Trip-Based Public Transit Routing

Sascha Witt
sascha.witt@kit.edu

Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany

We study the problem of computing all Pareto-optimal journeys in a public transit network regarding the two criteria of arrival time and number of transfers taken. We take a novel approach, focusing on trips and transfers between them, allowing fine-grained modeling. Our experiments on the metropolitan network of London show that the algorithm computes full 24-hour profiles in 70 ms after a preprocessing phase of 30 s, allowing fast queries in dynamic scenarios.

1. Introduction

Recent years have seen great advances in route planning on continent-sized road networks [2]. Unfortunately, adapting these algorithms to public transit networks is harder than expected [4]. On road networks, one is usually interested in the shortest path between two points, according to some criterion. On public transit networks, several variants of point-to-point queries exist. The simplest is the *earliest arrival query*, which takes a departure time as an additional input and returns a journey that arrives as early as possible. A natural extension is the *multi-criteria* problem of minimizing both arrival time and the number of transfers, resulting in a set of journeys. A *profile query* determines all optimal journeys departing during a given period of time.

In the past, these problems have been solved by modeling the timetable information as a graph and running Dijkstra's algorithm or variants thereof on that graph. Traditional graph models include the time-expanded and the time-dependent model [14]. More recently, algorithms such as RAPTOR [10] and Connection Scan [11] have eschewed the use of graphs (and priority queues) in favor of working directly on the timetable.

In this work, we present a new algorithm that uses trips (vehicles) and the transfers between them as its fundamental building blocks. Unlike existing algorithms, it does not assign labels to stops. Instead, trips are labeled with the stops at which they are boarded.

*The final authenticated version is available online at https://doi.org/10.1007/978-3-662-48350-3_85

Then, a precomputed list of transfers to other trips is scanned and newly reached trips are labeled. When a trip reaches the destination, a journey is added to the result set. The algorithm terminates when all optimal journeys have been found.

A motivating observation behind this is the fact that labeling stops with arrival (or departure) times is not sufficient once minimum change times are introduced. Some additional information is required to track which trips can be reached. For example, the realistic time-expanded model of Pyrga et al. [16] introduces additional nodes to deal with minimum change times, while Connection Scan [11] uses additional labels for trips. In contrast, once we know passengers boarded a trip at a certain stop, their further options are fully defined: Either they transfer to another trip using one of the precomputed transfers, or their current trip reaches the destination, in which case we can look up the arrival time in the timetable. In either case, there is no need to explicitly track arrival times at intermediary stops.

The core of the algorithm is similar to a breadth-first search, where levels correspond to the number of transfers taken so far. As a result, it is inherently multi-criterial, similar to RAPTOR [10]. Although a graph-like structure is used, there is no need for a priority queue. A preprocessing step is required to compute transfers, but can be parallelized trivially and only takes minutes, even on large networks (Section 4). By omitting unnecessary transfers, space usage and query times can be improved at the cost of increased preprocessing time.

Section 2 introduces necessary notations and definitions, before Section 3 describes the algorithm and its variants. Section 4 presents the experimental evaluation. Finally, Section 5 concludes the paper.

2. Preliminaries

2.1. Notation

We consider public transit networks defined by an aperiodic *timetable*, consisting of a set of stops, a set of footpaths and a set of trips. A *stop* p represents a physical location where passengers can enter or exit a vehicle, such as a train station or a bus stop. Changing vehicles at a stop p may require a certain amount of time $\Delta\tau_{\text{ch}}(p)$ (for example, in order to change platforms).[1] *Footpaths* allow travelers to walk between two stops. We denote the time required to walk from stop p_1 to p_2 by $\Delta\tau_{\text{fp}}(p_1, p_2)$ and define $\Delta\tau_{\text{fp}}(p, p) = \Delta\tau_{\text{ch}}(p)$ to simplify some algorithms. A *trip* t corresponds to a vehicle traveling along a sequence of stops $\vec{p}(t) = \langle p_t^0, p_t^1, \ldots \rangle$. Note that stops may occur multiple times in a sequence. For each stop p_t^i, the timetable contains the arrival time $\tau_{\text{arr}}(t, i)$ and the departure time $\tau_{\text{dep}}(t, i)$ of the trip at this stop. Additionally, we group trips with identical stop sequences into *lines*[2] such that all trips t and u that share a line can be totally ordered by

$$t \preceq u \iff \forall i \in [0, |\vec{p}(t)|) : \tau_{\text{arr}}(t, i) \leq \tau_{\text{arr}}(u, i) \tag{1}$$

and define

$$t \prec u \iff t \preceq u \wedge \exists i \in [0, |\vec{p}(t)|) : \tau_{\text{arr}}(t, i) < \tau_{\text{arr}}(u, i)\,. \tag{2}$$

[1] More fine-grained models, such as different change times for specific platforms, can be used without affecting query times, since minimum change times are only relevant during preprocessing (Section 3.1).

[2] We chose *line* over *route* to avoid confusion with *routing* and the usage of *route* in the context of road networks.

