Round-Based Public Transit Routing*

Daniel Delling
Microsoft Research Silicon Valley, dadellin@microsoft.com

Thomas Pajor†
Karlsruhe Institute of Technology, pajor@kit.edu

Renato F. Werneck
Microsoft Research Silicon Valley, renatow@microsoft.com

We study the problem of computing all Pareto-optimal journeys in a dynamic public transit network for multiple criteria, such as arrival time and number of transfers. Existing algorithms consider this as a graph problem, and solve it using variants of Dijkstra's algorithm. Unfortunately, this leads to either high query times or suboptimal solutions. We take a different approach. We introduce RAPTOR, our novel round-based public transit router. Unlike previous algorithms, it is not Dijkstra-based, looks at each route (such as a bus line) in the network at most once per round, and can be made even faster with simple pruning rules and parallelization using multiple cores. Because it does not rely on preprocessing, RAPTOR works in fully dynamic scenarios. Starting from arrival time and number of transfers as criteria, it can be easily extended to handle flexible departure times or arbitrary additional criteria. As practical examples we consider fare zones and reliability of transfers. When run on complex public transportation networks (such as London), RAPTOR computes all Pareto-optimal journeys between two random locations an order of magnitude faster than previous approaches, which easily enables interactive applications.

Key words: timetable information, public transit, shortest paths, multi-criteria optimization, dynamic programming, multi-core

1. Introduction
We study the problem of computing best journeys in public transit networks. A common approach to solve this problem is to model the network as a graph and to run a shortest path algorithm on it (see Pyrga et al. (2008) for a survey). At first glance, this is tempting: one can just use Dijkstra’s algorithm (1959), possibly augmented by a variety of speedup techniques that attempt to accelerate queries using auxiliary data computed in a preprocessing stage (see Delling et al. (2009b) or Sommer (2014) for an overview). Unfortunately, there are several downsides to this approach. Although known speedup techniques, such as the one by Abraham et al. (2011), can achieve speedups over Dijkstra’s algorithm of up to several millions on road networks, Bast (2009) shows that they fall short when applied to public transportation networks, which have a much different structure. More importantly, unlike in road networks, travel times are usually not enough to compute good journeys

* This paper is the full version of the extended abstract by Delling et al. (2012b) published at ALENEX 2012.
† This work was partly done while the author was at Microsoft Research Silicon Valley. Partially supported by DFG grant WA 654/16-1 and by the EU FP7/2007-2013 (DG INFSo.G4-ICT for Transport), under grant agreement no. 288094 (project eCOMPASS).
using public transit; other criteria, such as number of transfers and costs, can be just as important. Pyrga et al. (2008) handle this by reporting all Pareto-optimal journeys between two points using augmented versions of Dijkstra’s algorithm. This increases running times significantly and makes acceleration techniques even more complicated (see Berger et al. (2009), Delling et al. (2009a), Disser et al. (2008), Müller-Hannemann and Schnee (2007), Müller–Hannemann and Schnee (2009)). Berger et al. (2009) conclude that the efficient computation of multi-criteria journeys is an elusive goal. Moreover, the dynamic nature of public transit systems, with frequent delays and cancellations, makes preprocessing-based techniques impractical.

A feature shared by most previous approaches is that they operate on a graph. This complicates exploiting properties specific to transit networks, such as the fact that vehicles operate on predefined lines. One exception is the concept of transfer patterns introduced by Bast et al. (2010): It can answer queries fast, but its preprocessing effort (thousands of CPU hours) is so high that optimality must be dropped in order to make it practical. It is unclear how it can be used in a dynamic scenario.

This article introduces RAPTOR, our novel Round-bAsed Public Transit Optimized Router. For two given stops, it computes all Pareto-optimal journeys—minimizing the arrival time and the number of transfers made—between them. Unlike previous approaches, RAPTOR is not Dijkstra-based. Instead, it operates in rounds, one per transfer, and computes arrival times by traversing every route (such as a bus line) at most once per round. Our algorithm boils down to a dynamic program with simple data structures and excellent memory locality. Unlike Dijkstra-based algorithms, which are notoriously hard to parallelize (see Madduri et al. (2009) and Meyer and Sanders (2003)), with RAPTOR we can easily distribute independent routes among multiple CPU cores.

We also introduce two extensions of RAPTOR. The first, McRAPTOR, generalizes RAPTOR to handle more criteria, beyond arrival time and transfers. As an example we use fare zones, a common pricing model, and reliability of transfers. The second extension we propose, rRAPTOR, computes bicriteria range queries, which output full Pareto sets of journeys for all departures within a time range. Because our algorithms do not rely on preprocessing, they are fully dynamic, easily handling delays, cancellations, or route changes. Moreover, since our algorithms output sets of journeys, they immediately provide sensible alternatives to the user. Finally, our experiments show that on our main benchmark instance, the full network of London, with over 20 thousand stops and 5 million daily departure events, RAPTOR computes all Pareto-optimal bicriteria journeys between two random locations in under 6 ms (on a single core using standard server hardware). This is fast enough for practical use in interactive applications.

This article is organized as follows. Section 2 has formal problem definitions and discusses existing solutions that are relevant to this work. Section 3 introduces RAPTOR, our main contribution, and Section 4 shows how to extend it to handle more criteria (McRAPTOR) and range queries (rRAPTOR). Section 5 reports experimental results. Finally, Section 6 has concluding remarks.

2. Preliminaries
Our algorithms work on a timetable \((\Pi, S, T, R, F)\) where \(\Pi \subset \mathbb{N}_0\) is the period of operation (think of it as the seconds of a day), \(S\) is a set of stops, \(T\) a set of trips, \(R\) a set of routes, and \(F\) a set of transfers (or foot-paths). Each stop in \(S\) corresponds to a distinct location in the network where one can board or get off a vehicle (bus, tram, train, etc.). Typical examples are bus stops and train platforms. Each trip \(t \in T\) represents a sequence of stops a specific vehicle (train, bus, subway, ...) visits along a line. At each stop in the sequence, it may drop off or pick up passengers. Moreover, each stop \(p\) in a trip \(t\) has associated arrival and departure times \(\tau_{\text{arr}}(t, p), \tau_{\text{dep}}(t, p) \in \Pi\), with \(\tau_{\text{arr}}(t, p) \leq \tau_{\text{dep}}(t, p)\). The first and last stops of a trip have undefined arrival and departure times, respectively. The trips in \(T\) are partitioned into routes: Each route in \(R\) consists of the trips that share the same sequence of stops. Also, we require the trips within a route to be non-overtaking (i.e., no trip overtakes any other within the same route). Typically, there are many more trips than
routes. Foot-paths in \( F \) model walking connections (or transfers) between stops. Each transfer consists of two stops \( p_1 \) and \( p_2 \) with an associated constant walking time \( \ell(p_1, p_2) \). We require \( F \) to be transitive: If \( p_1 \) and \( p_2 \) are indirectly connected by foot-paths, \((p_1, p_2)\) is contained in \( F \) as well. Finally, a stop \( p \in S \) may have an associated minimum change time \( \tau_{ch}(p) \), the minimum time required to change trips at \( p \) (due to long walking distances within \( p \), for example).

Any journey-planning algorithm operating on a timetable outputs a set of journeys \( J \). A journey is defined as a sequence of trips and foot-paths in the order of travel. In addition, each trip in the sequence is associated with two stops, corresponding to the pick-up and drop-off points. Note that a journey containing \( k \) trips has exactly \( k - 1 \) transfers. Journeys are associated with several optimization criteria. We say a journey \( J_1 \) dominates a journey \( J_2 \), denoted by \( J_1 \preceq J_2 \), if \( J_1 \) is no worse in any criterion than \( J_2 \). A set of pairwise nondominating journeys is a \( \text{Pareto set} \). In our algorithms we use \( \text{labels} \) (often associated with stops) for intermediate journeys. The definition of domination translates to labels naturally.

