUs clocked at 2.25 GHz, with a boost frequency of 3.4 GHz, 1 024 GB of DDR4-3200 RAM, and 256 MB of L3 cache. Partitioning and queries were performed on a machine with two 8-core Intel Xeon Gold-6144 CPUs clocked at 3.5 GHz, with a boost frequency of 4.2 GHz, 192 GB of DDR4-2666 RAM, and 24.75 MB of L3 cache.

Datasets. Our datasets [53] include a metropolitan region, two country-wide networks and a Europe-wide network; cf. Table 1 (additional datasets are evaluated in the full version). The datasets of Switzerland and Paris were extracted from official General Transit Feed Specification (GTFS) feeds [39]. The Germany dataset was extracted from gtfs.de [38]. We created the Europe dataset by extracting the datasets from all European countries from Transitous [59] and merging with using the tool gtfstidy [28]. The footpaths are taken from the GTFS data except for Europe, where we connected all pairs of stops closer than 100 m. For all networks, we then computed the transitive closure. As a result, the Europe dataset contains many footpaths longer than 100 m. To compute the traversal times, we assumed a walking speed of 1.43 m/s. For each dataset, we selected a service period of two consecutive weekdays. We grouped the trips into lines with the simple greedy algorithm described in [8].

Table 1 An overview of the networks used in our experiments. Also shown are the sizes of the TB transfer set $\mathfrak{T}$ and the compact layout graph G_L.

	Europe	Germany	Switzerland	Paris
Stops	1 346 013	435 550	29 045	41 757
Stop events	107 252 199	30 680 868	5 032 795	4 636 238
Lines	384 075	202 238	15 967	9 558
Trips	4 848 876	1 547 126	319 159	215 526
Footpaths	1 752 418	1 116 976	22 186	445 912
TB $\lvert\mathfrak{T}\rvert$	269 766 664	59 181 359	8 075 627	23 284 123
$\lvert V_L\rvert$	778 419	308 004	24 257	13 765
$\lvert E_L\rvert$	2 298 358	1 038 032	60 154	45 882

Setup. We compare T-REX to TB, RAPTOR, CSA, ACSA and FLASH-TB. In order to use compact data structures, we limit all round-based algorithms to 16 rounds. As previously argued in [60] and [20], optimal journeys with more than 16 trips are rare and very unlikely to be considered desirable by users. We report average query times over 10 000 queries, with the source and target stops chosen uniformly at random, and the departure times chosen uniformly at random within the first day of the service period. The time for journey unpacking, which takes a few microseconds, is excluded. For ACSA, we evaluated the same partitions as for T-REX and report the configuration with the lowest query times. For FLASH-TB, we computed the partitions using KaHIP, as in [37].

Partitioning. To compute the multilevel partition of the compact layout graph, we use a custom version of the open-source partitioner Mt-KaHyPar [34, 33, 36, 30]. Although Mt-KaHyPar was designed primarily for hypergraph partitioning, it also supports ordinary

Table 2 Preprocessing times for T-REX and ACSA by phase, measured in seconds.

Network	Partitioning	T-REX			ACSA	
		Transfers	Customization	Total	Customization	Total
Europe	53.52	24.02	54.28	131.82	94.19	147.71
Germany	22.67	5.40	10.97	39.04	12.89	35.56
Switzerland	1.49	0.42	0.85	2.76	2.72	4.21
Paris	1.04	2.09	4.82	7.95	17.19	18.22

graphs. Normally, Mt-KaHyPar adaptively reduces the allowed imbalance on the higher levels to ensure that the final 2^K-partition still respects the global imbalance bound ε [56]. Our custom version [31] instead uses the given imbalance ε on each level. Allowing extreme imbalances at the lower levels can negatively affect both the customization time, as the larger cells are more expensive to process, and the query time, as more source-target pairs have a low LCL. However, we found that these effects were outweighed by the smaller cut size. We parallelized Mt-KaHyPar with 6 cores, as we observed that adding further cores degraded the partition quality. In all subsequent experiments, we evaluated four imbalance values (25 %, 50 %, 75 % and 100 %) for each network and each choice for the number of levels K. We report the configuration with the lowest query times (cf. Table 4).

Details on the choice of the partitioner are given in the full version. We observed that our version of Mt-KaHyPar produces much better results on our networks than KaHIP [51, 29]. In particular, the cuts on the highest few levels are drastically smaller, with a reduction in the number of IBEs by up to two orders of magnitude. The partitioning time is reduced by a factor of 14.4, the customization time by a factor of 2.0 and the query time by a factor of 2.5.

Preprocessing. On the Europe instance, the customization requires 46 minutes when run sequentially. The efficiency of the parallelization, i.e., the speedup over sequential execution divided by the number of used threads, remains above 0.6 for up to 64 threads. For 128 threads, the efficiency decreases to 0.4, for a total customization time of 54 seconds. The decrease in the efficiency is likely to the limited memory bandwidth of the AMD Eypc machine, which has been observed before [37]. Table 2 reports the times for the different preprocessing phases of T-REX and ACSA. The T-REX customization takes roughly twice as long as the transfer generation. The total T-REX preprocessing time is similar to that of ACSA on the larger networks and faster on the smaller ones, despite having an additional phase. On the smaller networks, the transfer generation and customization are fast enough that they can be re-run every few seconds to incorporate updates. This is not feasible on Europe, where processing the entire two-day timetable takes 78 seconds. However, updates typically only affect a small time window, which can be processed much faster. For example, processing the entire rush-hour time window between 7 and 9 AM on the first day requires 8 seconds for the transfer generation and 5 seconds for the customization.

Memory Consumption. T-REX without overlays has a memory overhead 5–9 % over TB, required for the cell IDs and transfer ranks. The overlay variant requires 2–4 times as much space as TB and 12–18 times as much as the timetable itself. This is mainly due to the $\text{succ}(T[i], \ell)$ data structure and the transfer overlays, which exist per level. For Europe, the total memory consumption is 8.1 GB for TB, 8.5 GB for basic T-REX and 27.8 GB for the overlay variant. Of these, 5.2 GB are required for the timetable and the transfers. We note that both TB and T-REX are tuned to prioritize query speed over a low memory footprint. We discuss further details and ways to reduce the memory consumption in the full version.