If two trips have the same stop sequence, but cannot be ordered (because one overtakes the other), we assign them to different lines. We denote the line of a trip t by L_t and define $\vec{p}(L_t) = \vec{p}(t)$. We also define the set of lines at stop p as

$$\boldsymbol{L}(p) = \left\{(L, i) \;\middle|\; p = p_L^i \text{ where } L \text{ is a line and } \vec{p}(L) = \left\langle p_L^0, p_L^1, \ldots \right\rangle\right\}. \tag{3}$$

A *trip segment* $p_t^b \to p_t^e$ represents a trip t traveling from stop p_t^b to stop p_t^e. A *transfer* between trips t and u ($t \neq u$) is denoted by $p_t^e \to p_u^b$, where passengers exit t at the eth stop and board u at the bth. For all transfers,

$$p_t^e \to p_u^b \implies \tau_{\text{arr}}(t, e) + \Delta\tau_{\text{fp}}\left(p_t^e, p_u^b\right) \leq \tau_{\text{dep}}(u, b) \tag{4}$$

must hold. Finally, a *journey* is a sequence of alternating trip segments and transfers, with optional footpaths at the beginning and end. Each leg of a journey must begin at the stop where the previous one ended.

We consider two well-known problems. Since both of them are multi-criteria problems, the results are *Pareto sets* representing non-dominated journeys. A journey dominates another if it is no worse in any criterion; if they are equal in every criterion, we break ties arbitrarily. Although multi-criteria Pareto optimization is NP-hard in general, it is efficiently tractable for natural criteria in public transit networks [15]. In the *earliest arrival problem*, we are given a source stop p_{src}, a target stop p_{tgt}, and a departure time τ. The result is a Pareto set of tuples (τ_{jarr}, n) of arrival time and number of transfers taken during non-dominated journeys from p_{src} to p_{tgt} that leave no earlier than τ. For the *profile problem*, we are given source stop p_{src}, target stop p_{tgt}, an earliest departure time τ_{edt}, and a latest departure time τ_{ldt}. Here, we are asked to compute a Pareto set of tuples $(\tau_{\text{jdep}}, \tau_{\text{jarr}}, n)$ representing non-dominated journeys between p_{src} and p_{tgt} with $\tau_{\text{edt}} \leq \tau_{\text{jdep}} \leq \tau_{\text{ldt}}$. Note that for Pareto-optimality, later departure times are considered to be better than earlier ones.

2.2. Related work

Some existing approaches solve these problems by modeling timetable information as a graph, using either the *time-expanded* or the *time-dependent* model. In the (simple) time-expanded model, a node is introduced for each event, such as a train departing or arriving at a station. Edges are then added to connect nodes on the same trip, as well as between nodes belonging to the same stop (corresponding to a passenger waiting for the next train). To model minimum change times, additional nodes and edges are required [16]. One advantage of this model is that all edge weights are constant, which allows the use of speedup techniques developed for road networks, such as contraction. Unfortunately, it turns out that due to different network structures, these techniques do not perform as well for public transit networks [4]. Also, time-expanded graphs are rather large.

The time-dependent approach produces much smaller graphs in comparison. In the simple model, nodes correspond to stops. Edges no longer have constant weight, but are instead associated with (piecewise linear) travel time functions, which map departure times to travel times (or, equivalently, arrival times). The weight then depends on the time at which this function is evaluated. This model can be extended to allow for minimum change times by

adding a node for each line at each stop [16]. Some speedup techniques have been applied successfully to time-dependent graphs, such as ALT [6] and Contraction [12], although not for multi-criteria problems. For these, several extensions to Dijkstra's algorithm exist, among them the *Multicriteria Label-Setting* [13], the *Multi-Label Correcting* [7], the *Layered Dijkstra* [5], and the *Self-Pruning Connection Setting* [9] algorithms. However, as Dijkstra-variants, each of them has to perform rather costly priority queue operations.

Other approaches do not use graphs at all. *RAPTOR* (Round-bAsed Public Transit Optimized Router) [10] is a dynamic program. In each round, it computes earliest arrival times for journeys with n transfers, where n is the current round number. It does this by scanning along lines and, at each stop, checking for the earliest trip of that line that can be reached. It outperforms Dijkstra-based approaches in practice. The *Connection Scan Algorithm* [11] operates on *elementary connections* (trip segments of length 1). It orders them by departure time into a single array. During queries, this array is then scanned once, which is very fast in practice due to the linear memory access pattern.

A number of speedup techniques have been developed for public transit routing. *Transfer Patterns* [1,3] is based on the observation that for many optimal journeys, the sequence of stops where transfers occur is the same. By precomputing these transfer patterns, journeys can be computed very quickly at query time. *Public Transit Labeling* [8] applies recent advances in hub labeling to public transit networks, resulting in very fast query times. Another example is the *Accelerated Connection Scan Algorithm* [17], which combines CSA with multilevel overlay graphs to speed up queries on large networks. The algorithm presented in this work, however, is a new base algorithm; development of further speedup techniques is a subject for future research.