The simplest problem we consider is the \textit{Earliest Arrival Problem}. Given a source stop \( p_s \), a target stop \( p_t \), and a departure time \( \tau \), it asks for a journey that departs \( p_s \) no earlier than \( \tau \), and arrives at \( p_t \) as early as possible. The \textit{Multi-Criteria Problem} is a generalization with more than one optimization criterion (always including earliest arrival time). More precisely, it asks for a full Pareto set of journeys, with each journey leaving \( p_s \) no earlier than \( \tau \). For example, one journey can arrive at 4 p.m. with 2 transfers, and another one at 3:30 p.m. with 3 transfers. Finally, the \textit{Range Problem} asks for a set of journeys with varying departure times. More precisely, for every departure time \( \tau \in \Delta \) where \( \Delta \subseteq \Pi \), we ask for a journey that leaves \( p_s \) no later than \( \tau \) and arrives at \( p_t \) as early as possible. It is a special case of the multi-criteria problem using arrival and departure time as criteria with domination \( J_1 \preceq J_2 \) if \( \tau_{dep}(J_1) \geq \tau_{dep}(J_2) \) and \( \tau_{arr}(J_1) \leq \tau_{arr}(J_2) \).

Note that each of these problems we can define a \textit{latest departure} variant, which is equivalent. Here, we are given an arrival time \( \tau \) at the target stop \( p_t \) as input, and ask for journeys that arrive no later than \( \tau \) and depart from \( p_s \) as late as possible.

In practice, one is often interested in journeys between arbitrary \textit{locations} rather than stops. This problem can be solved by temporarily defining (at query time) \textit{virtual} source and target stops \( p_s, p_t \), which are connected by appropriate foot-paths (e.g., determined via an auxiliary pedestrian network) to their closest stops from \( S \). The query is then simply run between these virtual stops \( p_s \) and \( p_t \).

### 2.1. Existing Graph-Based Approaches

Previous work on journey planning focused on graph-based models, in particular the time-expanded and the time-dependent approaches surveyed by Pyrga et al. (2008). The \textit{time-expanded} approach models each event in the timetable (e.g., departure or arrival of a trip at a stop) by a separate vertex. This results in large graphs yielding poor query performance. In contrast, the \textit{time-dependent model} uses time-dependent functions to group several trips along edges. In general, the size of the resulting graph is linear in the number of stops and routes, which can be orders of magnitude smaller than the number of events. We therefore focus on the more efficient time-dependent model.

The time-dependent \textit{route model} creates a stop-vertex for each stop \( p \in S \). In addition, a route-vertex \( r_p \) is created for each stop \( p \) and every route \( r \in R \) that serves \( p \). Edges are added within each stop between the stop-vertex and every route-vertex (and vice versa) to allow transfers. Their constant weights represent the transfer time (if any) between trips serving \( p \). To model trips, time-dependent edges are added between route-vertices. More precisely, if a trip \( t \in T \) serves two subsequent stops \( p_1, p_2 \) along its route \( r \in R \), an edge from \( \tau_{p_1} \) to \( \tau_{p_2} \) is required. This edge is time-dependent, and its function reflects a travel time of \( \tau_{dep}(t, p_2) - \tau_{dep}(t, p_1) \) at departure time \( \tau_{dep}(t, p_1) \). Edge costs can be modeled as special piecewise linear functions that can be efficiently evaluated, as shown by Delling (2011) and Delling et al. (2012a). To incorporate foot-paths, for each \((p_i, p_j)\) in \( F \) a time-independent edge is added between the corresponding stop-vertices, weighted by \( \ell(p_i, p_j) \). The time-dependent \textit{station model} is a condensed version with only one vertex per stop, combined with time-dependent edges. Although it leads to smaller graphs, Berger et al. (2009) and Geisberger (2010) show that it complicates the query algorithms in order to incorporate transfers correctly.
Algorithms. To solve the earliest arrival problem from a source stop $p_s$ at time $\tau$ on the time-dependent model, we can use an augmented variant of Dijkstra’s algorithm (1959). It scans vertices in increasing order of arrival time, and evaluates each edge $e = (u, v)$ using the (already determined) arrival time at $u$. The algorithm stops as soon as the target stop-vertex is scanned. We refer to this algorithm as Time-Dijkstra (TD).

The Multi-Criteria Problem on the time-dependent route model can be solved by a multi-label-correcting algorithm (MLC) as shown by Pyrga et al. (2008). It handles arbitrary criteria that are modeled by edge costs and generalizes Dijkstra’s algorithm as follows. Labels now contain multiple values, one for each optimization criterion. Each vertex $u$ maintains a bag $B_u$ representing a Pareto set of nondominated labels. The algorithm maintains a priority queue of unprocessed labels, ordered lexicographically. Each step extracts the minimum label $L_v$ from the queue and processes the corresponding vertex $u$. For every edge $(u, v)$, a new label $L_v$ is created. If $L_v$ is not dominated by a label in $B_v$, $L_v$ is inserted into $B_v$ (possibly eliminating some labels in $B_v$). The priority queue is updated accordingly. Disser et al. (2008) present several improvements to MLC: (1) hopping-reduction avoids propagating a label back to the vertex it originated from; (2) label-forwarding does not use the priority queue for new labels that have no increase in cost; and (3) target-pruning eliminates labels $L$ that are dominated by a label from the target vertex’s bag. We do not use goal-direction for MLC; although Disser et al. (2008) show it may be helpful for long-distance rail networks, we focus on dense urban networks, where it does not help (see Bauer et al. (2011)). Even with these improvements, MLC is much more costly than plain Time-Dijkstra: not only must it scan the same vertex multiple times, but it also has to handle more complicated data structures, such as bags.

When the only additional criterion (besides arrival time) is number of transfers, Brodal and Jacob (2004) propose the simpler Layered Dijkstra (LD) algorithm. It also works for other second criteria, as long as they are discrete. As an example, let $K$ be a bound on the number of transfers. During preprocessing, the graph is copied into $K$ layers, with transfer edges rewired to point to the layer directly above. Running Time-Dijkstra from the source vertex on the bottom layer results for each $k \leq K$ in a journey having exactly $k$ transfers for vertices on layer $k$. Instead of copying the graph, we can use an array of $K$ labels for each vertex and read/write the $k$-th entry in layer $k$. Moreover, to implement domination, a label at vertex $u$ on layer $k$ can be pruned if there exists a label with earlier arrival time at $u$ on a layer lower than $k$. Similarly, the label can be pruned if the target vertex has a label with smaller arrival time on any layer up to $k$. We can drop the requirement for the bound $K$ as input by dynamically extending the labels whenever necessary.

A known efficient solution to the Range Problem is the Self-Pruning Connection-Setting algorithm (SPCS) introduced by Delling et al. (2012a). It first determines all trips departing at $p_s$. Then, it initializes a priority queue with all these trips, which uses arrival times as keys. The search algorithm is very similar to TD, with additional pruning. When a label $L$ is extracted from the queue at vertex $v$, it can be pruned if $v$ has already been scanned with a label $L'$ for which $\tau_{dep}(L') \geq \tau_{dep}(L)$ holds. Target pruning can be incorporated by keeping track of the maximum departure time of any journey that reached the target; any journey with a later departure time (anywhere in the graph) can then be pruned. A multi-core version of this algorithm partitions the departing trips of $p_s$ among the available cores, which then run SPCS independently. In the end, the resulting journeys are merged, and dominated ones discarded. Note that SPCS does not consider number of transfers as a criterion.

3. Our Approach: RAPTOR
We now introduce the basic version of RAPTOR, our algorithm. It solves the bicriteria problem minimizing arrival time and number of transfers—like LD or MLC. However, our method is not based on Dijkstra’s algorithm. In fact, it does not even need a priority queue. Let $p_s \in S$ be the source stop, and $\tau \in \Pi$ the departure time. Recall that our goal is to compute for every $k$ a
We maintain the following invariant: at the beginning of round \( k \), walking path, if one exists. The algorithm can be stopped after round \( p \) with each stop its stops in order until we find a stop. More precisely, during round \( k \), it suffices to traverse only routes that contain at least one stop.

The algorithm works in rounds. Round \( k \) computes the fastest way of getting to every stop with at most \( k - 1 \) transfers (i.e., by taking at most \( k \) trips). Note that some stops may not be reachable at all. To explain the algorithm, we bound the number of rounds by \( K \) (which can be dynamically extended during the algorithm, if necessary). More precisely, the algorithm associates with each stop \( p \) a multilabel \( (\tau_0(p), \tau_1(p), \ldots, \tau_k(p)) \), where \( \tau_i(p) \) represents the earliest known arrival time at \( p \) with up to \( i \) trips. All values in all labels are initialized to \( \infty \). We then set \( \tau_0(p_u) = \tau \). We maintain the following invariant: at the beginning of round \( k \) (for \( k \geq 1 \)), the first \( k \) entries in \( \tau(p) \) (from \( \tau_0(p) \) to \( \tau_{k-1}(p) \)) are correct, i.e., entry \( \tau_i(p) \) represents the earliest arrival time at \( p \) using at most \( i \) trips. The remaining entries are set to \( \infty \). The goal of round \( k \) is to compute \( \tau_k(p) \) for all \( p \). It does so in three stages.