Query. Table 4 reports the performance of T-REX on Europe depending on the number of levels. The query time correlates most closely with the number of relaxed transfers, as most time is spent in the `Enqueue` method. Adding more levels consistently improves the performance until a plateau is reached at nine levels. With a mean query time of 8.7 ms, T-REX is fast enough for interactive applications.

Table 3 Performance of algorithms, averaged over 10 000 random queries.

Network	Algorithm	Pareto	Query [μs]	Preprocessing [hh:mm:ss]	k
Europe	CSA	○	392 486	–	–
	ACSA	○	36 110	00:02:28	16 384
	RAPTOR	●	715 512	–	–
	TB	●	196 536	00:00:24	–
	T-REX	●	8 725	00:02:12	16 384
Germany	CSA	○	83 101	–	–
	ACSA	○	19 206	00:00:36	1 024
	RAPTOR	●	212 255	–	–
	TB	●	65 060	00:00:05	–
	T-REX	●	4 442	00:00:39	16 384
	FLASH-TB (moderate)	●	7 871	29:39:30	8
	FLASH-TB (fastest)	●	82	31:08:21	8 192
Switzerland	CSA	○	3 931	–	–
	ACSA	○	1 901	00:00:04	256
	RAPTOR	●	10 719	–	–
	TB	●	4 569	< 00:00:01	–
	T-REX	●	710	00:00:03	512
	FLASH-TB (moderate)	●	454	00:23:53	8
	FLASH-TB (fastest)	●	15	00:16:43	8 192
Paris	CSA	○	3 280	–	–
	ACSA	○	6 312	00:00:18	64
	RAPTOR	●	8 562	–	–
	TB	●	3 472	00:00:02	–
	T-REX	●	1 641	00:00:08	256
	FLASH-TB (moderate)	●	899	00:29:25	8
	FLASH-TB (fastest)	●	22	00:29:46	16 384

Table 3 compares the performance of T-REX to other algorithms. The speedup over TB is 22.2 on Europe, 14.6 on Germany, 6.4 on Switzerland and 2.1 on Paris. Compared to RAPTOR, which is the fastest Pareto-optimal algorithm without significant preprocessing, T-REX achieves a speedup between 5.2 and 82.0. The speedup over TB on each network is strongly correlated with the distribution of the ranks among the transfer set, depicted in Figure 1. On Europe and Germany, more than 50 % of all transfers have a rank of 0, which indicates that they are only relevant for local journeys. On the smaller networks, this share is smaller but still a sizable plurality. The smaller speedup on Paris is explained by the flatter

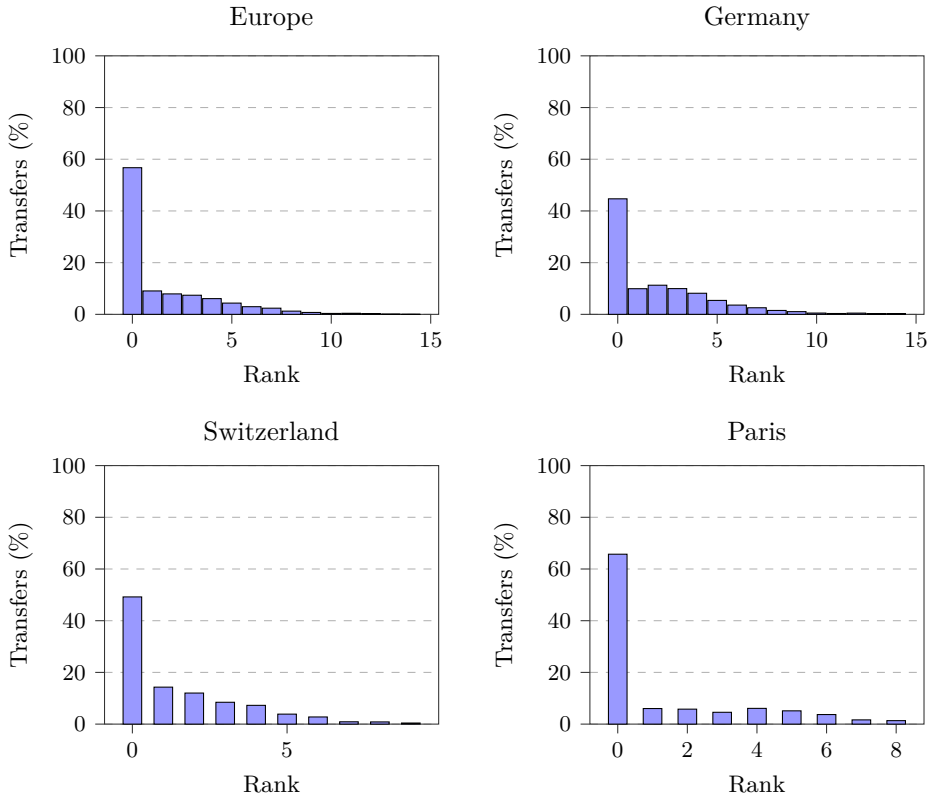

Figure 1 Distribution of the ranks among the transfer set.

rank distribution. This is in line with similar findings for ACSA [20] and FLASH-TB [37], which suggest that metropolitan networks exhibit only a weak hierarchy. However, we stress that RAPTOR is already fast enough for interactive applications on these networks.

FLASH-TB does not scale to Europe due to prohibitive preprocessing times. On the other networks, the fastest configuration is faster than T-REX by up to two orders of magnitude. A moderate configuration with 8 cells (leading to a memory overhead similar to T-REX without overlays) has query times similar to T-REX (more details in the full version). Compared to ACSA, which only minimizes the arrival time, T-REX is faster by a factor of 3–4 and consistently achieves a larger speedup over TB than ACSA over CSA. On Paris, ACSA even causes a slowdown because the overhead for merging the relevant connections dominates the scanning time. By contrast, T-REX achieves a speedup due to a much lower overhead during the query. In the full version, we additionally evaluate how the query time of T-REX scales with the distance between source and target. We also evaluate profile queries, in which the objective is to find Pareto-optimal journeys within a range of possible departure times.