3. Algorithm

3.1. Preprocessing

We precompute transfers so they can be looked up quickly during queries. A key observation is that the majority of possible transfers is not needed in order to find Pareto-optimal journeys, and can be safely discarded. Preprocessing is divided into several steps: Initial computation and reduction. Initial computation of transfers is relatively straightforward. For each trip t and each stop p_t^i of that trip, we examine p_t^i and all stops reachable via (direct) footpaths from p_t^i. For each of these stops q, we iterate over $(L, j) \in \boldsymbol{L}(q)$ and find the first trip u of line L such that a valid transfer $p_t^i \to p_u^j$ satisfying (4) exists. Since, by definition, trips do not overtake other trips of the same line, we can discard any transfers to later trips of line L. Additionally, we do not add any transfers from the first stop ($i = 0$) or to the last stop ($j = |\vec{p}(L)|-1$) of a trip. Furthermore, transfers to trips of the same line are only kept if either $u \prec t$ or $j < i$; otherwise, it is better to simply remain in the current trip. See Algorithm 1 for a pseudocode description of this.

After initial computation is complete, we perform a number of reduction steps, where we discard transfers that are not necessary to find Pareto-optimal journeys. First, we discard any

Algorithm 1 Initial transfer computation

Input: Timetable data
Output: Transfer set T

$T \leftarrow \varnothing$
for each trip t **do**
  for each stop p_t^i on trip t with $i > 0$ **do**
    for each stop q such that $\Delta\tau_{\text{fp}}\left(p_t^i, q\right)$ is defined **do**
      for each line $(L, j) \in \boldsymbol{L}(q)$ with $j < |\vec{p}(L)| - 1$ **do**
        $u \leftarrow$ earliest trip of line L such that $\tau_{\text{arr}}(t,i) + \Delta\tau_{\text{fp}}\left(p_t^i, q\right) \leq \tau_{\text{dep}}(u,j)$
        if $L \neq L_t \vee u \prec t \vee j < i$ **then**
          $T \leftarrow T \cup \left(p_t^i \rightarrow p_u^j\right)$ ▷ Add transfer

Algorithm 2 Remove U-turn transfers

Input: Timetable data, transfer set T
Output: Reduced transfer set T

for each transfer $p_t^i \rightarrow p_u^j \in T$ **do**
  if $p_t^{i-1} = p_u^{j+1} \wedge \tau_{\text{arr}}(t, i-1) + \Delta\tau_{\text{ch}}\left(p_t^{i-1}\right) \leq \tau_{\text{dep}}(u, j+1)$ **then**
    $T \leftarrow T \setminus p_t^i \rightarrow p_u^j$ ▷ Remove U-turn transfer

transfers $p_t^i \rightarrow p_u^j$ where $p_u^{j+1} = p_t^{i-1}$ (we call these *U-turn transfers*) as long as

$$\tau_{\text{arr}}(t, i-1) + \Delta\tau_{\text{ch}}\left(p_t^{i-1}\right) \leq \tau_{\text{dep}}(u, j+1) \tag{5}$$

holds (Algorithm 2). In this case, we can already reach u from t at the previous stop, and because

$$\begin{aligned} \tau_{\text{arr}}(t,i-1) &\leq \tau_{\text{dep}}(t,i-1) \leq \tau_{\text{arr}}(t,i) \\ \leq \tau_{\text{dep}}(u,j) &\leq \tau_{\text{arr}}(u,j+1) \leq \tau_{\text{dep}}(u,j+1), \end{aligned} \tag{6}$$

all trips that can reach t at the previous stop can also reach u, and all trips reachable from u are also reachable from t. Equation (5) may not hold if the stops in question have different minimum change times.

Next, we further reduce the number of transfers by analyzing which transfers lead to improved arrival times. We do this by moving backwards along a trip, keeping track of where and when passengers in that trip can arrive, either by simply exiting the trip or by transferring to another trip reachable from their current position. Again, we iterate over all trips t. For each trip, we maintain two mappings τ_{A} and τ_{C} from stops to arrival time and earliest change time, respectively. Initially, they are set to ∞ for all stops. During execution of the algorithm, they are updated to reflect when passengers arrive (τ_{A}) or can board the next trip (τ_{C}) at each stop.[3] We then iterate over stops p_t^i of trip t in decreasing index order, meaning we examine later stops first. At each stop, we update the arrival time and change time for that

[3] If there are no minimum change times, then $\tau_{\text{A}} = \tau_{\text{C}}$ and we only maintain τ_{A}.