The first stage of round \( k \) sets \( \tau_k(p) = \tau_{k-1}(p) \) for all stops \( p \); this sets an upper bound on the earliest arrival time at \( p \) with at most \( k \) trips.

The second stage then processes each route in the timetable exactly once. Consider a route \( r \), and let \( T(r) = \{ t_0, t_1, \ldots, t_{|T(r)|-1} \} \) be the sequence of trips that follow route \( r \), from earliest to latest. When processing route \( r \), we consider journeys where the last \((k\text{-th})\) trip taken is in route \( r \). Recall that \( \tau_{ch}(p_i) \) is the minimum change time at \( p_i \) required for changing trips. Let \( et(r, p_i) \) be the earliest trip in route \( r \) that one can catch at stop \( p_i \), i.e., the earliest trip \( t \) such that \( \tau_{dep}(t, p_i) \geq \tau_{k-1}(p_i) + \tau_{ch}(p_i) \). Note that (1) this trip may not exist, in which case \( et(r, p_i) \) is undefined, and (2) in the first round we do not need to add the minimum change time \( \tau_{ch}(p_i) \). To process the route, we visit its stops in order until we find a stop \( p_i \) such that \( et(r, p_i) \) is defined. This is when we can “hop on” the route. Let the corresponding trip \( t \) be the current trip for \( k \). We keep traversing the route. For each subsequent stop \( p_j \), we can update \( \tau_i(p_j) \) using this trip. To reconstruct the journey, we set a parent pointer to the stop at which \( t \) was boarded. Moreover, we may need to update the current trip for \( k \): At each stop \( p_i \), it may be possible to catch an earlier trip (because a quicker path to \( p_i \) has been found in a previous round). Thus, we have to check if \( \tau_{ch}(p_i) < \tau_{dep}(t, p_i) \) and update \( t \) by recomputing \( et(r, p_i) \). Again, we do not need to consider the minimum change time \( \tau_{ch}(p_i) \) in the first round.

Finally, the third stage of round \( k \) considers foot-paths. For each foot-path \((p_i, p_j) \in F\) it sets \( \tau_k(p_j) = \min \{ \tau_k(p_i), \tau_i(p_i) + \ell(p_i, p_j) \} \). Note that since \( F \) is transitive, we always find the fastest walking path, if one exists. The algorithm can be stopped after round \( k \), if no label \( \tau_k(p) \) was improved.

The worst-case running time of our algorithm can be bounded as follows. In every round, we scan each route \( r \in R \) at most once. If \( |r| \) is the number of stops along \( r \), then we look at \(\sum_{r \in R} |r| \) stops in total to process the route. For each stop, we must find the earliest trip \( et(r, \cdot) \). If we keep the list of trips serving \( r \) sorted by time, while traversing \( r \) we can find all \( et(r, \cdot) \) values with a single sweep over this list, since \( et(r, \cdot) \) can only decrease. In total, RAPTOR takes \( O(K(\sum_{r \in R} |r| + |T| + |F|)) \) time, where \( K \) is the number of rounds. Note that the running time per round is potentially \textit{sublinear} in the size of the input: The work per route is linear in the number of trips and the size of the route, but most of the departure/arrival times associated with individual trips are not considered. Constant access to the stops along routes and the arrival and departure times of specific trips can be achieved by a few arrays (see the appendix for details). In contrast, a similar analysis for the route-based model reveals that MLC and LD are slower by at least a logarithmic factor, due to the priority queues.

### 3.1. Improvements

Having set up the basic version of our algorithm, we now propose some optimizations.

Iterating over all routes in every round seems wasteful. Indeed, there is no need to traverse routes that cannot be reached by the previous round, since there is no way to “hop on” to any of its trips. More precisely, during round \( k \), it suffices to traverse only routes that contain at least one stop.
reached with exactly $k - 1$ trips. To see why, consider a route whose last improvement happened at round $k' < k - 1$. The route was visited again during round $k' + 1 < k$, and no stop along the route improved. There is no point in traversing it again until at least one of its stops improves (due to some other route). To implement this version of the algorithm, we mark during round $k - 1$ the stops $p_i$ for which we improved the arrival time $\tau_k(p_i)$. At the beginning of round $k$, we loop through all marked stops to find all routes that contain them. Only routes from the resulting set $Q$ are considered for scanning in round $k$. Moreover, since the marked stops are exactly those where we potentially “hop on” a trip in round $k$, we only have to traverse a route beginning at the earliest marked stop it contains. To enable this, while adding routes to $Q$, we also remember the earliest marked stop in each route. See also Figure 1.

Another useful technique is **local pruning**. For each stop $p_i$, we keep a value $\tau_\ast(p_i)$ representing the earliest known arrival time at $p_i$. Since we are only interested in Pareto-optimal paths, we now only mark a stop during route traversal at round $k$ when the arrival time with $k$ trips is earlier than $\tau_\ast(p_i)$. Local pruning, thus, allows us to drop the first stage of each round (copying the labels from the previous round): The value $\tau_\ast(p_i)$ automatically keeps track of the earliest possible time to get to $p_i$.

Note that, as described, RAPTOR computes journeys to all stops of the network (which may be useful in some applications). If we are only interested in journeys to a target stop $p_t$, the performance of RAPTOR can be improved by **target pruning**: During round $k$, there is no need to mark stops whose arrival times are greater than $\tau_\ast(p_t)$ (the best known arrival time at $p_t$). A description in pseudocode including marking and pruning can be found in Algorithm 1.

### 3.2. Transfer Preferences and Strict Domination

Berger et al. (2010) show that MLC can be extended to the scenario where one is interested in Pareto-optimal solutions with respect to **strict domination**, which means one journey only dominates another if it is strictly better in at least one criterion. This leads to bigger Pareto sets. The motivation for this extension is to output journeys that have transfers at preferred locations. The best journey can be determined in a postprocessing step by looking at all possible combinations of transfer locations.

RAPTOR can handle transfer preferences without extending the Pareto set, as follows: When scanning a route $r$ in round $k$ while using trip $t$, we keep track of the stop (among those where $t$ can be boarded) that maximizes the transfer preference value. Then, whenever we write a label $\tau_k(p)$, we set its parent pointer immediately to the stop with the maximum preference encountered so far. For applications where strict domination is preferred, we propose the following heuristic. Whenever we write a label $\tau_k(p)$, instead of keeping a single parent pointer we add pointers to every stop $p'$ where the current trip $t$ could be boarded, i.e., to those stops $p'$ where $\tau_{k-1}(p') \leq \tau_{\text{dep}}(t, p')$ hold. To avoid dynamic allocation at every stop, we can write tuples of parent pointer and arrival time to a separate log in memory. Since parent pointers may change, we reconstruct the final parent pointers by linearly sweeping over the log in a postprocessing step. Note that some journeys from...
Algorithm 1 RAPTOR

1: procedure RAPTOR(source and target stops \(p_s, p_t\), and departure time \(\tau\))
2:     for all \(i\) do \(\triangleright\) initialization of the algorithm
3:         \(\tau_i(\cdot) \leftarrow \infty\)
4:     end for
5:     \(\tau^*(\cdot) \leftarrow \infty\)
6:     \(\tau_0(p_s) \leftarrow \tau\)
7:     \(\tau^*(p_s) \leftarrow \tau\)
8:     mark \(p_s\)
9:     for all \(k \leftarrow 1, 2, \ldots\) do \(\triangleright\) accumulate routes serving marked stops from previous round
10:        clear \(Q\)
11:        for all marked stops \(p\) do
12:            for all routes \(r\) serving \(p\) do
13:                if \((r, p') \in Q\) for some stop \(p'\) then
14:                    substitute \((r, p')\) by \((r, p)\) in \(Q\) if \(p\) comes before \(p'\) in \(r\)
15:                else
16:                    add \((r, p)\) to \(Q\)
17:                end if
18:            end for
19:        unmark \(p\)
20:    end for
21:    for all routes \((r, p) \in Q\) do \(\triangleright\) traverse each route
22:       \(t \leftarrow \bot\) \(\triangleright\) the current trip
23:       for all stops \(p_i\) of \(r\) beginning with \(p\) do
24:           if \(t \neq \bot\) and \(\text{arr}(t, p_i) < \min\{\tau^*(p_i), \tau^*(p_t)\}\) then
25:               \(\tau_k(p_i) \leftarrow \tau_{\text{arr}}(t, p_i)\)
26:               \(\tau^*(p_i) \leftarrow \tau_{\text{arr}}(t, p_i)\)
27:               mark \(p_i\)
28:           end if
29:           if \(\tau_{k-1}(p_i) + \tau_{\text{ch}}(p_i) < \tau_{\text{dep}}(t, p_i)\) then \(\triangleright\) can we catch an earlier trip at \(p_i\)?
30:               \(t \leftarrow \text{et}(r, p_i)\)
31:           end if
32:       end for
33:    end for
34:    for all marked stops \(p\) do \(\triangleright\) look at foot-paths
35:        for all foot-paths \((p, p') \in \mathcal{F}\) do
36:            \(\tau_k(p') \leftarrow \min\{\tau_k(p'), \tau_k(p) + \ell(p, p')\}\)
37:        mark \(p'\)
38:    end for
39:    end for
40:    if no stops are marked then \(\triangleright\) stopping criterion
41:       stop
42:    end if
43:    end for
44: end procedure
the exact (strict) Pareto set may be missed by this heuristic, since it only keeps parent pointers for one (arrival) trip per stop.

