Public Transit Labeling[⋆]

Daniel Delling[1], Julian Dibbelt[2], Thomas Pajor[3], and Renato F. Werneck[4]

[1] Sunnyvale, USA, `daniel.delling@gmail.com`
[2] Karlsruhe Institute of Technology, Germany, `dibbelt@kit.edu`
[3] Microsoft Research, USA, `tpajor@microsoft.com`
[4] San Francisco, USA, `rwerneck@acm.org`

Abstract. We study the journey planning problem in public transit networks. Developing efficient preprocessing-based speedup techniques for this problem has been challenging: current approaches either require massive preprocessing effort or provide limited speedups. Leveraging recent advances in Hub Labeling, the fastest algorithm for road networks, we revisit the well-known time-expanded model for public transit. Exploiting domain-specific properties, we provide simple and efficient algorithms for the earliest arrival, profile, and multicriteria problems, with queries that are orders of magnitude faster than the state of the art.

1 Introduction

Recent research on route planning in transportation networks [5] has produced several speedup techniques varying in preprocessing time, space, query performance, and simplicity. Overall, queries on road networks are several orders of magnitude faster than on public transit [5]. Our aim is to reduce this gap.

There are many natural query types in public transit. An *earliest arrival* query seeks a journey that arrives at a target stop t as early as possible, given a source stop s and a departure time (e. g., "now"). A *multicriteria* query also considers the number of transfers when traveling from s to t. A *profile* query reports all quickest journeys between two stops within a time range.

These problems can be approached by variants of Dijkstra's algorithm [13] applied to a graph modeling the public transit network, with various techniques to handle time-dependency [18]. In particular, the *time-expanded* (TE) graph encodes time in the vertices, creating a vertex for every *event* (e. g., a train departure or arrival at a stop at a specific time). Newer approaches, like CSA [12] and RAPTOR [11], work directly on the timetable. Speedup techniques [5] such as Transfer Patterns [4,6], Timetable Contraction Hierarchies [14], and ACSA [20] use preprocessing to create auxiliary data that is then used to accelerate queries.

For aperiodic timetables, the TE model yields a *directed acyclic graph* (DAG), and several public transit query problems translate to reachability problems. Although these can be solved by simple graph searches, this is too slow for our application. Different methodologies exist to enable faster reachability computation [7,15,16,19,21,22,23]. In particular, the *2-hop labeling* [8] scheme associates

[⋆] Work done mostly while all authors were at Microsoft Research Silicon Valley.

with each vertex two labels (forward and backward); reachability (or shortest-path distance) can be determined by intersecting the source's forward label and the target's backward label. On continental road networks, 2-hop labeling distance queries take less than a microsecond [2].

In this work, we adapt 2-hop labeling to public transit networks, improving query performance by orders of magnitude over previous methods, while keeping preprocessing time practical. Starting from the time-expanded graph model (Section 3), we extend the labeling scheme by carefully exploiting properties of public transit networks (Section 4). Besides earliest arrival and profile queries, we address multicriteria and location-to-location queries, as well as reporting the full journey description quickly (Section 5). We validate our Public Transit Labeling (PTL) algorithm by careful experimental evaluation on large metropolitan and national transit networks (Section 6), achieving queries within microseconds.

2 Preliminaries

Let $G = (V, A)$ be a (weighted) *directed graph*, where V is the set of vertices and A the set of arcs. An arc between two vertices $u, v \in V$ is denoted by (u, v). A *path* is a sequence of adjacent vertices. A vertex v is *reachable* from a vertex u if there is a path from u to v. A *DAG* is a graph that is both directed and acyclic.

We consider *aperiodic* timetables, consisting of sets of stops S, events E, trips T, and footpaths F. *Stops* are distinct locations where one can board a transit vehicle (such as bus stops or subway platforms). *Events* are the scheduled departures and arrivals of vehicles. Each event $e \in E$ has an associated stop $\texttt{stop}(e)$ and time $\texttt{time}(e)$. Let $E(p) = \{e_0(p), \dots, e_{k_p}(p)\}$ be the list (ordered by time) of events at a stop p. We set $\texttt{time}(e_i(p)) = -\infty$ for $i < 0$, and $\texttt{time}(e_i(p)) = \infty$ for $i > k_p$. For simplicity, we may drop the index of an event (as in $e(p) \in E(p)$) or its stop (as in $e \in E$). A *trip* is a sequence of events served by the same vehicle. A pair of a consecutive departure and arrival events of a trip is a *connection. Footpaths* model transfers between nearby stops, each with a predetermined walking duration.