Algorithm 3 Transfer reduction

Input: Timetable data, transfer set T
Output: Reduced transfer set T

for each trip t **do**
  $\tau_A(\cdot) \leftarrow \infty$ ▷ Arrival time at stops
  $\tau_C(\cdot) \leftarrow \infty$ ▷ Earliest change time at stops
  for $i \leftarrow |\vec{p}(t)| - 1, \dots, 1$ **do**
    $\tau_A(p_t^i) \leftarrow \min(\tau_A(p_t^i), \tau_{arr}(t,i))$
    for each stop q such that $\Delta\tau_{fp}(p_t^i, q)$ is defined **do**
      $\tau_A(q) \leftarrow \min(\tau_A(q), \tau_{arr}(t,i) + \Delta\tau_{fp}(p_t^i, q))$
      $\tau_C(q) \leftarrow \min(\tau_C(q), \tau_{arr}(t,i) + \Delta\tau_{fp}(p_t^i, q))$
    for each transfer $p_t^i \to p_u^j \in T$ **do**
      keep $\leftarrow$ **false**
      for each stop p_u^k on trip u with $k > j$ **do**
        keep $\leftarrow$ *keep* $\vee\ \tau_{arr}(u,k) < \tau_A(p_u^k)$
        $\tau_A(p_u^k) \leftarrow \min(\tau_A(p_u^k), \tau_{arr}(u,k))$
        for each stop q such that $\Delta\tau_{fp}(p_u^k, q)$ is defined **do**
          $\eta \leftarrow \tau_{arr}(u,k) + \Delta\tau_{fp}(p_u^k, q)$
          keep $\leftarrow$ *keep* $\vee\ \eta < \tau_A(q) \vee \eta < \tau_C(q)$
          $\tau_A(q) \leftarrow \min(\tau_A(q), \eta)$
          $\tau_C(q) \leftarrow \min(\tau_C(q), \eta)$
      if $\neg$*keep* **then**
        $T \leftarrow T \setminus p_t^i \to p_u^j$ ▷ No improvement, remove transfer

stop if they are improved:

$$\tau_A(p_t^i) \leftarrow \min(\tau_A(p_t^i), \tau_{arr}(t,i)) \quad \text{and}$$
$$\tau_C(p_t^i) \leftarrow \min(\tau_C(p_t^i), \tau_{arr}(t,i) + \Delta\tau_{ch}(p_t^i)).$$

Similarly, we update τ_A and τ_C for all stops q reachable via footpaths from p_t^i:

$$\tau_A(q) \leftarrow \min(\tau_A(q), \tau_{arr}(t,i) + \Delta\tau_{fp}(p_t^i, q)) \quad \text{and}$$
$$\tau_C(q) \leftarrow \min(\tau_C(q), \tau_{arr}(t,i) + \Delta\tau_{fp}(p_t^i, q)).$$

We then determine, for each transfer $p_t^i \to p_u^j$ from t at that stop, if u improves arrival and/or change times for any stop. To do this, we iterate over all stops p_u^k of u with $k > j$ and perform the same updates to τ_A and τ_C as we did above, this time for p_u^k and all stops reachable via footpaths from p_u^k. If this results in any improvements to either τ_A or τ_C, we keep the transfer, otherwise we discard it. Discarded transfers are not required for Pareto-optimal journeys, since we have shown that (a) taking later transfers (or simply remaining in the current trip) leads to equal or better arrival times (τ_A), and (b) all trips reachable via that transfer can also be reached via those later transfers (τ_C). Refer to Algorithm 3 for a pseudocode description.

All these algorithms are trivially parallelized, since each trip is processed independently. Also, there is no need to perform them as separate steps; they can easily be merged into one. We decided to keep them distinct to showcase the separation of concerns. Furthermore, more complex reduction steps are possible, where there are dependencies between trips. For example, to minimize the size of the transfer set, one could compute full profiles between all stops (all-to-all), then keep only those transfers required for optimal journeys. However, that would be computationally expensive. In contrast, the comparatively simple computations presented here can be performed within minutes, even for large networks, while still resulting in a greatly reduced transfer set (see Section 4 for details).

Note that this explicit representation of transfers allows fine-grained control over them. For instance, one can easily introduce transfers between specific trips that would otherwise violate the minimum change time or footpath restrictions, or remove transfers from certain trips. Transfer preferences are another example. If two trips travel in parallel (for part of their stop sequence), there may be multiple possible transfers between them. The algorithm described above discards all but the last of them; by modifying it, preference could be given to transfers that are more accessible, for instance. Since this only has to be considered during preprocessing, query times are unaffected.

3.2. Earliest Arrival Query

As a reminder, the input to an earliest arrival query consists of the source stop p_{src}, the target stop p_{tgt}, and the (earliest) departure time τ, and the objective is to calculate a Pareto set of $\big(\tau_{\text{jarr}}, n\big)$ tuples representing Pareto-optimal journeys arriving at time τ_{jarr} after n transfers. During the algorithm, we remember which parts of each trip t have already been processed by maintaining the index $R(t)$ of the first reached stop, initialized to $R(t) \leftarrow \infty$ for all trips. We also use a number of queues Q_n of trip segments reached after n transfers and a set $\mathscr{L}$ of tuples $(L, i, \Delta\tau)$. The latter indicates lines reaching the target stop p_{tgt}, and is computed by

$$\mathscr{L} = \big\{(L, i, 0) \,\big|\, (L, i) \in \boldsymbol{L}\big(p_{\text{tgt}}\big)\big\} \\ \cup \big\{\big(L, i, \Delta\tau_{\text{fp}}\big(q, p_{\text{tgt}}\big)\big) \,\big|\, (L, i) \in \boldsymbol{L}(q) \wedge \exists \text{ a footpath from } q \text{ to } p_{\text{tgt}}\big\}$$

