Round-Based Public Transit Routing

Daniel Delling* Thomas Pajor† Renato F. Werneck‡

Abstract

We study the problem of computing all Pareto-optimal journeys in a dynamic public transit network for two criteria: 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. Moreover, it can be easily extended to handle flexible departure times or arbitrary additional criteria, such as fare zones. When run on London's complex public transportation network, RAPTOR computes all Pareto-optimal journeys between two random locations an order of magnitude faster than previous approaches, which easily enables interactive applications.

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 [25]. At first glance, this is tempting: one can just use Dijkstra's algorithm [14], 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. [13] for an overview). Unfortunately, there are several downsides to this approach. Although known speedup techniques can achieve speedups of up to several million on road networks [1], they fall short when applied to public transportation networks [2], which have a much different structure. More importantly, unlike in road networks, travel times are usually not enough to compute best journeys in public transit; other criteria, such as number of transfers and costs, can be just as important. This is often handled by reporting all *Pareto-optimal* journeys between two points using augmented versions of Dijkstra's algorithm [25]. This increases running times significantly and makes acceleration techniques even more complicated [5, 12, 15, 23, 24], rendering the efficient computation of multi-criteria journeys an elusive goal [5]. 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 [3]: it can answer queries extremely 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 work 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 [20, 22], with RAPTOR we can 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. The second extension is rRAPTOR: it computes bicriteria *range queries*, outputting 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. Finally, our experiments show that on the full network of London, with over 20 thousand

*Microsoft Research Silicon Valley. dadellin@mircosoft.com

†Karlsruhe Institute of Technology. pajor@kit.edu. This work was done while the author was at Microsoft Research Silicon Valley.

‡Microsoft Research Silicon Valley. renatow@microsoft.com

stops and 5 million daily departure events, RAPTOR computes all Pareto-optimal bicriteria journeys between two random locations in under 8 ms—fast enough for practical use.

The paper 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* $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$ where $\Pi \subset \mathbb{N}_0$ is the *period of operation* (think of it as the seconds of a day), $\mathcal{S}$ is a set of *stops*, $\mathcal{T}$ a set of *trips*, $\mathcal{R}$ a set of *routes*, and $\mathcal{F}$ a set of *transfers* (or *foot-paths*). Each stop in $\mathcal{S}$ 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 $t \in \mathcal{T}$ 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 an undefined arrival and departure time, respectively. The trips in $\mathcal{T}$ are partitioned into routes: each route in $\mathcal{R}$ consists of the trips that share the same sequence of stops. Typically, there are many more trips than routes. Finally, foot-paths in $\mathcal{F}$ model walking connections (or transfers) between stops. Each transfer consists of two stops p_1 and p_2 with an associated constant walking time $\ell(p_1, p_2)$. Note that $\mathcal{F}$ is transitive: if p_1 and p_2 are indirectly connected by foot-paths, (p_1, p_2) is contained in $\mathcal{F}$ as well.

The output produced by any journey-planning algorithm on a timetable is a set of *journeys* $\mathcal{J}$. 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 $J_1 \preccurlyeq J_2$, if J_1 is no worse in any criterion than J_2. A set of pairwise non-dominating journeys is a *Pareto-set*. In our algorithms we use *labels* (often associated with stops) for intermediate journeys. The definition of domination translates to labels naturally.

The simplest problem we consider is the *Earliest Arrival Problem*. Given a source stop p_s, a target stop p_t, and a departure time τ, it asks for a journey that departs p_s no earlier than τ, and arrives at p_t as early as possible. The *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 τ. 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 *Range Problem* asks for alternate journeys with varying departure time. 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 τ 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 $J_1 \preccurlyeq J_2$ iff $\tau_{\text{dep}}(J_1) \geq \tau_{\text{dep}}(J_2)$ and $\tau_{\text{arr}}(J_1) \leq \tau_{\text{arr}}(J_2)$.

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 [25]. The *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 *time-dependent* model groups trips along edges into time-dependent functions. In general, the size of the resulting graph is linear in the number of stops and routes, which is orders of magnitude smaller than the number of events. We therefore focus on the more efficient time-dependent model.