3.3. Parallelization

While Dijkstra-based algorithms are notoriously hard to parallelize (see e.g. Meyer and Sanders (2003), Madduri et al. (2009)), RAPTOR can be easily extended to work in parallel. Most of the work is spent dealing with individual routes, which are processed in no particular order. If several CPU cores are available, each can handle a different subset of the routes (in each round). During round \( k \), however, multiple threads may attempt to write simultaneously to the same memory location \( \tau_k(p) \). Race conditions could be avoided with standard synchronization primitives (such as locks), but that can be costly. Instead, we propose two lock-free parallelization approaches for our algorithm.

If the hardware architecture ensures atomic writes for the values of \( \tau_k(p) \), we can just “blindly” write to \( \tau_k(p) \). The corresponding memory position will always have a valid upper bound on the arrival time at \( p \); even if a thread could not successfully write a better value. To restore consistency after the route scanning stage, each thread maintains a log of its update attempts on any value \( \tau_k(p) \). At the end of the round, the master thread uses the logs to correct the labels sequentially. The same technique can also be used to keep \( \tau_\ast(p) \) consistent. We call this approach update log parallelization.

If atomic writes are not guaranteed, we can still avoid locks with the conflict graph approach. We use the fact that any two routes that have no stop in common can be safely scanned in parallel. In a quick preprocessing step, we build an undirected conflict graph \( G \), where vertices correspond to routes and there are edges between any two routes that share at least one stop. We then greedily color the routes such that no two adjacent routes share the same color. Routes with the same color can always be processed independently.

To implement this approach efficiently, we order the routes according to their colors (with ties broken arbitrarily) to obtain a sequence \( R = \{r_0, r_1, \ldots, r_{|R|-1}\} \). We then compute for every route \( r_i \) a dependent route \( \text{pre}(r_i) = r_j \), defined as the highest-indexed conflicting route that appears before \( i \) in the order \( j < i \). The route scanning stage is now modified as follows. When a core becomes available, it is assigned the next (in index order) available unprocessed route \( r_i \) and waits (in a busy loop) until all routes up to \( \text{pre}(r_i) \) have been fully processed. (Since in practice conflicting routes are on average far apart in the list, the overhead of busy waiting is negligible.) Once this happens, it can safely process \( r_i \); Conflicting routes \( r_j \) with \( j < i \) have already been processed, and those with \( j > i \) will wait until \( r_i \) is finished. Threads can use shared memory to communicate to others that their own routes have been processed, ensuring no two threads ever write to the same location. Unmarked routes can be skipped and set to processed. In dynamic scenarios, route dependencies must be updated whenever a route changes, but this takes negligible time.

3.4. Timetable Compression

Realistic timetables are often (partially) periodic: Trips of the same route operate with fixed frequencies over certain timespans during the day. For example, a bus line operates every 10 minutes from 6 am to 4 pm, and every 15 minutes from 4 pm to 9 pm. Up to now, we stored each individual trip explicitly. To save memory, we now exploit such periodicities to compress our data structures in a quick preprocessing step. For each route \( r \) we consider its trips in order, from earliest to latest, and group contiguous sequences of trips iff each pair of subsequent trips (in the group) shares the same departure/arrival time interval at every stop along the route. More formally, for subsequent trips \( t_i, t_{i+1} \) and \( t_j, t_{j+1} \) (in the group) we require \( \tau_{\text{arr}}(t_{i+1}, p) - \tau_{\text{arr}}(t_i, p), \tau_{\text{arr}}(t_{j+1}, p) - \tau_{\text{arr}}(t_j, p), \tau_{\text{dep}}(t_{i+1}, p) - \tau_{\text{dep}}(t_i, p), \) and \( \tau_{\text{dep}}(t_{j+1}, p) - \tau_{\text{dep}}(t_j, p) \) to be equal among all stops \( p \) of the route \( r \). We refer to this time interval as the periodicity of the trip group. We now delete all but the first (earliest) trip \( t \) in each group and additionally store with \( t \) its periodicity \( \text{per}(t) \) and the size of its group as \( \text{size}(t) \). Note that for aperiodic trips we have \( \text{size}(t) = 1 \), and \( \text{per}(t) \) is undefined.

To make use of the compressed timetable, we modify RAPTOR to expand trips on the fly. When processing a route \( r \), we not only keep track of the current trip \( t \), but also of an offset \( 0 \leq i < \text{size}(t) \).
Whenever we evaluate the departure time of $t$ at a stop $p$, we compute $\tau_{dep}(t, p) + i \cdot \text{per}(t)$ (the arrival time is computed analogously). Accordingly, $\text{et}(r, p)$ now returns a trip/offset pair $(t, i)$. We refer to RAPTOR on a frequency-compressed timetable as FRAPTOR.

4. Extensions

In this section we show how RAPTOR can be extended to handle additional criteria, such as fare zones and reliability. We call the resulting algorithm McRAPTOR (for “more criteria RAPTOR”). For the special case of bicriteria range queries, Section 4.2 will present a tailored extension, which we call rRAPTOR.

4.1. More Criteria: McRAPTOR

Recall that plain RAPTOR stores exactly one value $\tau_i(p)$ per stop and round. To extend the algorithm to more criteria, we keep multiple nondominated labels for each stop $p$ in round $k$, similarly to MLC (cf. Section 2.1). We store these labels in bags, denoted by $B_k(p)$.

The algorithm is then modified as follows. When processing a route $r$, we first create an empty route bag $B_r$ which keeps track of all good journeys whose last trip is in route $r$. Therefore, each label $L$ in the route bag has an associated active trip $t(L)$. When traversing the stops of $r$ in order, we process each stop $p$ in three steps. The first step updates the arrival times of every label $L \in B_r$ to the arrival times of their associated trips $t(L)$ at $p$. Note that if two labels have the same associated trip, one might be eliminated. In the second step, we merge $B_r$ into $B_k(p)$ by copying all labels from $B_r$ to $B_k(p)$, and discarding dominated labels in $B_k(p)$. The final step merges $B_{k-1}(p)$ into $B_r$ and assigns trips to all newly-added labels.

The foot-paths stage of the algorithm is also modified. When looking at a foot-path $(p_i, p_j)$, we create a temporary copy of $B_k(p_i)$ and add $\ell(p_i, p_j)$ to the arrival time of every label. Then we merge this bag into $B_k(p_j)$.

We also can adapt local and target pruning. Similarly to $\tau^*$ in RAPTOR, we keep for every stop $p$ a best bag $B^*(p)$ that represents the nondominated set of labels over all previous rounds. Thus, whenever we are about to add a label $L$ to a bag $B_k(p)$, we check if $L$ is dominated by $B^*(p)$ or $B^*(p_i)$ (recall that $p_i$ is the target stop). If either is the case, $L$ is not added to $B_k(p)$. Otherwise, we also update $B^*(p)$ by adding $L$ to $B^*(p)$, if necessary. See the appendix for details on the implementation of bags.

Like RAPTOR, McRAPTOR scans routes in no particular order, and thus can be parallelized in the same way. However, since updates to $B_i(p)$ cannot be atomic, we must use the conflict graph approach.