We start by identifying the trips travelers can reach from p_{src} at time τ. For this, we examine p_{src} and all stops reachable via footpaths from p_{src}. For each of these stops q, we iterate over $(L, i) \in \boldsymbol{L}(q)$ and find the first trip t of line L such that

$$\tau_{\text{dep}}(t, i) \geq \begin{cases} \tau & \text{if } q = p_{\text{src}}, \\ \tau + \Delta\tau_{\text{fp}}(p_{\text{src}}, q) & \text{otherwise.} \end{cases}$$

For each of those trips, if $i < R(t)$, we add the trip segment $p_t^i \to p_t^{R(t)}$ to queue Q_0 and then update $R(u) \leftarrow \min(R(u), i)$ where $t \preceq u \wedge L_t = L_u$, meaning we update the first reached stop for t and all later trips of the same line. Due to the way $\preceq$ is defined in (1), none of these later trips u can improve upon t. By marking them as reached, we eliminate them from the search and avoid redundant work.

Algorithm 4 Earliest arrival query

Input: Timetable, transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ
Output: Result set J

$J \leftarrow \emptyset$
$\mathscr{L} \leftarrow \emptyset$
$Q_n \leftarrow \emptyset$ for $n = 0, 1, \ldots$
$R(t) \leftarrow \infty$ for all trips t
for each stop q such that $\Delta\tau_{\text{fp}}(q, p_{\text{tgt}})$ is defined **do**
    $\Delta\tau \leftarrow 0$ if $p_{\text{tgt}} = q$, else $\Delta\tau_{\text{fp}}(q, p_{\text{tgt}})$
    for each $(L, i) \in \boldsymbol{L}(q)$ **do**
        $\mathscr{L} \leftarrow \mathscr{L} \cup \{(L, i, \Delta\tau)\}$
for each stop q such that $\Delta\tau_{\text{fp}}(p_{\text{src}}, q)$ is defined **do**
    $\Delta\tau \leftarrow 0$ if $p_{\text{src}} = q$, else $\Delta\tau_{\text{fp}}(p_{\text{src}}, q)$
    for each $(L, i) \in \boldsymbol{L}(q)$ **do**
        $t \leftarrow$ earliest trip of L such that $\tau + \Delta\tau \leq \tau_{\text{dep}}(t, i)$
        ENQUEUE$(t, i, 0)$
$\tau_{\text{min}} \leftarrow \infty$
$n \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
    for each $p_t^b \to p_t^e \in Q_n$ **do**
        for each $(L_t, i, \Delta\tau) \in \mathscr{L}$ with $b < i$ and $\tau_{\text{arr}}(t, i) + \Delta\tau < \tau_{\text{min}}$ **do**
            $\tau_{\text{min}} \leftarrow \tau_{\text{arr}}(t, i) + \Delta\tau$
            $J \leftarrow J \cup \{(\tau_{\text{min}}, n)\}$, removing dominated entries
        if $\tau_{\text{arr}}(t, b+1) < \tau_{\text{min}}$ **then**
            for each transfer $p_t^i \to p_u^j \in T$ with $b < i \leq e$ **do**
                ENQUEUE$(u, j, n+1)$
    $n \leftarrow n + 1$

procedure ENQUEUE(trip t, index i, number of transfers n)
    if $i < R(t)$ **then**
        $Q_n \leftarrow Q_n \cup \left\{p_t^i \to p_t^{R(t)}\right\}$
        for each trip u with $t \preceq u \wedge L_t = L_u$ **do**
            $R(u) \leftarrow \min(R(u), i)$

After the initial trips have been found, we operate on the trip segments in $Q_0, Q_1, \ldots$ until there are no more unprocessed elements. For each trip segment $p_t^b \to p_t^e \in Q_n$, we perform the following three steps. First, we check if this trip reaches the target stop. For each $(L_t, i, \Delta\tau) \in \mathscr{L}$ with $i > b$, we generate a tuple $(\tau_{\text{arr}}(t, i) + \Delta\tau, n)$ and add it to the result set, maintaining the Pareto property. Second, we check if this trip should be pruned because it cannot lead to a non-dominated journey. This is the case if we already found a journey with $\tau_{\text{jarr}} < \tau_{\text{arr}}(t, b+1)$. Third, if the trip is not pruned, we examine its transfers. For each transfer $p_t^i \to p_u^j$ with $b < i \leq e$, we check if $j < R(u)$. If so, we add $p_u^j \to p_u^{R(u)}$ to

Q_{n+1} and update $R(v) \leftarrow \min(R(v), j)$ for all v with $u \preceq v \wedge L_u = L_v$. Otherwise, we already reached u or an earlier trip of the same line at j or an earlier stop, and we skip the transfer. A pseudocode description can be found in Algorithm 4.