A journey planning algorithm outputs a set of *journeys.* A journey is a sequence of trips (each with a pair of pick-up and drop-off stops) and footpaths in the order of travel. Journeys can be measured according to several criteria, such as arrival time or number of transfers. A journey j_1 *dominates* a journey j_2 if and only if j_1 is no worse in any criterion than j_2. In case j_1 and j_2 are equal in all criteria, we break ties arbitrarily. A set of non-dominated journeys is called a *Pareto set.* Multicriteria Pareto optimization is NP-hard in general, but practical for natural criteria in public transit networks [11,12,17,18]. A journey is *tight* if there is no other journey between the same source and target that dominates it in terms of departure and arrival time, e. g., that departs later and arrives earlier.

Given a timetable, stops s and t, and a departure time τ, the (s, t, τ)-*earliest arrival* (EA) problem asks for an s–t journey that arrives at t as early as possible and departs at s no earlier than τ. The (s, t)-*profile* problem asks for a Pareto set of all tight journeys between s and t over the entire timetable period. Finally, the (s, t, τ)-*multicriteria* (MC) problem asks for a Pareto set of journeys departing at s no earlier than τ and minimizing the criteria arrival time and number of transfers. We focus on computing the *values* of the associated optimization criteria of the journeys (i. e., departure time, arrival times, number of transfers), which is enough for many applications. Section 5 discusses how the full journey description can be obtained with little overhead.

Our algorithms are based on the 2-hop labeling scheme for directed graphs [8]. It associates with every vertex v a *forward label* $L_f(v)$ and a *backward label* $L_b(v)$. In a *reachability labeling*, labels are subsets of V, and vertices $u \in L_f(v) \cup L_b(v)$ are *hubs* of v. Every hub in $L_f(v)$ must be reachable from v, which in turn must be reachable by every hub in $L_b(v)$. In addition, labels must obey the *cover property*: for any pair of vertices u and v, the intersection $L_f(u) \cap L_b(v)$ must contain at least one hub on a u–v path (if it exists). It follows from this definition that $L_f(u) \cap L_b(v) \neq \emptyset$ if and only if v is reachable from u.

In a *shortest path labeling*, each hub $u \in L_f(v)$ also keeps the associated distance $\text{dist}(u, v)$ (or $\text{dist}(v, u)$, for backward labels), and the cover property requires $L_f(u) \cap L_b(v)$ to contain at least one hub on a *shortest* u–v path. If labels are kept sorted by hub ID, a *distance label query* efficiently computes $\text{dist}(u, v)$ by a coordinated linear sweep over $L_f(u)$ and $L_b(v)$, finding the hub $w \in L_f(u) \cap L_b(v)$ that minimizes $\text{dist}(u, w) + \text{dist}(w, v)$. In contrast, a *reachability label query* can stop as soon as any matching hub is found.

In general, smaller labels lead to less space and faster queries. Many algorithms to compute labelings have been proposed [2,3,7,15,21,23], often for restricted graph classes. We leverage (as a black box) the recent RXL algorithm [9], which efficiently computes small shortest path labelings for a variety of graph classes at scale. It is a sampling-based greedy algorithm that builds labels one hub at a time, with priority to vertices that cover as many relevant paths as possible.

Different approaches for transforming a timetable into a graph exist (see [18] for an overview). In this work, we focus on the *time-expanded model*. Since it uses scalar arc costs, it is a natural choice for adapting the labeling approach. In contrast, the *time-dependent model* (another popular approach) associates functions with the arcs, which makes adaption more difficult.

3 Basic Approach

We build the time-expanded graph from the timetable as follows. We group all departure and arrival events by the stop where they occur. We sort all events at a stop by time, merging events that happen at the same stop and time. We then add a vertex for each unique event, a *waiting arc* between two consecutive events of the same stop, and a *connection arc* for each connection (between the corresponding departure and arrival event). The cost of arc (u, v) is $\texttt{time}(v) - \texttt{time}(u)$, i. e., the time difference of the corresponding events. To account for footpaths between two stops a and b, we add, from each vertex at stop a, a *foot arc* to the first reachable vertex at b (based on walking time), and vice versa. As events and vertices are tightly coupled in this model, we use the terms interchangeably.

Any label generation scheme (we use RXL [9]) on the time-expanded graph creates two (forward and backward) *event labels* for every vertex (event), enabling *event-to-event queries*. For our application *reachability* labels [21], which only store hubs (without distances), suffice. First, since all arcs point to the future, time-expanded graphs are DAGs. Second, if an event e is reachable from another event e' (i. e., $L_f(e') \cap L_b(e) \neq \emptyset$), we can compute the time to get from e' to e as $\texttt{time}(e) - \texttt{time}(e')$. In fact, *all* paths between two events have equal cost.