Fare Zones. We now consider a practical scenario: fare zones. Transit agencies often assign each stop $p$ to one (or multiple) fare zones from a set $Z$. The price of a journey is then determined by which fare zones it touches. Since it is often not clear how to handle prices directly during the algorithm, it is simpler to keep track of fare zones instead. Thus, we are interested in computing all Pareto-optimal journeys including the set of touched fare zones as a criterion. Precise fare information can then be determined in a (quick) post-processing step.

We handle this scenario as follows. Each label is a tuple $L = (\tau(L), z(L))$, where $z(L) \subseteq Z$ is the set of touched fare zones so far. (Recall that number of transfers are not part of the label since they are handled implicitly by RAPTOR.) Here, a label $L_1$ dominates $L_2$ iff $\tau(L_1) \leq \tau(L_2)$ and $z(L_1) \subseteq z(L_2)$. Note that $z(p)$ is a cost imposed by stops rather than travel. We initialize the source bag $B_0(p_i)$ with a label $(\tau, z(p_i))$. Moreover, each time we are about to merge a label $L$ into a bag $B_k(p)$, we first update $z(L) \leftarrow z(L) \cup z(p)$. To implement $z(\cdot)$ efficiently, we use integers as bit sets (one bit per fare zone). Domination is tested by bitwise-and, and set-union is equivalent to bitwise-or.
Reliability. Another practical scenario we consider is the reliability of transfers. Following Disser et al. (2008), we consider the reliability of one transfer (i.e., change of trips) as a function of the buffer time of a transfer to the interval [0, 1]. Here, the buffer time is the time difference between departure and arrival of two subsequent trips $t_1 \neq t_2$ of a journey at some stop. It represents the maximum time $t_1$ may be delayed before $t_2$ is missed. The reliability of a transfer therefore represents the probability that the transfer will be made successfully. The reliability of a journey is the product over the reliabilities of all its transfers.

Our experiments consider two natural reliability functions. The first is a (piecewise) linear function $\text{rel}: \tau \mapsto \min(a \cdot \tau + b, 1)$, the second a discretized exponential function $\text{rel}: \tau \mapsto 1 - e^{\ln(1-a) - b/\tau}$ (proposed by Disser et al. (2008)). We may use $a$ and $b$ to set the reliability value for a buffer time of 0 as well as the buffer time for which the reliability reaches a “sufficiently high” value, e.g., 0.99. To limit the number of Pareto-optimal journeys, we may—like Disser et al. (2008)—further discretize the interval [0, 1] by subdividing it into a fixed number (e.g., 10) of equivalence classes of equal width.

We incorporate this criterion into McRAPTOR as follows. Each label is a tuple $L = (\tau(L), x(L))$, where $x(L) \in [0, 1]$ is the reliability value so far. A label $L_1$ dominates $L_2$ iff $\tau(L_1) \leq \tau(L_2)$ and $x(L_1) > x(L_2)$. The source bag $B_0(p_s)$ is initialized with a label $(\tau, 1)$. Moreover, we modify the third stage of every round (where we assign trips). Each label $(\tau(L), x(L)) \in B_{k-1}(p)$ may now result in several new labels (with assigned trips) in $B_k$: Taking a later trip (of $r$) in favor of a higher reliability may contribute to a Pareto-optimal solution. Hence, in a loop we create new labels $(\tau(L'), x(L'))$ for each trip $t$ that can be caught after $\tau(L)$ (ordered from earliest to latest). We set $x(L') = x(L) \cdot \text{rel}(\tau_{\text{dep}}(t, p) - \tau(L))$, possibly discretizing the value. We may stop the loop as soon as $\text{rel}(\tau_{\text{dep}}(t, p) - \tau(L))$ reaches 1 (which we ensure by discretizing $\text{rel}$ accordingly). Each newly created label $L'$ is then merged into $B_r$, thereby discarding dominated labels. Note that, to ensure that the first trip of a journey is always caught with probability 1, we do not create multiple labels in the first round of McRAPTOR. (We could create multiple labels if we wanted to represent the possibility of the user being delayed at the start of the journey.) Moreover, the algorithm is still correct if the reliability function is extended to also depend—in addition to the buffer times—on the particular stops where transfers are made.

Incorporating reliability into MLC requires modifications similar to McRAPTOR: One label may result in several new labels when evaluating edges of the graph. Additionally, one must use a weaker domination rule at route-vertices (cf. Section 2.1). Consider a vertex representing a route $r$. It may have two types of labels: transfer labels have the corresponding stop-vertex as their parent, while route labels have another route-vertex as parent. A transfer label can only dominate a route label iff their respective arrival times are more than the maximum buffer time (of the reliability function) apart. The reason for this is that the reliability for a transfer into route $r$ will only be considered when relaxing the next edge on route $r$.

4.2. Range Queries: rRAPTOR

As explained in Section 2, we can implement range queries using McRAPTOR by simply adding departure times as a criterion to the labels. In practice, however, we obtain a faster algorithm by extending RAPTOR using techniques originally proposed by Delling et al. (2012a) in the context of SPCS. In particular, it does not use costly bags. The resulting algorithm is called rRAPTOR.

Let $\Delta \subseteq \Pi$ be the input time range. First, we accumulate into a set $\Psi$ all departure times of trips $t$ at the source stop $p_s$ that depart within $\Delta$. We then run standard RAPTOR for every departure time $\tau \in \Psi$ independently. This results in a label $\tau_k(p)$ for every stop $p$, departure time $\tau$, and round $k$. However, not all journeys from $\Psi$ are useful to get to $p$. More precisely, a journey $J_1$ dominates a journey $J_2$ iff $\tau_{\text{dep}}(J_1) \geq \tau_{\text{dep}}(J_2)$ and $\tau_{\text{arr}}(J_1) \leq \tau_{\text{arr}}(J_2)$.

To integrate this domination rule, we order $\Psi$ from latest to earliest, and then run RAPTOR for every $\tau \in \Psi$ in order, but we keep the labels $\tau_k(p)$ between rounds instead of reinitializing them. To see why this is correct, note that $\tau_k(p)$ corresponds to an intermediate journey departing from $p_s$.
Table 1  Size figures for our input instances. The graph figures refer to the route model graph.

<table>
<thead>
<tr>
<th>Figure</th>
<th>London</th>
<th>Los Angeles</th>
<th>New York</th>
<th>Germany</th>
<th>Europe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stops</td>
<td>20843</td>
<td>15003</td>
<td>17894</td>
<td>6822</td>
<td>30517</td>
</tr>
<tr>
<td>Routes</td>
<td>2240</td>
<td>1099</td>
<td>1393</td>
<td>9348</td>
<td>44751</td>
</tr>
<tr>
<td>Trips</td>
<td>133011</td>
<td>16376</td>
<td>45299</td>
<td>104560</td>
<td>899485</td>
</tr>
<tr>
<td>Trips (Compressed)</td>
<td>52386</td>
<td>10554</td>
<td>20031</td>
<td>48377</td>
<td>443435</td>
</tr>
<tr>
<td>Foot Paths</td>
<td>45652</td>
<td>15482</td>
<td>49858</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Departure Events</td>
<td>5130905</td>
<td>931846</td>
<td>1825129</td>
<td>1019830</td>
<td>8341980</td>
</tr>
<tr>
<td>Min. Change Times</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Vertices (Graph)</td>
<td>101524</td>
<td>81657</td>
<td>66124</td>
<td>118365</td>
<td>550654</td>
</tr>
<tr>
<td>Edges (Graph)</td>
<td>285510</td>
<td>214369</td>
<td>193159</td>
<td>325292</td>
<td>1516012</td>
</tr>
</tbody>
</table>

no earlier than journeys computed in the current run (recall that Ψ is ordered). Thus, if \( \tau_k(p) \) is smaller, we also know how to reach \( p \) earlier. Hence, we can safely prune the current journey. However, we cannot use local pruning, since the best arrival times \( \tau^*(p) \) do not carry over to earlier departures. Instead, at the beginning of round \( k \) we set \( \tau_k(p) = \tau_{k-1}(p) \) for all stops where \( \tau_{k-1}(p) \) improves \( \tau_k(p) \).

RAPTOR’s parallelization techniques also work for rRAPTOR. However, since \( |\Psi| \) is usually larger than the number \( P \) of CPU cores, in practice we use the approach suggested by Delling et al. (2012a) for SPCS. We first partition \( \Psi \) into contiguous subsets \( \Psi_0, \ldots, \Psi_{P-1} \) of equal size, then each core \( i \) runs rRAPTOR on \( \Psi_i \) independently. The results are merged in the end, and dominated journeys are discarded.