Comparison with Scalable TP. As we do not have access to the code for Scalable TP, we compare to the performance reported in [5] for a Germany instance with a similar number of stop events as ours, but half as many stops. Reported are a sequential precomputation time of 16.5 h and a memory overhead of 1 160 MB. By contrast, T-REX without overlays requires 10 min and a memory overhead of 1 763 MB. The reported query times for Scalable TP are 0.1 ms for local queries in convex clusters, 5 ms in non-convex clusters, and 32 ms for global queries. T-REX without overlays requires 7.8 ms for random queries and much less

Table 4 Performance of T-REX (with overlays) with different numbers of levels on the Europe network, averaged over 10 000 random queries. For 0 levels, we report the metrics for TB.

Levels	Imbalance [%]	Scanned trips	Relaxed transfers	Query [μs]	Customization [mm:ss]
0	–	750 177	14 648 126	196 536	–
1	100	660 736	12 912 414	185 961	00:04
2	25	466 844	9 171 587	125 162	00:20
3	25	275 178	5 262 951	70 455	00:22
4	50	204 557	3 751 287	50 840	00:26
5	25	105 291	1 767 891	25 034	00:18
6	25	77 219	1 151 529	17 650	00:21
7	25	55 840	719 835	12 409	00:18
8	50	52 291	652 466	11 262	00:19
9	25	44 365	445 781	9 224	00:26
10	25	43 574	409 629	9 031	00:36
11	50	44 366	415 962	9 603	00:32
12	25	42 951	371 848	9 596	00:47
13	25	43 530	377 816	9 104	00:58
14	50	41 966	366 894	8 725	00:54
15	50	42 212	358 524	9 006	01:03
16	75	43 870	404 335	9 469	00:53

Table 5 Query performance on Europe (14 levels, 50 % imbalance), depending on which network element is ranked. In the overlay variant, scanned trips refer to the trip segments after splitting.

Ranks for	Overlays	Scanned trips	Relaxed transfers	Query [μs]
None	○	750 177	14 648 126	196 536
Stop events	○	79 774	3 272 304	46 706
Transfers	○	22 696	1 180 213	15 949
Transfers	●	41 966	366 894	8 725

for local queries (cf. full version). Accounting for differences in machines and instances, we observe that T-REX significantly outperforms Scalable TP in terms of preprocessing and query time and has a comparable memory consumption if overlays are not used.

Impact of Ranked Element and Overlays. Table 5 reports the performance of T-REX without overlays if the ranks are assigned to stop events instead of transfers, i.e., the `Enqueue` operation for a transfer $(T_a[j], T_b[i])$ performs the LCL test using $r(T_b[i])$. This variant achieves a speedup of 3.3 over TB. Ranking transfers instead yields a further speedup of 2.4, which supports the findings by Großmann et al. [37] that transfers offer more fine-grained information than stop events. With ranked transfers but without overlays, the number of relaxed transfers is reduced by a factor of 7.6 over TB, yielding a speedup of 7.9. Overlays lead to further improvements despite the overhead of the `Split` method: they reduce the number of relaxed transfers by a factor of 4.8 and provide a further speedup of 2.5. The impact of lazy target pruning is small but positive: on the Europe network, it increases the average number of scanned trip segments by 1.6 % and reduces the query time by 3.7 %.

6 Conclusion

We presented T-REX, a new public transit journey planning algorithm that integrates hierarchical pruning information into Trip-Based Public Transit Routing (TB) [60]. T-REX achieves a speedup of up to 20 over TB, offering query times of less than 10 ms on continent-sized networks with moderate memory overhead and only a few minutes of precomputation time. Together with TB and FLASH-TB, T-REX demonstrates that precomputed transfers are a powerful tool for accelerating public transit journey planning and that they harmonize much better with additional pruning information than other network elements.

An open problem is that the performance of the TB transfer generation step is dependent on the size of the footpath set. Although ULTRA [8] can be used to extend TB to unlimited footpaths, its preprocessing step does not scale well to large networks. For T-REX, we introduced further restrictions by requiring the footpath set to be transitively closed, and by contracting footpaths during partitioning. This was done mainly for ease of exposition, and we believe that these restrictions can be lifted. For example, initial footpaths that leave the source cell can be handled by treating their incident stop events as border events. However, this introduces new scalability issues, and thus extending T-REX to unlimited footpaths will likely require new ideas. Other interesting avenues for future research include combining hierarchical and goal-directed pruning in a TB-based algorithm, and extending T-REX to handle timetables that span longer time periods, which has already been done for TB [62]. Finally, T-REX only supports point-to-point queries in its current form because the hierarchical pruning is based on the source and target cells. For road networks, CRP can be extended to one-to-many search [24]; similar ideas could also be adapted for T-REX.

References

1 Prateek Agarwal and Tarun Rambha. Scalable algorithms for bicriterion trip-based transit routing, 2021. arXiv:2111.06654.

2 Hannah Bast. Car or public transport – two worlds. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science (LNCS)*, pages 355–367. Springer, 2009. doi:10.1007/978-3-642-03456-5_24.

3 Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science (LNCS)*, pages 290–301. Springer, 2010. doi:10.1007/978-3-642-15775-2_25.

4 Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm Engineering: Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science (LNCS)*, pages 19–80. Springer, 2016. doi:10.1007/978-3-319-49487-6_2.

5 Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable transfer patterns. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*, pages 15–29. Society for Industrial and Applied Mathematics, 2016. doi:10.1137/1.9781611974317.2.

6 Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 42–54. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. doi:10.4230/OASIcs.ATMOS.2013.42.

7 Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental study on speed-up techniques for timetable information systems. In *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'07)*, volume 7 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2007. doi:10.4230/OASIcs.ATMOS.2007.1169.

8 Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. ULTRA: Unlimited transfers for efficient multimodal journey planning. *Transportation Science*, 57:1536–1559, 2023. doi:10.1287/trsc.2022.0198.

9 Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating time-dependent multi-criteria timetable information is harder than expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'09)*, volume 12 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009. doi:10.4230/OASIcs.ATMOS.2009.2148.

10 Thomas Bläsius, Valentin Buchhold, Dorothea Wagner, Tim Zeitz, and Michael Zündorf. Customizable contraction hierarchies – A survey, 2025. doi:10.48550/arXiv.2502.10519.

11 Gerth S. Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact timetable queries. In *Proceedings of the 3rd Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'03)*, volume 92, pages 3–15. Elsevier, 2004. doi:10.1016/j.entcs.2003.12.019.

12 Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. Engineering graph-based models for dynamic timetable information systems. *Journal of Discrete Algorithms*, 46–47:40–58, 2017. doi:10.1016/j.jda.2017.09.001.