In practice, however, event-to-event queries are of limited use, as they require users to specify both departure *and* arrival times, one of which is usually unknown. Therefore, we discuss earliest arrival and profile queries, which *optimize* arrival time and are thus more meaningful. See Section 5 for multicriteria queries.

Earliest Arrival Queries. Given event labels, we answer an (s, t, τ)-EA query as follows. We first find the earliest event $e_i(s) \in E(s)$ at the source stop s that suits the departure time, i. e., with $\texttt{time}(e_i(s)) \geq \tau$ and $\texttt{time}(e_{i-1}(s)) < \tau$. Next, we search at the target stop t for the earliest event $e_j(t) \in E(t)$ that is reachable from $e_i(s)$ by testing whether $L_f(e_i(s)) \cap L_b(e_j(t)) \neq \emptyset$ and $L_f(e_i(s)) \cap L_b(e_{j-1}(t)) = \emptyset$. Then, $\texttt{time}(e_j(t))$ is the earliest arrival time. One could find $e_j(t)$ using linear search (which is simple and cache-friendly), but binary search is faster in theory and in practice. To accelerate queries, we *prune* (skip) all events $e(t)$ with $\texttt{time}(e(t)) < \tau$, since $L_f(e_i(s)) \cap L_b(e(t)) = \emptyset$ always holds in such cases. Moreover, to avoid evaluating $L_f(e_i(s))$ multiple times, we use *hash-based queries* [9]: we first build a hash set of the hubs in $L_f(e_i(s))$, then check the reachability for an event $e(t)$ by probing the hash with hubs $h \in L_b(e(t))$.

Profile Queries. To answer an (s, t)-profile query, we perform a coordinated sweep over the events at s and t. For the current event $e_i(s) \in E(s)$ at the source stop (initialized to the earliest event $e_0(s) \in E(s)$), we find the first event $e_j(t) \in E(t)$ at the target stop that is reachable, i. e., such that $L_f(e_i(s)) \cap L_b(e_j(t)) \neq \emptyset$ and $L_f(e_i(s)) \cap L_b(e_{j-1}(t)) = \emptyset$. This gives us the earliest arrival time $\texttt{time}(e_j(t))$. To identify the latest departure time from s for that earliest arrival event (and thus have a tight journey), we increase i until $L_f(e_i(s)) \cap L_b(e_j(t)) = \emptyset$, then add $(\texttt{time}(e_{i-1}(s)), \texttt{time}(e_j(t)))$ to the profile. We repeat the process starting from the events $e_i(s)$ and $e_{j+1}(t)$. Since we increase either i or j after each intersection test, the worst-case time to find all tight journeys is linear in the number of events (at s and t) multiplied by the size of their largest label.

4 Leveraging Public Transit

Our approach can be refined to exploit features specific to public transit networks. As described so far, our labeling scheme maintains reachability information for *all pairs* of events (by covering all paths of the time-expanded graph, breaking ties arbitrarily). However, in public transit networks we actually are only interested in *certain paths*. In particular, the labeling does *not* need to cover any path ending at a departure event (or beginning at an arrival event). We can thus discard forward labels from arrival events and backward labels from departure events.

Trimmed Event Labels. Moreover, we can disregard paths representing dominated journeys that depart earlier and arrive later than others (i. e., journeys that are not tight, cf. Section 2). Consider all departure events of a stop. If a certain hub is reachable from event $e_i(s)$, then it is also reachable from $e_0(s), \dots, e_{i-1}(s)$, and is thus potentially added to the forward labels of all these earlier events. In fact, experiments show that on average the same hub is added to 1.8–5.0 events per stop (depending on the network). We therefore compute *trimmed event labels* by discarding all but the latest occurrence of each hub from the forward labels. Similarly, we only keep the earliest occurrence of each hub in the backward labels. (Preliminary experiments have shown that we obtain very similar label sizes with a much slower algorithm that greedily covers tight journeys explicitly [2,9].)

Unfortunately, we can no longer just apply the query algorithms from Section 3 with trimmed event labels: if the selected departure event at s does not correspond to a tight journey toward t, the algorithm will not find a solution (though one might exist). One could circumvent this issue by also running the algorithm from subsequent departure events at s, which however may lead to quadratic query complexity in the worst case (for both EA and profile queries).