5. Experiments

In this section we present an experimental study to evaluate our algorithms. Our main input uses realistic data from Transport for London. It includes tube (subway), buses, tram, Dockland Light Rail (DLR), and ferries. We extracted a Tuesday from the periodic summer schedule of 2011, which is publicly available from the London Data Store (2011). The network has 20843 stops, 2240 routes served by 133011 trips, and a total of 5130905 distinct departure events (a trip departing from a stop). Moreover, there are 45652 foot-paths in the network. When applying timetable compression (cf. Section 3.4), the number of trips is reduced to 52386 (a factor of 2.5). Each tube and DLR station is also assigned to one of 11 fare zones. In London a tube ticket automatically includes unlimited bus rides. Thus, we assign bus stops to a special fare zone that every tube/DLR station is also a member of. We compare our algorithms to existing graph-based techniques, which use the route model graph (cf. Section 2.1). The resulting graph has 101524 vertices and 285510 edges. These figures are also shown in Table 1, together with the corresponding sizes for the other instances we consider (which we discuss in Section 5.4).

All experiments were done on a dual 8-core Intel Xeon E5-2670 machine (16 cores total, hyper-threading deactivated) clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM. We implemented all algorithms in C++ (with OpenMP for parallelization), and compiled them with GCC 4.6.2 (64 bit) with full optimization. To evaluate performance, we ran 10000 queries with source/target stops and departure time selected uniformly at random. Results for more realistic distributions are similar.

5.1. RAPTOR

In our first set of experiments we evaluate RAPTOR (cf. Section 3) and compare it to LD and MLC (cf. Section 2.1), which solve the same problem (finding Pareto sets according to arrival times and number of transfers). With RAPTOR we separately evaluate the impact of (1) target pruning, and (2) the overhead of tracking paths to output full journey descriptions. Additionally to RAPTOR, we also evaluate FRAPTOR, which uses timetable compression (cf. Section 3.4). Note
that we always make use of marking and local pruning. All other algorithms are fully optimized: LD has pruning enabled, and MLC uses pruning, label-forwarding, and hop-avoidance. For comparison, we also report the performance of Time-Dijkstra (TD), which solves the (simpler) earliest arrival problem, which does not take the number of transfers into account. The results are presented in Table 2. We report the average number of visits and label comparisons per stop, the average size of the Pareto sets (number of journeys) output, and the average (single-core) running time in milliseconds. Moreover, for RAPTOR we report the average numbers of rounds, as well as the average number of times each route is processed.

We observe that, on average, RAPTOR (without target pruning) performs 9.8 rounds before it can stop (i.e., no labels can be improved) and scans each route 4 times. Recall that without target pruning, we compute optimal journeys to all stops in the network (which may be useful in some applications). Enabling target pruning reduces the number of rounds to 8.3 with 3 route scans on average. FRAPTOR on the compressed timetable is only 4% slower than RAPTOR, which is due to expanding trips on the fly. However, it keeps 2.5 times fewer trips (with their associated departure/arrival times) in memory. When considering the number of label comparisons per stop, we see that RAPTOR, MLC, and LD are no more than a factor of 2 apart. However, RAPTOR strongly benefits from its simpler data structures, better locality, and lack of a priority queue: With an average query time of 5.4 ms, it is 9 times faster than MLC, and 5 times faster than LD. Even TD, which only minimizes arrival time (regardless of the number of transfers), is outperformed by RAPTOR: It outputs half the number of journeys in twice the amount of time. Although TD could be accelerated using models yielding smaller graphs (as in Berger et al. (2009), Delling et al. (2012a), Geisberger (2010)), Berger et al. (2009) show that these models would make multi-criteria queries more complicated. When we are also interested in unpacking full journey descriptions, we observe that managing parent pointers within RAPTOR increases the running time by 22% (to 6.6 ms on average).

Local Queries. We also evaluate local queries with RAPTOR using the Dijkstra rank method, introduced by Sanders and Schultes (2005) in the context of road networks. In our scenario, we determine ranks by running Time-Dijkstra queries without stopping criterion from 10 000 source stops \( p_s \) (with random departure times). For each \( 2^i \)-th (for integral \( i \)) vertex extracted from the priority queue, we look up its corresponding stop \( p_t \), and create a \( p_s \rightarrow p_t \) query. These queries are then run with RAPTOR in random order. Figure 2 presents results using a box and whiskers plot. Besides RAPTOR, we also evaluate MLC and LD. We observe that MLC consistently performs worse than both RAPTOR and LD, due to its complicated handling of bags. Interestingly, LD is up to an order of magnitude faster than RAPTOR for very local queries (rank below \( 2^{11} \)). The reason for this is that RAPTOR must process routes in full length to ensure correctness (even when the

Table 2  Evaluation of different variants of RAPTOR and FRAPTOR on the London instance, compared to Time-Dijkstra (TD), Layered Dijkstra (LD), and Multi-Label-Correcting (MLC). Bullets (•) indicate different features: minimize arrival time (Arr), minimize number of transfers (Tran), target pruning (Prn), unpacking journeys (Unp).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Arr.</th>
<th>Tran.</th>
<th>Prn.</th>
<th>Unp.</th>
<th># Rnd.</th>
<th># Relax.</th>
<th># Visits</th>
<th># Comp.</th>
<th># Jn.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAPTOR</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>○</td>
<td>9.8</td>
<td>4.0</td>
<td>15.2</td>
<td>12.3</td>
<td>1.8</td>
<td>7.6</td>
</tr>
<tr>
<td>RAPTOR</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>8.3</td>
<td>3.0</td>
<td>10.6</td>
<td>10.9</td>
<td>1.8</td>
<td>5.4</td>
</tr>
<tr>
<td>RAPTOR</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>8.3</td>
<td>3.0</td>
<td>10.6</td>
<td>10.9</td>
<td>1.8</td>
<td>6.6</td>
</tr>
<tr>
<td>FRAPTOR</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>○</td>
<td>8.3</td>
<td>3.0</td>
<td>10.6</td>
<td>10.9</td>
<td>1.8</td>
<td>5.6</td>
</tr>
<tr>
<td>TD</td>
<td>○</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>2.6</td>
<td>7.4</td>
<td>0.9</td>
<td>11.0</td>
</tr>
<tr>
<td>LD</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>7.1</td>
<td>15.6</td>
<td>1.8</td>
<td>28.7</td>
</tr>
<tr>
<td>MLC</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>—</td>
<td>—</td>
<td>6.0</td>
<td>23.7</td>
<td>1.8</td>
<td>50.0</td>
</tr>
</tbody>
</table>
source and target stops are close). For higher ranks, RAPTOR outperforms LD (and MLC) by up to an order of magnitude.

### 5.2. Extensions of RAPTOR

Next, we evaluate McRAPTOR and rRAPTOR (cf. Section 4). For rRAPTOR, we fix the time range to 2 hours, and for McRAPTOR, we consider three variants. The first emulates a two-hour range query by using departure time as an additional criterion, the second uses fare zones, and the third uses reliability (as discussed in Section 4). Fare zones are implemented using bit sets. For reliability, we use the exponential function mentioned in Section 4.1, given by rel: \( \tau \mapsto 1 - e^{\ln(1-a) - b/\tau} \). We set \( a \) and \( b \) such that rel(0 min) = 0.5, and rel(10 min) = 0.99, subdividing the codomain of rel into 10 equivalence classes of equal width. Also, we store all relevant values of rel (in the range [0 min, 10 min]) into a lookup table, which accelerates the evaluation of rel during queries.

We compare our algorithms to two versions of MLC: one optimizes arrival time, transfers, and fare zones, and the other arrival time, transfers, and reliability. We also compare our algorithm to SPCS (with a 2-hour range); recall that SPCS is a range query minimizing only arrival time (regardless of transfers). The results are presented in Table 3. Note that columns Arr (arrival time), Rng (range), Tran (transfers), Fare (fare zones), Rel (reliability) indicate which criteria each method takes into account.

### Table 3 Comparing several extensions of RAPTOR on the London instance (see Section 4). We also include the Multi-Label-Correcting (MLC) and Self-Pruning Connection-Setting (SPCS) algorithms. Besides arrival time (Arr), the criteria we may consider are number of transfers (Tran), range (Rng), fare zones (Fare), and reliability (Rel).