The main loop is similar to a breadth-first search: First, all trips reachable directly from the source stop are examined, then all trips reached after a transfer from those, etc. Therefore, we find journeys with the least number of transfers first. Any non-dominated journey discovered later cannot have a lower number of transfers and must therefore arrive earlier. This property enables the pruning in step two, which prevents us from having to examine all reachable trips regardless of the target. However, it also means that the journey with the earliest arrival time is the last one discovered, and all journeys with less transfers are found beforehand. This is why we only consider the multi-criteria problem variants.

3.3. Profile Query

We perform profile queries by running the main loop of an earliest arrival query for each distinct departure time in the given interval, preserving labels between runs to avoid redundant work. Later journeys dominate earlier journeys, provided arrival time and number of transfers are equal or better, while earlier journeys never dominate later ones. Therefore, we process later departures first. However, in order to reuse labels across multiple runs, we need to keep multiple labels for each trip, consisting of the index of the first reached stop and the number of transfers required to reach it. Since the number of transfers is limited in practice, we use $R_n(t)$ to denote the first stop reached on trip t after at most n transfers and update $R_{n+1}(t)$ (and following) whenever we update $R_n(t)$. To decide if a trip segment should be queued while processing Q_n, we compare against and update $R_{n+1}(t)$. We also change the pruning step so we compare against the minimum arrival time of journeys with no more than $n+1$ transfers.

To see why labels can be reused, consider two runs with departure times τ_1 and τ_2, where $\tau_1 < \tau_2$, which both reach trip t at stop i after n transfers. Continuing from this point, both will reach the destination at the same time and after the same number of transfers. However, since $\tau_1 < \tau_2$, the journeys departing at τ_2 dominate the journeys departing at τ_1. Knowing this, we can avoid computing them in the first place by computing τ_2 first and keeping the labels.

3.4. Implementation

We improve the performance of the algorithm by taking advantage of SIMD (single instruction, multiple data) instructions, avoiding dynamic memory allocations and increasing locality of reference (reducing cache misses). In our data instances, all lines have less than 200 stops. Also, none of our tests found Pareto-optimal journeys with 16 or more transfers. Thus, we set the maximum number of transfers to 15. During profile queries, we can then update $R_0(t)$ to $R_{15}(t)$ using a single 128-bit vector minimum operation.

To avoid memory allocations during query execution, we replace the n queues with a single, preallocated array. To see why this is possible, note that the maximum number of trip segments queued is bounded by the number of elementary connections. We use pointers to

keep track of the current element, the end of the queue, and the level boundaries (where the number of transfers n is increased).

We improve locality of reference by splitting the steps of the inner loop into three separate loops. Thus, we iterate three times over each level, each time updating the elements in the "queue", before increasing n and moving on to the next level. In the first iteration, we look up $\tau_{\text{arr}}(t, b+1)$ and store it next to the trip segment into the queue. Additionally, we check $\mathscr{L}$ to see if the trip reaches the destination, and update arrival times as necessary. In the second iteration, we perform the pruning step by comparing the time stored in the queue with the arrival time at the destination. If the element is not pruned, we replace it with two indices into the array of transfers, indicating the transfers corresponding to the trip segment. If the element is pruned, we set both indices to 0, resulting in an empty interval. Finally, in the third iteration, we examine this list of transfers and add new trip segments to the queue as necessary. Thus, arrival times $\tau_{\text{arr}}(\cdot, \cdot)$ are required only in the first loop, transfer indices only in the second loop, and transfers and reached stops $R_n(\cdot)$ only in the final loop. This leads to reduced cache pressure and therefore to less cache misses, which in turn results in improved performance (see Section 4). For more details on the data structures used, please refer to Appendix A.

3.5. Journey Descriptions

So far, we only described how to compute arrival time and number of transfers of journeys, which is enough for many applications. However, we can retrieve the full sequence of trip segments as follows. Whenever a trip segment is queued, we store with it a pointer to the currently processed trip segment. Since we replaced the queue with a preallocated array, all entries are preserved until the end of the query. Therefore, when we find a journey reaching the destination, we simply follow this chain of pointers to reconstruct the sequence of trip segments. If required, the appropriate transfers between the trips can be found by rescanning the list of transfers.

4. Experiments

We ran experiments on a dual 8-core Intel Xeon E5-2650 v2 processor clocked at 2.6 GHz, with 128 GB of DDR3-1600 RAM and 20 MB of L3 cache. Our code was compiled using g++ 4.9.2 with optimizations enabled. We used two test instances, summarized in Table 1. The first, available at `data.london.gov.uk`, covers Greater London and includes data for underground, bus, and Docklands Light Railway services for one day. The second consists of data used by `bahn.de` during winter 2011/2012, containing European long distance trains, German local trains, and many buses over two days.

Table 1 also reports the number of transfers before and after reduction, as well as the total space consumption (for the reduced transfers and all timetable data). Reduction eliminates about 84% of transfers for London, and almost 90% for Germany. The times required for preprocessing can be found in Table 2.