Stop Labels. We solve this problem by working with *stop labels*: For each stop p, we merge all forward event labels $L_f(e_0(p)), \dots, L_f(e_k(p))$ into a forward stop label $SL_f(p)$, and all backward event labels into a backward stop label $SL_b(p)$. Similar to distance labels, each stop label $SL(p)$ is a list of pairs $(h, \texttt{time}_p(h))$, each containing a hub and a time, sorted by hub. For a forward label, $\texttt{time}_p(h)$ encodes the latest departure time from p to reach hub h. More precisely, let h be a hub in an event label $L_f(e_i(p))$: we add the pair $(h, \texttt{time}(e_i(p)))$ to the stop label $SL_f(p)$ only if $h \notin L_f(e_j(p)), j > i$, i. e., only if h does not appear in the label of another event with a later departure time at the stop. Analogously, for backward stop labels, $\texttt{time}_p(h)$ encodes the earliest arrival time at p from h.

By restricting ourselves to these entries, we effectively discard dominated (non-tight) journeys to these hubs. It is easy to see that these stop labels obey a *tight journey cover property*: for each pair of stops s and t, $SL_f(s) \cap SL_b(t)$ contains at least one hub on each tight journey between them (or any equivalent journey that departs and arrives at the same time; recall from Section 2 that we allow arbitrary tie-breaking). This property does *not*, however, imply that the label intersection *only* contains tight journeys: for example, $SL_f(s)$ and $SL_b(t)$ could share a hub that is important for long distance travel, but not to get from s to t. The remainder of this section discusses how we handle this fact during queries.

Stop Label Profile Queries. To run an (s,t)-profile query on stop labels, we perform a coordinated sweep over both labels $SL_f(s)$ and $SL_b(t)$. For every matching hub h, i. e., $(h, \texttt{time}_s(h)) \in SL_f(s)$ and $(h, \texttt{time}_t(h)) \in SL_b(t)$, we consider the journey induced by $(\texttt{time}_s(h), \texttt{time}_t(h))$ for output. However, since we are only interested in reporting tight journeys, we maintain (during the algorithm) a tentative set of tight journeys, removing dominated journeys from it on-the-fly. (We found this to be faster than adding all journeys during the sweep and only discarding dominated journeys at the end.) We can further improve the efficiency of this approach in practice by (globally) reassigning hub IDs by the time of day. Note that every hub h of a stop label is still also an event and carries an event time $\texttt{time}(h)$. (Not to be confused with $\texttt{time}_s(h)$ and $\texttt{time}_t(h)$.) We assign sequential IDs to all hubs h in order of increasing $\texttt{time}(h)$, thus ensuring that hubs in the label intersection are enumerated chronologically. Note that this does not imply that journeys are enumerated in order of departure or arrival time, since each hub h may appear anywhere along its associated journey. However, preliminary experiments have shown that this approach leads to fewer insertions into the tentative set of tight journeys, reducing query time. Moreover, as in shortest path labels [9], we improve cache efficiency by storing the values for hubs and times separately in a stop label, accessing times only for matching hubs.

Overall, stop and event labels have different trade-offs: maintaining the profile requires less effort with event labels (any discovered journey is already tight), but fewer hubs are scanned with stop labels (there are no duplicate hubs).

Stop Label Earliest Arrival Queries. Reassigned hub IDs also enable fast (s, t, τ)-EA queries. We use binary search in $SL_f(s)$ and $SL_b(t)$ to find the earliest relevant hub h, i. e., with $\texttt{time}(h) \geq \tau$. From there, we perform a linear coordinated sweep as in the profile query, finding $(h, \texttt{time}_s(h)) \in SL_f(s)$ and $(h, \texttt{time}_t(h)) \in SL_b(t)$. However, instead of maintaining tentative profile entries $(\texttt{time}_s(h), \texttt{time}_t(h))$, we ignore solutions that depart too early (i. e., $\texttt{time}_s(h) < \tau$), while picking the hub h^* that minimizes the tentative best arrival time $\texttt{time}_t(h^*)$. (Note that $\texttt{time}(h) \geq \tau$ does not imply $\texttt{time}_s(h) \geq \tau$.) Once we scan a hub h with $\texttt{time}(h) \geq \texttt{time}_t(h^*)$, the tentative best arrival time cannot be improved anymore, and we stop the query. For practical performance, *pruning* the scan, so that we only sweep hubs h between $\tau \leq \texttt{time}(h) \leq \texttt{time}_t(h^*)$, is very important.

5 Practical Extensions

So far, we presented stop-to-stop queries, which report the departure and arrival times of the quickest journey(s). In this section, we address multicriteria queries, general location-to-location requests, and obtaining detailed journey descriptions.

Multicriteria Optimization and Minimum Transfer Time. Besides optimizing arrival time, many users also prefer journeys with fewer transfers. To solve the underlying multicriteria optimization problem, we adapt our labeling approach by (1) encoding transfers as arc costs in the graph, (2) computing shortest path labels based on these costs (instead of reachability labels on an unweighted graph), and (3) adjusting the query algorithm to find the Pareto set of solutions.