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Arr.</th>
<th>Tran.</th>
<th>Rng.</th>
<th>Fare.</th>
<th>Rel.</th>
<th># Rnd.</th>
<th># Relax.</th>
<th># Visits</th>
<th># Comp.</th>
<th># Jn.</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>McRAPTOR</td>
<td>• •</td>
<td>• o</td>
<td>o o</td>
<td>—</td>
<td>—</td>
<td>9.4</td>
<td>3.8</td>
<td>14.1</td>
<td>1056.4</td>
<td>15.9</td>
<td>219.9</td>
</tr>
<tr>
<td>rRAPTOR</td>
<td>• •</td>
<td>• o</td>
<td>o o</td>
<td>—</td>
<td>—</td>
<td>139.0</td>
<td>36.5</td>
<td>119.0</td>
<td>110.2</td>
<td>15.9</td>
<td>61.3</td>
</tr>
<tr>
<td>SPCS</td>
<td>• o</td>
<td>o o</td>
<td>o o</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>31.7</td>
<td>87.1</td>
<td>7.4</td>
<td>177.1</td>
</tr>
<tr>
<td>McRAPTOR</td>
<td>• •</td>
<td>• o</td>
<td>o o</td>
<td>—</td>
<td>—</td>
<td>10.6</td>
<td>4.5</td>
<td>16.4</td>
<td>277.5</td>
<td>8.8</td>
<td>100.9</td>
</tr>
<tr>
<td>MLC</td>
<td>• o</td>
<td>o o</td>
<td>o o</td>
<td>—</td>
<td>—</td>
<td>8.4</td>
<td>3.1</td>
<td>11.1</td>
<td>89.6</td>
<td>4.7</td>
<td>71.9</td>
</tr>
<tr>
<td>McRAPTOR</td>
<td>• o</td>
<td>o o</td>
<td>o o</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>17.3</td>
<td>286.6</td>
<td>4.7</td>
<td>239.8</td>
</tr>
</tbody>
</table>

![Figure 2](image-url) Running time of MLC, LD, and RAPTOR on the London instance subject to the (Time-)Dijkstra Rank. Smaller ranks indicate more local queries.
Recall that rRAPTOR runs RAPTOR repeatedly (without reinitializing labels). In this experiment, it actually does so 20.7 times on average. Its performance reflects this: It runs 17 times as many rounds, and takes 61 ms on average. Using McRAPTOR to emulate the same range queries reduces the number of rounds (relative to rRAPTOR), but running times more than triple. Again, we profit from the simpler data structures. McRAPTOR handles bags of labels instead of running more rounds, which is costly. Compared to pure RAPTOR, taking London’s fare zones into account results in 4.9 times more reported journeys. Using McRAPTOR, we achieve a running time of 101 ms, a factor of 3 faster than MLC. Using reliability with McRAPTOR yields similar figures: We output 2.6 times the number of journeys (compared to RAPTOR) in 72 ms. Note that McRAPTOR’s speedup over MLC is less than the factor of 9 for RAPTOR (cf. Table 2); unlike RAPTOR, McRAPTOR also uses costly bags.

Number of Rounds. Figure 3 (left) shows the number of scanned routes per round for RAPTOR, rRAPTOR, and McRAPTOR. We normalize rRAPTOR’s plot by the number of calls to RAPTOR within each query, i.e., we report the average number of routes visited in each call to RAPTOR. All algorithms reach the entire network within about 5 rounds, when most routes are scanned. Beyond that, fewer routes are useful, and the algorithms begin running dry. McRAPTOR takes longer to converge, while rRAPTOR generally scans less routes (per departure time) than RAPTOR, since it can prune across different departure times.

Impact of Reliability. Figure 3 (right) presents the performance of McRAPTOR when varying the reliability function (with arrival times and number of transfers as other criteria). We compare the linear function to the exponential function (cf. Section 4.1). We fix rel(0 min) = 0.5, and vary the maximum buffer time \( \tau_m \) (for which rel(\( \tau_m \)) = 0.99 is reached) from 5 to 60 minutes. Again, we use 10 equivalence classes of equal width to subdivide the codomain of rel (cf. Section 4.1). We plot the average running time over 1 000 queries for each value of \( \tau_m \).

We see that the running time increases from around 60 ms (for \( \tau_m = 5 \text{ min} \)) to almost 250 ms (for \( \tau_m = 60 \text{ min} \)): Higher maximum buffer times yield more Pareto-optimal journeys, and therefore more work for McRAPTOR. The exponential function always has faster running times compared to the linear function. While the linear function distributes the buffer times (in the range from 0 to \( \tau_m \)) evenly among the equivalence classes, the exponential function maps values to equivalence classes of high reliability earlier, thus reducing the number of Pareto-optimal solutions.

5.3. Parallelization

Table 4 shows the parallel performance of our algorithms. Since writes to the labels \( \tau_k(p) \) are atomic for RAPTOR, we use update logs; McRAPTOR is parallelized using conflict graphs. Among the
Dijkstra-based algorithms, only SPCS can be parallelized efficiently (across departure times). We ran each algorithm on one, four, eight, and 16 cores, pinning thread \( i \) to core \( i \). Note that by this configuration, we utilize only one of the two CPUs in our machine for up to eight cores.

Comparing the single-core execution of the parallel implementations (Table 4) with the sequential ones (Tables 2 and 3), we observe a slowdown of less than 10% for all algorithms. Some slowdown is expected because we introduce additional work for our parallel implementations (see Section 3.3). On eight cores, RAPTOR achieves a speedup of only 2.1. Recall that we only parallelize scanning routes, which limits the speedup due to Amdahl's Law (see Amdahl (1967)). Because McRAPTOR spends more time on each route (due to the costly processing of bags), it benefits more from parallelization (a factor of up to 5 with fare zones or range query emulation). Finally, rRAPTOR achieves a speedup of 3.7 on eight cores, which is consistent with SPCS. Using 16 cores hardly pays off. Compared to eight cores, for most of the algorithm the performance even gets slightly worse again. Increased memory contention is a factor in this case.

**Impact of Range.** Figure 4 evaluates our parallel implementation of rRAPTOR for varying range values: We fix the departure time to 6:00 am, but vary the range from 1 to 16 hours. For each range value we run 1000 queries between (the same) random pairs of stops. We report the average running time on one, two, four, eight, and 16 cores, using the same configuration as above.

We observe that increasing the range results in higher running times, with (almost) linear correlation. Note that journeys departing late (at night) might not always reach the target stop due to London’s hours of operation. This may in particular occur for high range values, which explains the slight sublinear growth rate for ranges greater than 12 hours. As Figure 4 shows, parallelizing

![Figure 4](image-url)
Table 5 Comparison of base RAPTOR, rRAPTOR, LD, MLC, and SPCS on other instances. A trailing "8" in the algorithm description refers to a parallel execution on eight cores.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Arr.</th>
<th>Tran.</th>
<th>Rng.</th>
<th>Rel.</th>
<th># Comp. Time [ms]</th>
<th>Los Angeles</th>
<th>New York</th>
<th>Germany</th>
<th>Europe</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAPTOR</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>11.5</td>
<td>2.5</td>
<td>7.1</td>
<td>2.3</td>
<td>36.5</td>
</tr>
<tr>
<td>RAPTOR-8</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>11.6</td>
<td>1.4</td>
<td>7.2</td>
<td>1.5</td>
<td>37.0</td>
</tr>
<tr>
<td>TD</td>
<td>•</td>
<td>○</td>
<td>○</td>
<td>◦</td>
<td>7.3</td>
<td>5.2</td>
<td>4.8</td>
<td>4.9</td>
<td>26.2</td>
</tr>
<tr>
<td>LD</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>13.5</td>
<td>14.8</td>
<td>9.6</td>
<td>13.3</td>
<td>55.5</td>
</tr>
<tr>
<td>MLC</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>16.6</td>
<td>27.9</td>
<td>13.5</td>
<td>22.0</td>
<td>82.4</td>
</tr>
<tr>
<td>rRAPTOR</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>46.1</td>
<td>12.7</td>
<td>41.1</td>
<td>16.6</td>
<td>68.6</td>
</tr>
<tr>
<td>rRAPTOR-8</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>56.2</td>
<td>9.3</td>
<td>51.2</td>
<td>7.8</td>
<td>167.4</td>
</tr>
<tr>
<td>SPCS</td>
<td>•</td>
<td>○</td>
<td>○</td>
<td>◦</td>
<td>35.9</td>
<td>39.7</td>
<td>33.3</td>
<td>48.0</td>
<td>69.5</td>
</tr>
<tr>
<td>SPCS-8</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>◦</td>
<td>52.7</td>
<td>18.8</td>
<td>54.4</td>
<td>17.7</td>
<td>185.5</td>
</tr>
<tr>
<td>McRAPTOR</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>◦</td>
<td>36.6</td>
<td>22.9</td>
<td>43.2</td>
<td>29.4</td>
<td>81.3</td>
</tr>
<tr>
<td>MLC</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>◦</td>
<td>73.4</td>
<td>69.0</td>
<td>115.4</td>
<td>82.3</td>
<td>180.7</td>
</tr>
</tbody>
</table>

rRAPTOR pays off in all cases: For a range of 16 hours we achieve speedup factors of 5.1 on eight, and 9.4 on 16 cores over the sequential algorithm. Using eight cores, we can compute multi-criteria range queries in less than 0.15 sec in all cases.