Running times reported for queries are averages over 10 000 queries with source and target

Table 1: Instances used for experiments

	London	Germany
Stops	20 764	249 724
Trips	129 263	2 389 253
Connections	4 991 130	46 116 453
Footpaths	45 624	100 470
Lines (Routes)	2 161	232 644
Transfers (full)	121 339 213	1 826 424 894
Transfers (reduced)	19 502 791	186 296 771
Space consumption	115.5 MiB	1 140.9 MiB

Table 2: Preprocessing times for transfer computation and reduction

	London 1 thread	London 16 threads	Germany 1 thread	Germany 16 threads
Computation	18 s	3 s	177 s	37 s
Reduction	357 s	27 s	2 174 s	183 s
Total	375 s	30 s	2 351 s	220 s

stops selected uniformly at random. For profile queries, the departure time range is the first day covered by the timetable; for earliest arrival queries, the departure time is selected uniformly at random from that range. We do not compute full journey descriptions.

We evaluated the optimizations described in Section 3.4, as well as the effect of transfer reduction, on the London instance (Table 3). SIMD instructions are only used in profile queries and enabling them has no effect on earliest arrival queries. With all optimizations, running time for profile queries is improved by a factor of 2. Transfer reduction improves running times by a factor of 3.

We compare our new algorithm to the state of the art in Table 4. We distinguish between algorithms which optimize arrival time only (∘) and those that compute Pareto sets optimizing arrival time and number of transfers (•), and between earliest arrival (∘) and profile (•) queries. We report the average number of label comparisons per stop[4], where available, and the average running time. Direct comparison with the Accelerated Connection Scan Algorithm (ACSA) [17] and Contraction Hierarchies (CH) [12] is difficult, since they do not support bicriteria queries.[5] We have faster query times than CSA [11] and RAPTOR [10], at the cost of a few minutes of preprocessing time. Transfer Patterns (TP) [1, 3] and Public Transit Labeling (PTL) [8] have faster query times (especially on larger instances), however, their preprocessing times are several orders of magnitude above ours.

[4]Note that in our algorithm, labels are not associated with stops, but with trips instead. For better comparison with previously published work, we divided the total number of label comparisons by the number of stops.

[5]ACSA uses transfers to break ties between journeys with equal arrival times.

Table 3: Evaluation of optimizations in Section 3.4, using the London instance

variant	earliest arr. (ms)	profile (ms)
Basic, without SIMD	1.7	145.9
Basic, with SIMD	1.7	113.2
Optimized	1.2	70.0
Optimized, all transfers	3.5	226.0

Table 4: Comparison with the state of the art. Results taken from [2, 3, 8, 17]. Bicriteria algorithms computing a set of Pareto-optimal journeys regarding arrival time and number of transfers are marked in column "tr." (others only optimize arrival time). Profile queries are marked in column "pr.".

algorithm	instance	stops ($\cdot 10^3$)	conn. ($\cdot 10^6$)	tr.	pr.	prep. (h)	comp. /stop	query (ms)
TripBased	London	20.8	5.0	•	∘	< 0.1	23.3	1.2
TP [2]	Madrid	4.6	4.8	•	∘	185.0	n/a	3.1
PTL [8]	London	20.8	5.1	•	∘	49.3	n/a	0.03
RAPTOR [10]	London	20.8	5.1	•	∘	—	10.9	5.4
CSA [11]	London	20.8	4.9	∘	∘	—	26.6	1.8
CH [12]	Europe (LD)	30.5	1.7	∘	∘	< 0.1	n/a	0.3
TripBased	Germany	249.7	46.1	•	∘	< 0.1	41.4	40.8
TP [3]	Germany	248.4	13.9	•	∘	372.0	n/a	0.3
CSA [17]	Germany	252.4	46.2	∘	∘	—	n/a	298.6
ACSA [17]	Germany	252.4	46.2	∘	∘	0.2	n/a	8.7
TripBased	London	20.8	5.0	•	•	< 0.1	1 061.7	70.0
TP [2]	Madrid	4.6	4.8	•	•	185.0	n/a	3.1
rRAPTOR [10]	London	20.8	5.1	•	•	—	1 634.0	922.0
CSA [11]	London	20.8	4.9	•	•	—	3 824.9	466.0
TripBased	Germany	249.7	46.1	•	•	< 0.1	228.0	301.7
TP [3]	Germany	248.4	13.9	•	•	372.0	n/a	5.0
ACSA [17]	Germany	252.4	46.2	∘	•	0.2	n/a	171.0