Reconsider the earliest arrival graph from Section 3. As before, we add a vertex for each unique event, linking consecutive events at the same stop with waiting arcs of cost 0. However, each connection arc (u, w) in the graph is subdivided by an intermediate *connection vertex* v, setting the cost of arc (u, v) to 0 and the cost of arc (v, w) to 1. By interpreting costs of 1 as leaving a vehicle, we can count the number of trips taken along any path. To model staying in the vehicle, consecutive connection vertices of the same trip are linked by zero-cost arcs.

A shortest path labeling on this graph now encodes the number of transfers as the shortest path distance between two events, while the duration of the journey can still be deduced from the time difference of the events. Consider a fixed source event $e(s)$ and the arrival events of a target stop $e_0(t), e_1(t), \ldots$ in order of increasing time. The minimum number of transfers required to reach the target stop t never increases with arrival times. (Hence, the whole Pareto set P of multicriteria solutions can be computed with a single Dijkstra run [18].)

We exploit this property to compute (s, t, τ)-EA multicriteria (MC) queries from the labels as follows. We initialize P as the empty set. We then perform an (s, t, τ)-EA query (with all optimizations described in Section 3) to compute the *fastest* journey in the solution, i. e., the one with most transfers. We add this journey to P. We then check (by performing distance label queries) for each subsequent event at t whether there is a journey with fewer transfers (than the most recently added entry of P), in which case we add the journey to P and repeat. The MC query ends once the last event at the target stop has been processed. We can stop earlier with the following optimization: we first run a distance label query on the *last* event at t to obtain the *smallest* possible number of transfers to travel from s to t. We may then already stop the MC query once we add a journey to P with this many transfers. Note that, since we do not need to check for domination in P explicitly, our algorithm maintains P in constant time per added journey.

Minimum Transfer Times. Transit agencies often model an entire station with multiple platforms as a single stop and account for the time required to change trips inside the station by associating a *minimum transfer time* $\text{mtt}(p)$ with each stop p. To incorporate them into the EA graph, we first locally replace each affected stop p by a *set* of new stops p^*, distributing *conflicting* trips (between which transferring is impossible due to $\text{mtt}(p)$) to different stops of p^*. We then add footpaths between all pairs of stops in p^* with length $\text{mtt}(p)$. A small set p^* can be computed by solving an appropriate coloring problem [10]. For the MC graph, we need not change the input. Instead, it is sufficient to *shift* each arrival event $e \in E(p)$ by adding $\text{mtt}(p)$ to $\texttt{time}(e)$ before creating the vertices.

Location-to-Location Queries. A query between arbitrary locations s^* and t^*, which may employ walking or driving as the first and last legs of the journey, can be handled by a two-stage approach. It first computes sets $\mathcal{S}$ and $\mathcal{T}$ of relevant stops near the origin s^* and destination t^* that can be reached by car or on foot. With that information, a *forward superlabel* [1] is built from all forward stop labels associated with $\mathcal{S}$. For each entry $(h, \texttt{time}_p(h)) \in SL_f(p)$ in the label of stop $p \in \mathcal{S}$, we adjust the departure time $\texttt{time}^*_s(h) = \texttt{time}_p(h) - \text{dist}(s^*, p)$ so that the journey starts at s^* and add $(h, \texttt{time}^*_s(h))$ to the superlabel. For duplicate hubs that occur in multiple stop labels, we keep only the latest departure time from s^*. This can be achieved with a coordinated sweep, always adding the next hub of minimum ID. A *backward superlabel* (for $\mathcal{T}$) is built analogously. For location-to-location queries, we then simply run our stop-label-based EA and profile query algorithms using the superlabels. In practice, we need not build superlabels explicitly but can simulate the building sweep during the query (which in itself is a coordinated sweep over two labels). A similar approach is possible for event labels. Moreover, point-of-interest queries (such as finding the closest restaurants to a given location) can be computed by applying known techniques [1] to these superlabels.

Journey Descriptions. While for many applications it suffices to report departure and arrival times (and possibly the number of transfers) per journey, sometimes a more detailed description is needed. We could apply known path unpacking techniques [1] to retrieve the full sequence of connections (and transfers), but in public transit it is usually enough to report the list of trips with associated transfer stops. We can accomplish that by storing with each hub the sequences of trips (and transfer stops) for travel between the hub and its label vertex.

6 Experiments

Setup. We implemented all algorithms in C++ using Visual Studio 2013 with full optimization. All experiments were conducted on a machine with two 8-core Intel Xeon E5-2690 CPUs and 384 GiB of DDR3-1066 RAM, running Windows 2008R2 Server. All runs are *sequential*. We use at most 32 bits for distances.