13 Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003. doi:10.1137/S0097539702403098.

14 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public transit labeling. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science (LNCS)*, pages 273–285. Springer, 2015. doi:10.1007/978-3-319-20086-6_21.

15 Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster transit routing by hyper partitioning. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'17)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/OASIcs.ATMOS.2017.8.

16 Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017. doi:10.1287/TRSC.2014.0579.

17 Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering time-expanded graphs for faster timetable information. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, volume 5868 of *Lecture Notes in Computer Science (LNCS)*, pages 182–206. Springer, 2009. doi:10.1007/978-3-642-05465-5_7.

18 Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2015. doi:10.1287/TRSC.2014.0534.

19 Mattia D'Emidio, Imran Khan, and Daniele Frigioni. Journey planning algorithms for massive delay-prone transit networks. *Algorithms*, 13(1):2, 2020. doi:10.3390/A13010002.

20 Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *Journal of Experimental Algorithmics*, 23, 2018. doi:10.1145/3274661.

21 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *Journal of Experimental Algorithmics*, 21:1.5:1–1.5:49, 2016. doi:10.1145/2886843.

22 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi:10.1007/BF01386390.

23 Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Proceedings of the 7th International Workshop on Experimental and Efficient Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science (LNCS)*, pages 347–361. Springer, 2008. doi:10.1007/978-3-540-68552-4_26.

24 Alexandros Efentakis and Dieter Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*, volume 8737 of *Lecture Notes in Computer Science (LNCS)*, pages 358–370. Springer, 2014. doi:10.1007/978-3-662-44777-2_30.

25 Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012. doi:10.1287/trsc.1110.0401.

26 Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. Multimodal dynamic journey-planning. *Algorithms*, 12(10), 2019. doi:10.3390/a12100213.

27 GitHub repository: TransitRouting/FLASH-TB. https://github.com/TransitRouting/FLASH-TB. [Accessed 04-03-2026].

28 GitHub repository: patrickbr/gtfstidy. https://github.com/patrickbr/gtfstidy. [Accessed 04-03-2026].

29 GitHub repository: KaHIP/KaHIP. https://github.com/KaHIP/KaHIP. [Accessed 04-03-2026].

30 GitHub repository: kahypar/mt-kahypar. https://github.com/kahypar/mt-kahypar. [Accessed 04-03-2026].

31 GitHub repository: PatrickSteil/mt-kahypar. https://github.com/PatrickSteil/mt-kahypar. [Accessed 04-03-2026].

32 Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. Society for Industrial and Applied Mathematics, 2005. doi:10.5555/1070432.1070455.

33 Lars Gottesbüren. *Parallel and Flow-Based High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, 2023. doi:10.5445/IR/1000157894.

34 Lars Gottesbüren and Michael Hamann. Deterministic parallel hypergraph partitioning. In *Proceedings of the 28th International Conference on Parallel and Distributed Computing (Euro-Par'22)*, volume 13440, pages 301–316. Springer, 2022. doi:10.1007/978-3-031-12597-3_19.

35 Lars Gottesbüren, Michael Hamann, Tim Niklas Uhl, and Dorothea Wagner. Faster and better nested dissection orders for customizable contraction hierarchies. *Algorithms*, 12(9):196, 2019. doi:10.3390/A12090196.

36 Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. Scalable high-quality hypergraph partitioning. *ACM Transactions on Algorithms*, 20(1):9:1–9:54, 2024. doi:10.1145/3626527.

37 Ernestine Großmann, Jonas Sauer, Christian Schulz, Patrick Steil, and Sascha Witt. FLASH-TB: Integrating arc-flags and trip-based public transit routing. *Transportation Science*, 60(2):242–263, 2026. doi:10.1287/trsc.2023.0481.

38 GTFS für Deutschland. https://gtfs.de/. [Accessed 04-03-2026].

39 General Transit Feed Specification. https://gtfs.org/. [Accessed 04-03-2026].

40 Michael Hamann and Ben Strasser. Graph bisection with Pareto optimization. *Journal of Experimental Algorithmics*, 23:1.2:1–1.2:34, 2018. doi:10.1145/3173045.

41 Peter E. Hart, Nils J. Nilsson, , and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968. doi:10.1109/TSSC.1968.300136.

42 Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *The Shortest Path Problem: Ninth*

DIMACS Implementation Challenge, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 41–72. American Mathematical Society, 2009. doi:10.1090/dimacs/074/03.

43 Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *Journal of Experimental Algorithmics*, 13(2.5):1–26, 2008. doi:10.5555/2791171.2791186.

44 Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14:1029–1046, 2002. doi:10.1109/TKDE.2002.1033772.

45 Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–40. American Mathematical Society, 2009. doi:10.1090/dimacs/074/02.

46 Vassilissa Lehoux and Christelle Loiodice. Faster preprocessing for the trip-based public transit routing algorithm. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*, volume 85 of *OpenAccess Series in Informatics (OASIcs)*, pages 3:1–3:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/OASIcs.ATMOS.2020.3.

47 Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup Dijkstra's algorithm. *Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006. doi:10.1145/1187436.1216585.

48 Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria Pareto search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science (LNCS)*, pages 246–263. Springer, 2007. doi:10.1007/978-3-540-74247-0_13.

49 Matthias Müller-Hannemann and Mathias Schnee. Efficient timetable information in the presence of delays. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science (LNCS)*, pages 249–272. Springer, 2009. doi:10.1007/978-3-642-05465-5_10.

50 Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics*, 12:2.4:1–2.4:39, 2008. doi:10.1145/1227161.1227166.

51 Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science (LNCS)*, pages 164–175. Springer, 2013. doi:10.1007/978-3-642-38527-8_16.

52 Jonas Sauer, Patrick Steil, and Sascha Witt. PatrickSteil/TREX. Software, swhId: swh:1:dir:098c3810dedbdf093d9ae741f0b3183c5fbd65e8 (visited on 2026-08-11). URL: https://github.com/PatrickSteil/TREX, doi:10.4230/artifacts.27611.

53 Jonas Sauer, Patrick Steil, and Sascha Witt. Datasets for "T-REX: Fast and dynamic journey planning for continental-scale public transit networks", April 2026. doi:10.5281/zenodo.19697732.

54 Jonas Sebastian Sauer. *Closing the Performance Gap Between Multimodal and Public Transit Journey Planning*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2024. doi:10.5445/IR/1000173225.