5.4. Additional Inputs

We now consider four more test inputs: Los Angeles and New York, which are also metropolitan networks, and two railway networks, Germany and Europe. We generated the local networks from publicly available feeds using the General Transit Feed Specification (2010). The railway data was kindly provided to us by HaCon. Los Angeles and New York contain subways and buses, and in both cases we use an extract of August 10, 2011 (a Wednesday) to create the timetable. The German network is based on the winter schedule of 2001/2002 and contains all trains operated by Deutsche Bahn. The European network is based on the winter schedule of 1996/1997 and contains mostly long-distance trains. Figures for the sizes of all networks are summarized in Table 1. Note that while Los Angeles and New York are the biggest publicly available GTFS networks at the time of writing, they are both smaller than the London instance. Moreover, for the metropolitan networks foot-path data was not available to us. Hence, we generated foot-paths on these instances with a heuristic proposed by Delling et al. (2012a). It creates cliques of foot-paths between stops that are close to the same intersection of the road network. For Germany and Europe we use minimum change times to model transfers within railway stations (cf. Section 2). Since no fare zone data is readily available for any of the networks, we did not run our multi-criteria algorithms that include fare zones. Table 5 shows the results for all relevant algorithms.

The results are consistent with the previous experiments: RAPTOR outperforms both LD and MLC on every instance. It can compute all Pareto-optimal journeys between two random stops within 2.5 ms on Los Angeles, 2.3 ms on New York, and 5.4 ms on Germany. On Europe, RAPTOR takes 42.2 ms, which is due to the very high number of routes in this instance (44751 compared to, e.g., 2240 on London, cf. Table 1). Parallelizing RAPTOR shows moderate effect: Speedups are below a factor of 2.6 for eight cores on all instances. McRAPTOR (with reliability) computes journeys in under 30 ms on all instances except Europe, where it takes 189 ms. Running rRAPTOR results in query times below 17 ms for all instances except Europe, where it is 59 ms. Parallelizing rRAPTOR only pays off on the dense urban networks, but speedups are still limited. On the two railway networks parallel rRAPTOR does not scale. Here, trips operate too infrequently, thus,
Figure 5 Illustration of the adjacency structure of routes.

too few of them can be distributed among the cores. Finally, we observe that rRAPTOR again outperforms SPCS by a factor of up to 3.2.

6. Conclusion
We have introduced RAPTOR, a novel algorithm for fast multi-criteria journey planning in public transit networks. Unlike previous algorithms, it neither operates on a graph nor requires a priority queue. Instead, it exploits the inherent structure of such networks by operating in rounds and processing each route of the network at most once per round. Moreover we extend it to range queries, and additional criteria (such as fare zones and reliability of transfers) can be added. Experiments on the transit network of London reveal that RAPTOR is more than an order of magnitude faster than previous approaches. RAPTOR can be easily parallelized, which accelerates queries even further. Finally, since RAPTOR does not rely on preprocessing, it can be directly used in dynamic scenarios, easily handling delays and cancellations.

Regarding future work, we are interested in using RAPTOR to handle public transport networks of continental size. For such networks, however, we most likely have to apply some preprocessing. Moreover, we are interested to incorporate uncertainty into RAPTOR beyond considering reliability. For example, a journey plan that also contains backups for the case that transfers are missed could be obtained by minimizing expected arrival time.

Acknowledgments
We would like to thank Dominic Green, Hatay Tuna, Kutay Tuna, and Simon Williams for inspirational discussions and processing the London transit data.

Appendix. Implementation Details
In this appendix, we present details on the data structures we use for RAPTOR. For simplicity, we assume all routes, trips, and stops have sequential integral identifiers, each starting at 0.

Route Traversal
For the main loop of the algorithm, we need to traverse routes. For route $r$, we need its sequence of stops (in order), as well as the list of all trips (from earliest to latest) that operate on that route.

To accomplish this, we store an array $Routes$ where the $i$-th entry holds information about route $r_i$. It stores the number of trips associated with $r_i$, as well as the number of stops in the route (which is the same for all its trips). It also stores pointers to two lists.

The first pointer in $Routes[i]$ is to a list representing the sequence of stops along route $r_i$. Instead of representing each list of stops separately (one for each route), we group them into a single array $RouteStops$. Its first entries are the sequence of stops of route 0, then those for route 1, and so on. The pointer in $Routes[i]$ is to the first entry in $RouteStops$ that refers to route $i$. See Figure 5.

The second pointer in $Routes[i]$ points to a representation of the list of trips that operate on that route. Once again, instead of keeping separate lists for different routes, we keep a single array $StopTimes$. (See Figure 5.) This array is divided into blocks, and the $i$-th block contains all trips corresponding to route $r_i$. 
Within a block, trips are sorted by departure time (at the first stop). Each trip is just a sequence of stop times, represented by the corresponding arrival and departure times.

A route \( r_i \) can be processed by traversing the stops in \( \text{RouteStops} \) associated with \( r_i \). To find the earliest trip departing from some stop \( p \) along the route after some time \( \tau \), we can quickly access the stop times of all trips at \( r_i \) at \( p \) in constant time per trip due to the way we sorted \( \text{StopTimes} \). In particular, when processing \( r_i \) with trip \( t \), the arrival time of the next stop is determined by the subsequent entry in \( \text{StopTimes} \). Furthermore, to check for an earlier valid trip of \( r_i \), we can jump \( |r_i| \) (the length of the route \( r_i \), which is stored in \( \text{Routes} \) entries) to the left to retrieve the departure time of the next earlier trip. Note that these data structures are also suited for computing latest departure queries (cf. Section 2).

**Timetable Compression**

To enable timetable compression (cf. Section 3.4) in our data structure, we store the periodicity and size of the respective trip group with each stop time (Recall that each stop time is part of a unique trip). For each trip group, only the first (earliest) trip is represented in \( \text{StopTimes} \). All subsequent trips (of that trip group) are stored implicitly and are, thus, discarded. Since every stop time knows about the periodicity and size of its trip group, the uncompressed stop times can be easily reconstructed on-the-fly. Note that the data structure could be compressed even further: Instead of storing the periodicity and the size of a trip group at each stop time (which is redundant), we could store them only once in a separate array indexed by trip id.

**Other Operations**

We still need to support some operations outside the main loop of the algorithm. For those, we need an array \( \text{Stops} \), which contains information about each individual stop. In particular, for each stop \( p_i \), we must know the list of all routes that serve it in order to mark the appropriate routes between rounds. Moreover, we also need the list of all foot-paths that can be taken out of \( p_i \), together with their corresponding lengths.

As before, we aggregate these two sets of lists in two arrays. \( \text{StopRoutes} \) contains the lists of routes associated with each stop: first the routes associated with \( p_0 \), than those associated with \( p_1 \), and so on. Similarly, \( \text{Transfers} \) represents the allowed foot-paths from \( p_0 \), followed by the allowed foot-paths from \( p_1 \), and so on. (Each individual foot-path from \( p_i \) is represented by its target stop \( p_j \) together with the transfer time \( \ell(p_i, p_j) \).) The \( i \)-th entry in \( \text{Stops} \) points to the first entries in \( \text{StopRoutes} \) and \( \text{Transfers} \) associated with stop \( p_i \). See Figure 6.

**Bags**

Some of our algorithms, such as McRAPTOR, use bags to represent sets of nondominated labels. Each bag \( B_i \) is represented by a dynamic (unordered) array which contains its labels. Whenever we merge another bag \( B_j \) into \( B_i \), we check domination between each pair of labels \( L_i \in B_i \) and \( L_j \in B_j \), adding labels to \( B_i \) accordingly. Dominated labels in \( B_j \) are marked and erased from the array at the end of the merge operation.

**References**