We consider four realistic inputs: the metropolitan networks of London (`data.london.gov.uk`) and Madrid (`emtmadrid.es`), and the national networks of Sweden (`trafiklab.se`) and Switzerland (`gtfs.geops.ch`). London includes all modes of transport, Madrid contains only buses, and the national networks contain both long-distance and local transit. We consider 24-hour timetables for the metropolitan networks, and two days for national ones (to enable overnight journeys). Footpaths were generated using a known heuristic [10] for Madrid; they are part of the input for the other networks. See Table 1 for size figures of the timetables and resulting graphs. The average number of unique events per stop ranges from 160 for Sweden to 644 for Madrid. (Recall from Section 3 that we merge all coincident events at a stop.) Note that no two instances dominate each other (w. r. t. number of stops, connections, trips, events per stop, and footpaths).

Preprocessing. Table 2 reports preprocessing figures for the unweighted earliest arrival graph (which also enables profile queries) and the multicriteria graph. For

Table 1. Size of timetables and the earliest arrival (EA) and multicriteria (MC) graphs.

						EA Graph		MC Graph	
Instance	Stops	Conns	Trips	Footp.	Dy.	$\lvert V\rvert$	$\lvert A\rvert$	$\lvert V\rvert$	$\lvert A\rvert$
London	20.8 k	5,133 k	133 k	45.7 k	1	4,719 k	51,043 k	9,852 k	72,162 k
Madrid	4.7 k	4,527 k	165 k	1.3 k	1	3,003 k	13,730 k	7,530 k	34,505 k
Sweden	51.1 k	12,657 k	548 k	1.1 k	2	8,151 k	34,806 k	20,808 k	93,194 k
Switzerland	27.1 k	23,706 k	2,198 k	29.8 k	2	7,979 k	49,656 k	31,685 k	170,503 k

Table 2. Preprocessing figures. Label sizes are averages of forward and backward labels.

	Earliest Arrival						**Multicriteria**			
		Event Labels			Stop Labels			Event Labels		
Instance	RXL [h:m]	Hubs p. lbl	Hubs p. stop	Space [MiB]	Hubs p. stop	Space [MiB]	RXL [h:m]	Hubs p. lbl	Hubs p. stop	Space [MiB]
London	0:54	70	15,480	1,334	7,075	1,257	49:19	734	162,565	26,871
Madrid	0:25	77	49,247	963	9,830	403	10:55	404	258,008	10,155
Sweden	0:32	37	5,630	1,226	1,536	700	36:14	190	29,046	12,637
Switzerland	0:42	42	11,189	1,282	2,970	708	61:36	216	58,022	12,983

earliest arrival (EA), preprocessing takes well below an hour and generates about one gigabyte, which is quite practical. Although there are only 37–70 hubs per label, the total number of hubs per stop (i. e., the combined size of all labels) is quite large (5,630–49,247). By eliminating redundancy (cf. Section 4), stop labels have only a fifth as many hubs (for Madrid). Even though they need to store an additional distance value per hub, total space usage is still smaller. In general, *average* labels sizes (though not total space) are higher for metropolitan instances. This correlates with the higher number of daily journeys in these networks.

Preprocessing the multicriteria (MC) graph is much more expensive: times increase by a factor of 26.2–54.8 for the metropolitan and 67.9–88 for the national networks. On Madrid, Sweden, and Switzerland labels are five times larger compared to EA, and on London the factor is even more than ten. This is immediately reflected in the space consumption, which is up to 26 GiB (London).

Queries. We now evaluate query performance. For each algorithm, we ran 100,000 queries between random source and target stops, at random departure times between 0:00 and 23:59 (of the first day). Table 3 reports detailed figures, organized in three blocks: event label EA queries, stop label EA queries, and profile queries (with both event and stop labels). We discuss MC queries later.

We observe that event labels result in extremely fast EA queries (6.9–14.7 µs), even without optimizations. As expected, pruning and hashing reduce the number of accesses to labels and hubs (see columns "Lbls." and "Hubs"). Although binary search cannot stop as soon as a matching hub is found (see the "=" column), it accesses fewer labels and hubs, achieving query times below 3 µs on all instances.

Using stop labels (cf. Section 4) in their basic form is significantly slower than using event labels. With pruning enabled, however, query times (3.6–6.2 µs) are within a factor of two of the event labels, while saving a factor of 1.1–2.4 in space. For profile queries, stop labels are clearly the best approach. It scans up to a factor of 5.1 fewer hubs and is up to 3.3 times faster, computing the profile of the full timetable period in under 80 µs on all instances. The difference in factors is due to the overhead of maintaining the Pareto set during the stop label query.