55 Aaron Schild and Christian Sommer. On balanced separators in road networks. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, volume 9125 of *Lecture Notes in Computer Science (LNCS)*, pages 286–297. Springer, 2015. doi:10.1007/978-3-319-20086-6_22.

56 Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*, pages 53–67. Society for Industrial and Applied Mathematics, 2016. doi:10.1137/1.9781611974317.5.

57 Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science (LNCS)*, pages 43–59. Springer, 2002. doi:10.5555/646680.702320.

58 Patrick Steil. A multilevel design for scalable Pareto-optimal public transit queries. Master's thesis, Karlsruher Institut für Technologie (KIT), 2025. doi:10.5445/IR/1000190843.

59 Transitous. https://transitous.org/. [Accessed 04-03-2026].

60 Sascha Witt. Trip-based public transit routing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science (LNCS)*, pages 1025–1036. Springer, 2015. doi:10.1007/978-3-662-48350-3_85.

61 Sascha Witt. Trip-based public transit routing using condensed search trees. In *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'16)*, volume 54 of *OpenAccess Series in Informatics (OASIcs)*, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/OASIcs.ATMOS.2016.10.

62 Sascha Witt. Extending the time horizon: Efficient public transit routing on arbitrary-length timetables, 2021. arXiv:2109.14143.

A Proof of Subjourney Closure

We give the proof of Theorem 2, which was omitted from Section 4:

▶ **Theorem 2.** *Let J be a journey returned by TB for a query Q and let $J[\varepsilon_s, \varepsilon_t]$ be an event subjourney of J. An Event-TB search from ε_s finds the journey $J[\varepsilon_s, \varepsilon_t]$.*

To prove this property, we formalize the search space of TB and the order in which it is explored. As a byproduct, we give (to our knowledge) the first complete proof of correctness for TB. Previously, Großmann et al. [37] gave a proof for the query phase but only in conjunction with a different transfer generation algorithm.

We say that a stop event $T[\ell]$ is *visited* by TB in round n if there is a trip segment $T[i, j] \in Q_n$ with $i < \ell \leq j$. In that case, journey unpacking by recursively following the parent pointer of $T[i, j]$ yields a p_s-$T[\ell]$-journey. We say that this is the journey *found* by TB for $T[\ell]$ with n trips. The following is immediate from the design of TB:

▶ **Observation 4.** *Let $J = \langle T_1[i_1, j_1], \ldots, T_k[i_k, j_k] \rangle$ be a journey that is found by (Event-)TB. Then for each $1 \leq n \leq k$ and $i_n < \ell \leq k_n$, the stop event $T_n[\ell]$ is visited and the corresponding found journey is $\langle T_1[i_1, j_1], \ldots, T_n[i_n, \ell] \rangle$.*

A.1 A Total Ordering of Feasible Journeys

In a *one-to-all query* $Q = (p_s, \tau_{\text{dep}})$, we specify a source stop p_s and departure time τ_{dep} but no target stop. The objective is to find the Pareto front and a representative set for every possible target stop in $\mathcal{S}$. TB can answer such a query if we disable target pruning and the maintenance of the tentative Pareto front $\mathcal{L}$. Instead, to reconstruct the solution for a target stop p_t, all visited stop events at p_t are examined, the corresponding found journeys are unpacked, and redundant representatives of the same cost vector are discarded.

Given a one-to-all query $Q = (p_s, \tau_{\text{dep}})$, let $\mathcal{J}(Q, \mathfrak{T})$ denote the set of all feasible journeys for Q that only use transfers in $\mathfrak{T}$. For simplicity, we also use the notation $\mathcal{J}(Q, \mathfrak{T})$ if Q is a stop-to-stop query with a specified target stop p_t; however, in this case, $\mathcal{J}(Q, \mathfrak{T})$ still contains journeys that end at all reachable stops, not just p_t. We formalize the order in which TB finds journeys by defining a total ordering $\prec_Q$ of $\mathcal{J}(Q, \mathfrak{T})$.

The TB query algorithm includes three steps that explore parts of the network without specifying any particular order: The outgoing footpaths of the source stop p_s and the outgoing transfers of a scanned stop event are relaxed in an arbitrary order. Furthermore, after relaxing an initial footpath (p_s, p), including the case $p_s = p$, the lines visiting p are examined in an arbitrary order to find and enqueue the earliest reachable trip. To allow for formal reasoning about the algorithm, we fix an order for these steps. In a practical implementation, this is achieved by assigning integer IDs to all lines, footpaths and transfers. The elements are then examined in ascending order of their ID. To reflect this, we define the functions $\text{ID}_{\mathcal{L}} \colon \mathcal{L} \to \{0, \dots, |\mathcal{L}|-1\}$, $\text{ID}_{\mathcal{F}} \colon \mathcal{F} \to \{0, \dots, |\mathcal{F}|-1\}$ and $\text{ID}_{\mathfrak{T}} \colon \mathfrak{T} \to \{0, \dots, |\mathfrak{T}|-1\}$. We require that for every stop p, the outgoing footpaths $(p, q) \in \mathcal{F}$ are assigned consecutive IDs. Similarly, for every stop event $T_a[j]$, the outgoing transfers $(T_a[j], T_b[i]) \in \mathfrak{T}$ are assigned consecutive IDs.

To describe the order in which TB finds journeys, we define a unique *tiebreaking sequence* for each journey in $\mathcal{J}(Q, \mathfrak{T})$. Let $J = \langle T_1[i_1, j_1], \dots, T_k[i_k, j_k] \rangle \in \mathcal{J}(Q, \mathfrak{T})$ be a journey. Then the tiebreaking sequence of J is given by $X(J) = \langle k \rangle \circ \tilde{X}(J)$ with

$$\tilde{X}(J) = \begin{cases} \langle \text{ID}_{\mathcal{F}}((p_s, p(T_1[i_1]))), \text{ID}_{\mathcal{L}}(L(T_1)), i_1, \tau_{\text{dep}}(T_1[1]), j_1 \rangle & \text{if } k = 1, \\ \langle \tilde{X}(J[1, k-1]) \circ \langle \text{ID}_{\mathfrak{T}}((T_{k-1}[j_{k-1}], T_k[i_k])), j_k \rangle & \text{otherwise.} \end{cases}$$

For two journeys $J, J' \in \mathcal{J}(Q, \mathfrak{T})$, we write $J \prec_Q J'$ if $X(J)$ is smaller than $X(J')$ lexicographically. Because the sequences are unique, $\prec_Q$ is a total ordering of $\mathcal{J}(Q, \mathfrak{T})$.