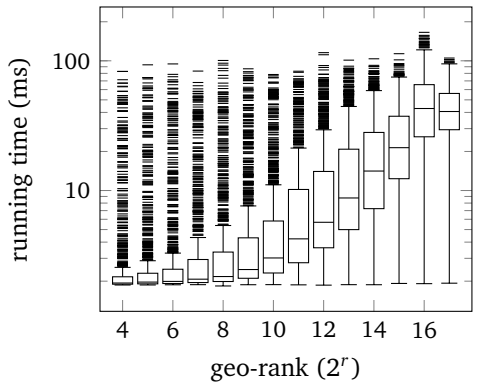

Figure 1: Earliest arrival query times by geo-rank on Germany

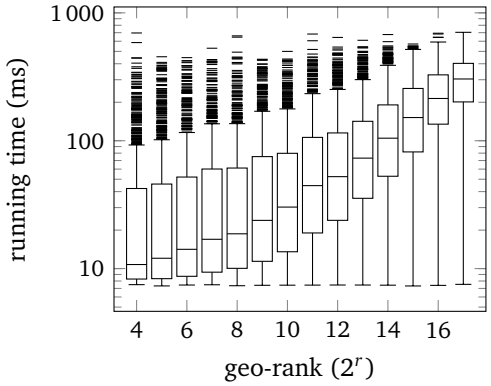

Figure 2: Profile query times by geo-rank on Germany

To examine query times further, we ran 1 000 geo-rank queries [17]. A geo-rank query picks a stop uniformly at random and orders all other stops by geographical distance. Queries are run from the source stop to the 2^r-th stop, where r is the geo-rank. Results for the Germany instance are reported in Figure 1 (earliest arrival queries) and Figure 2 (profile queries). Note the logarithmic scale on both axes. Query times for the maximum geo-rank are about the same as the average query time when selecting source and target uniformly at random, since randomly selected stops are unlikely to be near each other. Local queries, which are often more relevant in practice, are generally much faster (by an order of magnitude), although there is a significant number of outliers, since physically close locations do not necessarily have direct or fast connections.

5. Conclusion

We presented a novel algorithm for route planning in public transit networks. By focusing on trips and transfers between them, we computed multi-criteria profiles optimizing arrival time and number of transfers on a metropolitan network in 70 ms with a preprocessing time of just 30 s, occupying a Pareto-optimal spot among current state of the art algorithms. The explicit representation of transfers allows fine-grained modeling, while the short preprocessing time allows the use in dynamic scenarios. In addition, localized changes (such as trip delays or cancellations) do not necessitate a full rerun of the preprocessing phase. Instead, only a subset of the data needs to be updated. Development of suitable algorithms is a subject of future studies. Future work also includes efficiently extending the covered period of time by exploiting periodicity in timetables, making the algorithm more scalable by using network decomposition, and extending it to support more generic criteria such as fare zones or walking distance.

References

[1] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns. In *European Symposium on Algorithms (ESA)*, volume 6346, pages 290–301, 2010.

[2] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. *ArXiv e-prints*, April 2015, 1504.05140.

[3] Hannah Bast and Sabine Storandt. Frequency-based search for public transit. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22. ACM Press, November 2014.

[4] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, 2009.

[5] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. In *Electronic Notes in Theoretical Computer Science*, volume 92, pages 3–15, 2004.

[6] 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. In *Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, pages 46–61. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[7] Brian C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[8] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public Transit Labeling. In *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science (LNCS)*, pages 273–285. Springer, 2015.

[9] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17, 2012.

[10] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 2012. Advance online publication.

[11] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly Simple and Fast Transit Routing. In *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science (LNCS)*, pages 43–54. Springer, Heidelberg, 2013.

[12] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science (LNCS)*, pages 71–82. Springer, Heidelberg, 2010.

[13] Pierre Hansen. Bicriterion Path Problems. In *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer, Heidelberg, 1980.

[14] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science (LNCS)*, pages 67–90. Springer, Heidelberg, 2007.

[15] Matthias Müller-Hannemann and Karsten Weihe. On the cardinality of the Pareto set in bicriteria shortest path problems. *Annals of Operations Research*, 147(1):269–286, 2006.

[16] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics*, 12:1, 2008.

[17] Ben Strasser and Dorothea Wagner. Connection Scan Accelerated. In *Algorithm Engineering and Experiments (ALENEX)*, pages 125–137, 2014.

A. Data Structures

We assign consecutive integer IDs, starting from 0, to stops, lines, and trips. For trips, we assign IDs such that trips of the same line are consecutive, with earlier trips having lower IDs. An array maps lines to their first trip. We store the remaining data using a forward star representation. For example, to store footpaths, we use two arrays, `Footpaths` and `FootpathIndex`. `Footpaths` contains information about footpaths, namely the destination stop and the length, for all footpaths. It is ordered such that footpaths starting at stop 0 come first, then footpaths starting at stop 1, etc. `FootpathIndex` contains, for each stop, the index of the first footpath starting at that stop (plus a sentinel value equal to the size of `Footpaths`). To examine footpaths at stop i, we iterate from `Footpaths[FootpathIndex[`i`]]` to `Footpaths[FootpathIndex[`$i+1$`]]`. The lines at each stop and the stops on each line are similarly stored. For arrival times, we store all times of the first trip in order, then all times of the second trip etc. and store the index of the first entry for each trip. Thus, $\tau_{\text{arr}}(t,i)$ is implemented by looking up `ArrivalTimes[TripTimeIndex[`t`]+`i`]`. Departure times use the same index array, as do transfers, albeit with an additional indirection: Transfers from trip t at index i can be found starting at `Transfers[TransferIndex[TripTimeIndex[`t`]+`i`]]`. For the transfers themselves, we only store the target trip and board index. Finally, we redundantly store an array mapping trips to lines and a second list of footpaths, this time indexed by their destination stops. Although not strictly necessary, these allow fast lookups during queries.