Comparison. Table 4 compares our new algorithm (indicated as *PTL*, for Public Transit Labeling) to the state of the art and also evaluates multicriteria queries. In

Table 3. Evaluating earliest arrival queries. Bullets (•) indicate different features: profile query (Prof.), stop labels (St. lbs.), pruning (Prn.), hashing (Hash), and binary search (Bin.). The column "=" indicates the average number of matched hubs.

					London				Sweden				Switzerland			
Prof.	St. lbs.	Prn.	Hash	Bin.	Lbls.	Hubs	=	[µs]	Lbls.	Hubs	=	[µs]	Lbls.	Hubs	=	[µs]
○	○	○	○	○	108.4	6,936	1	14.7	68.0	2,415	1	6.9	89.0	3,485	1	8.7
○	○	●	○	○	16.1	1,360	1	5.9	34.4	1,581	1	5.4	33.5	1,676	1	5.8
○	○	●	●	○	16.1	1,047	1	4.2	34.4	1,083	1	3.6	33.5	1,151	1	3.8
○	○	●	●	●	7.0	332	4	2.8	6.5	179	3	2.1	7.6	204	4	2.1
○	●	○	○	○	2.0	13,037	1,126	54.8	2.0	2,855	81	10.0	2.0	5,707	218	20.4
○	●	●	○	○	2.0	861	62	6.2	2.0	711	16	3.6	2.0	699	19	3.8
●	○	○	○	○	658.5	40,892	211	141.7	423.7	13,590	118	39.4	786.6	29,381	240	81.4
●	●	○	○	○	2.0	13,037	1,126	74.3	2.0	2,855	81	12.1	2.0	5,707	218	24.5

this experiment, PTL uses event labels with pruning, hashing and binary search for earliest arrival (and multicriteria) queries, and stop labels for profile queries. We compare PTL to CSA [12] and RAPTOR [11] (currently the fastest algorithms without preprocessing), as well as Accelerated CSA (ACSA) [20], Timetable Contraction Hierarchies (CH) [14], and Transfer Patterns (TP) [4,6] (which make use of preprocessing). Since RAPTOR always optimizes transfers (by design), we only include it for the MC problem. Note that the following evaluation should be taken with a grain of salt, as no standardized benchmark instances exist, and many data sets used in the literature are proprietary. Although precise numbers are not available for several competing methods, it is safe to say they use less space than PTL, particularly for the MC problem.

Table 4 shows that PTL queries are very efficient. Remarkably, they are faster on the national networks than on the metropolitan ones: the latter are smaller in most aspects, but have more frequent journeys (that must be covered). Compared to other methods, PTL is 2–3 orders of magnitude faster on London than CSA and RAPTOR for EA (factor 643), profile (factor 2,167), and MC (factor 203) queries. We note, however, that PTL is a point-to-point algorithm (as are ACSA, TP, and CH); for one-to-all queries, CSA and RAPTOR would be faster.

PTL has 1–2 orders of magnitude faster preprocessing and queries than TP for the EA and profile problems. On Madrid, EA queries are 233 times faster while preprocessing is faster by a factor of 48. Note that Sweden (PTL) and Germany (TP) have a similar number of connections, but PTL queries are 95 times faster. (Germany does have more stops, but recall that PTL query performance depends more on the frequency of trips.) For the MC problem, the difference is smaller, but both preprocessing and queries of PTL are still an order of magnitude faster than TP (up to 48 times for MC queries on Madrid).

Compared to ACSA and CH (for which figures are only available for the EA and profile problems), PTL has slower preprocessing but significantly faster queries (even when accounting for different network sizes).

Table 4. Comparison with the state of the art. Presentation largely based on [5], with some additional results taken from [6]. The first block of techniques considers the EA problem, the second the MC problem and the third the profile problem.