In the following, we also consider event-based queries. Here, instead of a source stop p_s and a departure time τ_{dep}, we are given an initial stop event ε_{s} and journeys are feasible if they start with ε_{s}. The tiebreaking sequence changes accordingly: we have $\tilde{X}(J) = \langle j_1 \rangle$ in the case $k = 1$, but the other components remain the same.

▶ **Lemma 5.** *For a query Q, (Event-)TB finds journeys in increasing order of $\prec_Q$.*

Proof. This follows directly by comparing the definition of the tiebreaking sequence with the algorithm description. A detailed proof is given in the full version. ◀

A.2 Stability of the Total Ordering

To prove subjourney closure, we need to examine how the relative order of two journeys is affected when we add or remove prefixes or suffixes. The following is immediately apparent from the recursive construction of the tiebreaking sequence:

▶ **Observation 6.** *For two p_s-p_t-journeys $J \neq J' \in \mathcal{J}(Q, \mathfrak{T})$ with k trips, let n be an index with $1 \leq n \leq k$ such that $J[1, n] \neq J'[1, n]$. Then $J \prec_Q J'$ iff $J[1, n] \prec_Q J'[1, n]$.*

In particular, this means that the relative order of two journeys does not change when we add the same proper suffix to both. We say that the ordering is *stable* under the addition of proper suffixes. For prefixes, the situation is slightly more complicated. If the added prefix is a proper journey, then the initial footpath becomes an intermediate transfer. Because footpaths and transfers are handled with different tiebreaking rules, it cannot be guaranteed that the relative order stays the same. However, we can show that the ordering is stable under the addition of prefixes that end in the middle of a trip. For this, we introduce notation for the concatenation of two journeys.

▶ **Definition 7.** *Let $J_1 = \langle T_a[i_a, j_a], \ldots, T_b[i_b, x]\rangle$ and $J_2 = \langle T_b[x, j_b], \ldots, T_c[i_c, j_c]\rangle$ be two journeys that coincide in the event $T_b[x]$. We define the concatenated journey $J_1 \circ J_2 = \langle T_a[i_a, j_a], \ldots, T_b[i_b, j_b], \ldots T_c[i_c, j_c]\rangle$. Note that this journey uses $|J_1| + |J_2| - 1$ trips. We observe: If $J_1 \in \mathcal{J}(Q, \mathfrak{T})$ for some query Q and $J_2 \in \mathcal{J}(Q', \mathfrak{T})$ for the event-based query $Q' = (T_b[x])$, then it follows that $J_1 \circ J_2 \in \mathcal{J}(Q, \mathfrak{T})$ because all of its transfers are in J_1 or J_2.*

▶ **Lemma 8.** *Let $J \in \mathcal{J}(Q, \mathfrak{T})$ be a journey for a query Q, let J_s be a suffix of J that starts at some stop event ε, and let J_p be the corresponding prefix such that $J = J_p \circ J_s$. Let $J'_s \in \mathcal{J}(Q', \mathfrak{T})$ for the event-based query $Q' = (\varepsilon)$, and let $J' = J_p \circ J'_s$. Then we have $J \prec_Q J'$ iff $J_s \prec_{Q'} J'_s$.*

Proof. Let $\tilde{X}_p$ denote the sequence that consists of all but the final entry of $\tilde{X}(J_p)$, which is the exit index of the final trip segment. Then we have $X(J) = \langle |J_p| + |J_s| - 1\rangle \circ \tilde{X}_p \circ \tilde{X}(J_s)$ and $X(J') = \langle |J_p| + |J'_s| - 1\rangle \circ \tilde{X}_p \circ \tilde{X}(J'_s)$. Compared to the tiebreaking sequences of J_s and J'_s, the first entries are shifted by the same value $|J_p| - 1$ and the shared infix $\tilde{X}_p$ is inserted afterwards. Neither change has an effect on the relative order of the sequences. ◀

A.3 Correctness of TB

In this section, we precisely characterize the representative that TB finds for each cost vector in the Pareto front: it is the $\prec_Q$-minimal one. We can then use this characterization, together with the stability properties established for $\prec_Q$, to prove the subjourney closure property.

We begin by characterizing the journeys that are discarded by TB due to reached index pruning. Let J be a journey ending with a stop event $T[i]$. We say that a journey J' *event-dominates* J if $J' \prec_Q J$ and J' ends in some $T'[i]$ with $T' \preceq T$. A journey J is *pruned* if a prefix of J is event-dominated by a non-pruned journey. We say that J is *stop-dominated* by J' if they end at the same stop, J' does not arrive later and $J' \prec_Q J$. If J is event-dominated by J', it is also stop-dominated (but not necessarily vice versa).

▶ **Observation 9.** *For a query Q, (Event-)TB without target pruning finds exactly the non-pruned journeys in $\mathcal{J}(Q, \mathfrak{T})$.*

This allows us to show that TB finds $\prec_Q$-minimal journeys among the non-pruned ones.

▶ **Lemma 10.** *For a query Q, the representative returned by (Event-)TB for the cost vector (τ_{arr}, k) is $\prec_Q$-minimal among all non-pruned representatives in $\mathcal{J}(Q, \mathfrak{T})$ that are not stop-dominated by a non-pruned journey.*

Proof. We show the claim for a stop-based query $Q = (p_s, p_t, \tau_{\text{dep}})$; the proof for event-based queries is analogous. The returned journey J is the first found p_s-p_t-journey with arrival time τ_{arr}, and there are no found p_s-p_t-journeys with an earlier arrival time in round k. Assume momentarily that target pruning is disabled. Then it follows from Lemma 5 and Observation 9 that J is $\prec_Q$-minimal among all non-pruned representatives for (τ_{arr}, k) in $\mathcal{J}(Q, \mathfrak{T})$. For every other non-pruned representative J', we either have $\tau_{\text{arr}}(J') > \tau_{\text{arr}}$, or $\tau_{\text{arr}}(J') = \tau_{\text{arr}}$ and $J \prec_Q J'$. Hence, J' is stop-dominated by the non-pruned journey J.

Now consider a representative J' that is not found due to target pruning. Then we have $\tau_{\text{arr}}(J') \geq \tau_{\mathsf{min}}$ at the moment of pruning. If J has not been found yet, then we have $\tau_{\text{arr}}(J') \geq \tau_{\mathsf{min}} > \tau_{\text{arr}}$. Otherwise, we have $\tau_{\text{arr}}(J') \geq \tau_{\text{arr}}$ and $J \prec_Q J'$. In both cases, J' is stop-dominated by J. ◀

To show that the returned representative is in fact $\prec_Q$-minimal among *all* representatives, including the pruned ones, we need to analyze how the pruning rules in the transfer generation step and the query algorithm interact with each other. The proof below is given for the optimized transfer generation algorithm of Lehoux and Loiodice [46], but it also holds for original one by Witt [60], which uses weaker pruning rules in the first step.

▶ **Lemma 11** (Transfer Generation Correctness). *Let $J = \langle T_1[i_1, j_1], \ldots, T_k[i_k, j_k]\rangle$ be a journey such that $\mathfrak{t}_x = (T_x[j_x], T_{x+1}[i_{x+1}])$ is the first transfer in J that is not in $\mathfrak{T}$. Then there is a journey $J' = \langle T_1[i_1, j_1], \ldots, T_x[i_x, j'_x], T'_{x+1}[i'_{x+1}, j'_{x+1}], \ldots, T'_\ell[i'_\ell, j'_\ell]\rangle \in \mathcal{J}(Q, \mathfrak{T})$ with $\ell \leq k$ that arrives at p_t not later than J.*

Proof. We first show that each step of the transfer generation algorithm upholds the claim for $k = 2$ as an invariant. Let $J = \langle T_1[i_1, j_1], T_2[i_2, j_2]\rangle$ be a journey such that $\mathfrak{t} = (T_1[j_1], T_2[j_2]) \notin \mathfrak{T}$: Then there is a journey $J' \in \mathcal{J}(Q, \mathfrak{T})$ with $J' = \langle T_1[i_1, j'_1], T'_2[i'_2, j'_2]\rangle$ or $J' = \langle T_1[i_1, j'_1]\rangle$ that arrives at p_t not later than J.

If $\mathfrak{t}$ is not generated, this is for one of two reasons: (1) Another transfer $(T_1[j'_1], T'_2[i'_2])$ with $j'_1 \geq j_1$, $i'_2 \leq i_2$ and $T_2 \succeq T'_2$ is generated. Then the claim holds with $J' = \langle T_1[i_1, j'_1], T'_2[i'_2, j_2]\rangle$. (2) We have $T_1 \preceq T_2$ and $j_1 \leq i_2$. Then the claim holds with $J' = \langle T_1[i_1, j_2]\rangle$. If $\mathfrak{t}$ is removed due to the U-turn rule, then we have $p(T_1[j_1 - 1]) = p(T_2[i_2 + 1])$ and $(T_1[j_1 - 1], T_2[i_2 + 1]) \in \mathfrak{T}$ and the claim holds with $J' = \langle T_1[i_1, j_1 - 1], T_2[i_2 + 1, j_2]\rangle$. For the latest-exit rule, it is easy to verify that it upholds the invariant by design.

We now prove the claim for general k by induction over x. The first $x - 1$ intermediate transfers in J are in $\mathfrak{T}$. We show that there is a journey J' with $\ell < k$ trips that arrives at p_t not later than J and for which the first x intermediate transfers are in $\mathfrak{T}$. Consider the subjourney $J_s = \langle T_x[i_x, j_x], T_{x+1}[i_{x+1}, j_{x+1}]\rangle$. By the $k = 2$ case, there is a journey $J'_s \in \mathcal{J}(Q, \mathfrak{T})$ with at most two trips that arrives at p_t not later than J_s. We consider the case that J'_s uses exactly two trips, i.e., $J'_s = \langle T_x[i_x, j'_x], T'_{x+1}[i'_{x+1}, j'_{x+1}]\rangle$; the other case is analogous. Consider the journey

$$J' = \langle T_1[i_1, j_1], \ldots, T_x[i_x, j'_x], T'_{x+1}[i'_{x+1}, j'_{x+1}], T_{x+2}[i_{x+2}, j_{x+2}] \ldots, T_\ell[i_\ell, j_\ell]\rangle$$

that replaces J_s with J'_s in J. This journey uses the same transfers as J with two exceptions: the transfer $\mathfrak{t}_x$ is replaced by $\mathfrak{t}'_x = (T_x[j'_x], T'_{x+1}[i'_{x+1}])$ and $\mathfrak{t}_{x+1} = (T_{x+1}[j_{x+1}], T_{x+2}[i_{x+2}])$ is replaced by $\mathfrak{t}'_{x+1} = (T'_{x+1}[j'_{x+1}], T_{x+2}[i_{x+2}])$. We observe that $\mathfrak{t}'_x \in \mathfrak{T}$ and that $\mathfrak{t}'_{x+1}$ is feasible because $\mathfrak{t}_{x+1}$ is feasible and we have $p(T_{x+1}[j_{x+1}]) = p(T'_{x+1}[j'_{x+1}])$ and $\tau_{\text{arr}}(T'_{x+1}[j'_{x+1}]) \leq \tau_{\text{arr}}(T_{x+1}[j_{x+1}])$. Hence, J' is feasible, arrives at p_t not later than J, and the first x intermediate transfers are in $\mathfrak{T}$. ◀

It is not immediately clear that the latest-exit rule is compatible with reached index pruning during the query: The idea behind reached index pruning is that if stop events $T_a[j]$ and $T'_a[j]$ with $T_a \prec T'_a$ are both reachable, then it is always preferable to use $T_a[j]$. However, consider a journey that uses a transfer $(T_a[j], T_b[i])$. The latest-exit rule may discard this transfer if there is a dominating alternative for every possible target stop, but it may retain the transfer $(T'_a[j], T_b[i])$. This is only correct if the dominating alternative is itself not pruned and can therefore be found by TB; see Figure 2. We show that this is the case.