	Instance				**Criteria**					
Algorithm	Name	Stops $[\cdot 10^3]$	Conns $[\cdot 10^6]$	Dy.	Arr.	Tran.	Prof.	Prep. [h]	Jn.	Query [ms]
CSA [12]	London	20.8	4.9	1	●	○	○	—	n/a	1.8
ACSA [20]	Germany	252.4	46.2	2	●	○	○	0.2	n/a	8.7
CH [14]	Europe (LD)	30.5	1.7	p	●	○	○	< 0.1	n/a	0.3
TP [5]	Madrid	4.6	4.8	1	●	○	○	19	n/a	0.7
TP [6]	Germany	248.4	13.9	1	●	○	○	249	0.9	0.2
PTL	London	20.8	5.1	1	●	○	○	0.9	0.9	0.0028
PTL	Madrid	4.7	4.5	1	●	○	○	0.4	0.9	0.0030
PTL	Sweden	51.1	12.7	2	●	○	○	0.5	1.0	0.0021
PTL	Switzerland	27.1	23.7	2	●	○	○	0.7	1.0	0.0021
RAPTOR [11]	London	20.8	5.1	1	●	●	○	—	1.8	5.4
TP [5]	Madrid	4.6	4.8	1	●	●	○	185	n/a	3.1
TP [6]	Germany	248.4	13.9	1	●	●	○	372	1.9	0.3
PTL	London	20.8	5.1	1	●	●	○	49.3	1.8	0.0266
PTL	Madrid	4.7	4.5	1	●	●	○	10.9	1.9	0.0643
PTL	Sweden	51.1	12.7	2	●	●	○	36.2	1.7	0.0276
PTL	Switzerland	27.1	23.7	2	●	●	○	61.6	1.7	0.0217
CSA [12]	London	20.8	4.9	1	●	○	●	—	98.2	161.0
ACSA [20]	Germany	252.4	46.2	2	●	○	●	0.2	n/a	171.0
CH [14]	Europe (LD)	30.5	1.7	p	●	○	●	< 0.1	n/a	3.7
TP [6]	Germany	248.4	13.9	1	●	○	●	249	16.4	3.3
PTL	London	20.8	5.1	1	●	○	●	0.9	81.0	0.0743
PTL	Madrid	4.7	4.5	1	●	○	●	0.4	110.7	0.1119
PTL	Sweden	51.1	12.7	2	●	○	●	0.5	12.7	0.0121
PTL	Switzerland	27.1	23.7	2	●	○	●	0.7	31.5	0.0245

7 Conclusion

We introduced PTL, a new preprocessing-based algorithm for journey planning in public transit networks, by revisiting the time-expanded model and adapting the Hub Labeling approach to it. By further exploiting structural properties specific to timetables, we obtained simple and efficient algorithms that outperform the current state of the art on large metropolitan and country-sized networks by orders of magnitude for various realistic query types. Future work includes developing tailored algorithms for hub computation (instead of using RXL as a black box), compressing the labels (e. g., using techniques from [6] and [9]), exploring other hub representations (e. g., using trips instead of events, as in 3-hop labeling [21]), using multicore- and instruction-based parallelism for preprocessing and queries, and handling dynamic scenarios (e. g., temporary station closures and train delays or cancellations [5]).

References

1. I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. HLDB: Location-based services in databases. In *SIGSPATIAL*, pp. 339–348. ACM, 2012.
2. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, LNCS 7501, pp. 24–35. Springer, 2012.
3. T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, pp. 349–360. 2013.
4. H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. Fast routing in very large public transportation networks using transfer patterns. In *ESA*, LNCS 6346, pp. 290–301. Springer, 2010.
5. H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. Technical Report MSR-TR-2014-4, Microsoft Research, 2014.
6. H. Bast and S. Storandt. Frequency-based search for public transit. In *SIGSPATIAL*, pp. 13–22. ACM, 2014.
7. J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. In *SIGMOD*, pp. 193–204. 2013.
8. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.
9. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *ESA*, LNCS 8737, pp. 321–333. Springer, 2014.
10. D. Delling, B. Katz, and T. Pajor. Parallel computation of best connections in public transportation networks. *ACM JEA*, 17(4):4.1–4.26, 2012.
11. D. Delling, T. Pajor, and R. F. Werneck. Round-based public transit routing. *Transportation Science*, 2014. Accepted for publication.
12. J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Intriguingly simple and fast transit routing. In *SEA*, LNCS 7933, pp. 43–54. Springer, 2013.
13. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
14. R. Geisberger. Contraction of timetable networks with realistic transfers. In *SEA*, LNCS 6049, pp. 71–82. Springer, 2010.
15. R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. In *VLDB*, 6(14):1978–1989, 2013.
16. F. Merz and P. Sanders. PReaCH: A fast lightweight reachability index using pruning and contraction hierarchies. In *ESA*, LNCS 8737, pp. 701–712. Springer, 2014.
17. M. Müller–Hannemann and K. Weihe. On the cardinality of the Pareto set in bicriteria shortest path problems. *Ann. Oper. Res.*, 147(1):269–286, 2006.
18. E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM JEA*, 12(2.4):1–39, 2008.
19. S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pp. 1009–1020. 2013.
20. B. Strasser and D. Wagner. Connection scan accelerated. In *ALENEX*, pp. 125–137. SIAM, 2014.
21. Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pp. 1601–1606. ACM, 2013.
22. H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: Scalable reachability index for large graphs. In *VLDB*, 3(1):276–284, 2010.
23. A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: A total order approach. In *SIGMOD*, pp. 1323–1334. ACM, 2014.