▶ **Lemma 12** (Reached Index Pruning Correctness). *For each pruned optimal journey $J \in \mathcal{J}(Q, \mathfrak{T})$, there exists a non-pruned journey $J' \in \mathcal{J}(Q, \mathfrak{T})$ that stop-dominates J.*

Proof. We show the existence of a journey $J' \in \mathcal{J}(Q, \mathfrak{T})$ that stop-dominates J. If J' is also pruned, then we can apply the same argument iteratively because stop domination is transitive. Because $J' \prec_Q J$ and because the number of journeys in $\mathcal{J}(Q, \mathfrak{T})$ with at most $|J|$ trips is finite, it follows that there is a choice of J' such that J' is non-pruned.

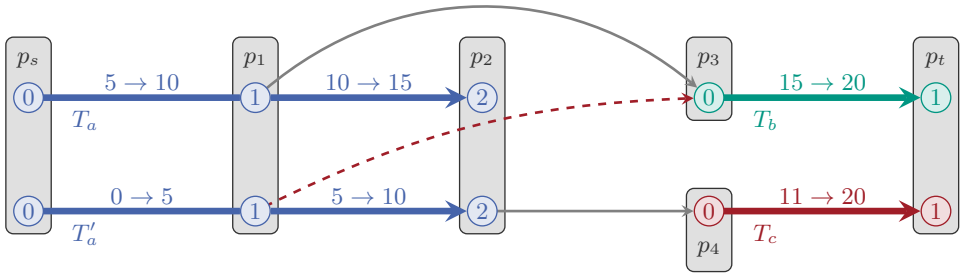

Figure 2 An example showing the potential conflict between the latest-exit rule and reached index pruning. Gray boxes represent stops. Nodes within the boxes represent stop events and are labeled with their indices along the respective trip. Colored edges represent trips, which are labeled with the departure and arrival times of the respective stop events. Gray edges represent transfers. The red dashed edge represents the transfer $(T'_a[1], T_b[0])$, which is discarded due to latest-exit pruning. The reason for this is the transfer $(T'_a[2], T_c[0])$, which allows the stop p_t to be reached with the same arrival time. On the other hand, the transfer $(T_a[2], T_c[0])$ is not feasible, and there is no footpath from p_1 to p_4 or from p_2 to p_3. During a query from p_s to p_t with departure time 0, reached index pruning ensures that T'_a is scanned but T_a is not. Hence, the transfer $(T_a[1], T_b[0])$ is not relaxed. To resolve the conflict, it must be ensured that the query algorithm relaxes the transfer $(T'_a[2], T_c[0])$.

Because J is pruned, there is a prefix J_p of J that is event-dominated by a non-pruned journey $J'_p \in \mathcal{J}(Q, \mathfrak{T})$. Let $T_x[i_x, j_x]$ denote the final trip segment of J_p. Then the final stop event of J'_p is $T'_x[j_x]$ with $T'_x \preceq T_x$. Because the transfer $\mathfrak{t}_x = (T_x[j_x], T_{x+1}[j_{x+1}])$ is feasible, the transfer $\mathfrak{t}'_x = (T'_x[j_x], T_{x+1}[j_{x+1}])$ is feasible as well. Hence, if we replace J_p with J'_p in J, we obtain a feasible journey J_c with $|J|$ trips that arrives at p_t not later than J. We distinguish between two cases:

- If $\mathfrak{t}'_x \in \mathfrak{T}$, then we have $J_c \in \mathcal{J}(Q, \mathfrak{T})$. Furthermore, we have $J_p = J[1, x]$ and $J'_p = J'[1, x]$, so it follows from Observation 6 and $J'_p \prec_Q J_p$ that $J_c \prec_Q J$. Hence, $J' = J_c$ is the desired journey.
- If $\mathfrak{t}'_x \notin \mathfrak{T}$, then we have $\mathfrak{t}'_x \neq \mathfrak{t}_x$ and therefore $T'_x \prec T_x$. By Lemma 11, there is a journey $J_r \in \mathcal{J}(Q, \mathfrak{T})$ with at most $|J|$ trips that is identical to J'_p up to $T'[i_x]$ and arrives at p_t not later than J_c and therefore also not later than J'. We show that $J_r \prec_Q J$ and therefore $J' = J_r$ is the desired journey. Note that $J_r[1, x-1] = J'_p[1, x-1]$ and $J[1, x-1] = J_p[1, x-1]$. Thus, if $J'_p[1, x-1] \neq J_p[1, x-1]$, then it follows from Observation 6 and $J'_p \prec_Q J_p$ that $J_c \prec_Q J$. Otherwise, the relative order of the journeys depends only on trip segment x, which is $T'_x[i_x, j'_x]$ for J_r, $T'_x[i_x, j_x]$ for J'_p, and $T_x[i_x, j_x]$ for J and J_p. It follows from $J'_p \prec_Q J_p$ that trip segments starting with $T'_x[i_x]$ come before those starting with $T_x[i_x]$ in the order, so we have $J_r \prec_Q J$. ◀

▶ **Theorem 13** (TB Correctness). *For a query Q, the representative returned by (Event-)TB for the cost vector (τ_{arr}, k) is the $\prec_Q$-minimal representative.*

Proof. Let J be the returned representative. By Lemma 10, it suffices to show that no pruned journey stop-dominates J. If a pruned journey J' stop-dominates J, then by Lemma 12, there is a non-pruned journey J_u that stop-dominates J'. Because stop domination is transitive, J_u also stop-dominates J, which contradicts Lemma 10. ◀

A.4 Subjourney Closure Property

Finally, we put the pieces together and prove the subjourney closure property.

▶ **Theorem 2.** *Let J be a journey returned by TB for a query Q and let $J[\varepsilon_s, \varepsilon_t]$ be an event subjourney of J. An Event-TB search from ε_s finds the journey $J[\varepsilon_s, \varepsilon_t]$.*

Proof. Let $Q' = (\varepsilon, p_t)$ be the event-to-stop query and let J_s be the ε-p_t-subjourney of J. It follows from the optimality of J that J_s is optimal. We show that J_s is the $\prec_{Q'}$-minimal representative for its cost vector. Then it follows by Theorem 13 that Event-TB returns it and it follows by Observation 4 that the prefix $J[\varepsilon_s, \varepsilon_t]$ is found. Let J'_s be another ε-p_t-journey. Let J_p be the journey such that $J = J_p \circ J_s$ and let $J' = J_p \circ J'_s$. By Theorem 13, we have $J \prec_Q J'$. Then it follows by Lemma 8 that $J_s \prec_{Q'} J'_s$. ◀