The time-dependent *route model* [25] creates a stop-vertex for each stop $p \in \mathcal{S}$. In addition, a route-vertex r_p is created for each stop p and every route $r \in \mathcal{R}$, where r is serving 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 $t \in \mathcal{T}$ serves two subsequent stops p_1, p_2 along its route $r \in \mathcal{R}$, an edge from r_{p_1} to r_{p_2} is required. This edge is time-dependent, and its function reflects a travel time of $\tau_{\text{arr}}(t, p_2) - \tau_{\text{dep}}(t, p_1)$ at departure time $\tau_{\text{dep}}(t, p_1)$. Edge costs can be modeled as special piecewise linear functions, and can be efficiently evaluated [9, 10]. To incorporate foot-paths, for each $(p_i, p_j) \in \mathcal{F}$ a time-independent edge is added between the corresponding stop-vertices, weighted by $\ell(p_i, p_j)$. The time-dependent *station model* is a condensed version with only one vertex per stop, combined with time-dependent edges. Although it leads to smaller graphs, it complicates the query algorithms in order to incorporate transfers correctly [5, 16].

Algorithms. To solve the earliest arrival problem from a source stop p_s at time τ on the time-dependent model, we can use an augmented variant of *Dijkstra's algorithm* [14]. It scans vertices with increasing arrival time, but evaluates each edge $e = (u, v)$ at time $\text{dist}(u)$, which is the 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) [25]. 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_u 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. Several improvements to MLC exist [15]: (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 helpful for long-distance rail networks [15], it does not help on dense urban networks [4]. Even with these improvements, MLC is much more costly than plain Time-Dijsktra: 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, one can use the simpler *Layered Dijkstra* (LD) algorithm [7]. It works when values of the second criterion 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 smaller than k. Similarly, the label can be pruned if the target vertex has a label with smaller arrival time on any layer $\leq k$. 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) [10]. It first assembles all departing trips at p_s. Then, it initializes a priority queue with all these trips, using their 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 $\tau_{\text{dep}}(L') \geq \tau_{\text{dep}}(L)$ 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.

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. We start with a basic version of the algorithm, then propose some optimizations. Let $p_s \in \mathcal{S}$ be the source stop, and $\tau \in \Pi$ the departure time. Recall that our goal is to compute for every k a nondominated journey to a target stop p_t with minimum arrival time having at most k trips.

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. (We show that this bound can be dropped later.) 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 ∞. We then set $\tau_0(p_s) = \tau$. 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 ∞. 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

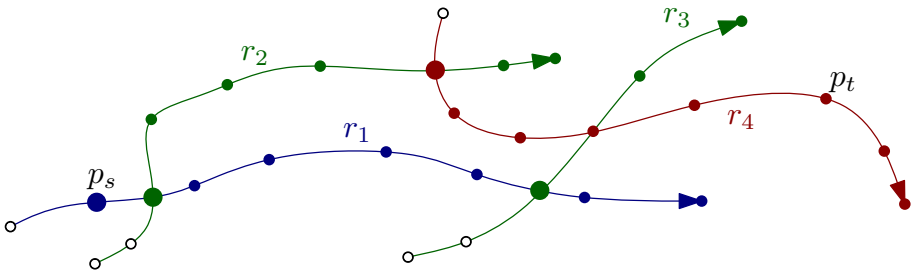

Figure 1: Scanning routes for a query from p_s to p_t. Route r_1 is first scanned in round 1, routes r_2 and r_3 in round 2, and finally, route r_4 in round 3. Scanning a route begins at the earliest marked stop (bold). Hollow stops are never visited.

timetable exactly once. Consider a route r, and let $\mathcal{T}(r) = (t_0, t_1, \ldots, t_{|\mathcal{T}(r)|-1})$ be the sequence of trips that follow route r, from earliest to latest. When processing route r, we consider journeys where the last (k'th) trip taken is in route r. Let $\text{et}(r, p_i)$ be the earliest trip in route r that one can catch at stop p_i, i. e., the earliest trip t such that $\tau_{\text{dep}}(t, p_i) \geq \tau_{k-1}(p_i)$. (Note that this trip may not exist, in which case $\text{et}(r, p_i)$ is undefined.) To process the route, we visit its stops in order until we find a stop p_i such that $\text{et}(r, p_i)$ 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 $\tau_k(p_j)$ 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 along r 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 $\tau_{k-1}(p_i) < \tau_{\text{arr}}(t, p_i)$ and update t by recomputing $\text{et}(r, p_i)$.

Finally, the third stage of round k considers foot-paths. For each foot-path $(p_i, p_j) \in \mathcal{F}$ it sets $\tau_k(p_j) = \min\{\tau_k(p_j), \tau_k(p_i) + \ell(p_i, p_j)\}$. Note that since $\mathcal{F}$ 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. Also note that we can dynamically extend the multilabels beyond K items.

The worst-case running time of our algorithm can be bounded as follows. In every round, we scan each route $r \in \mathcal{R}$ at most once. If $|r|$ is the number of stops along r, then we look at $\sum_{r \in \mathcal{R}} |r|$ stops in total. Caching $\text{et}(r, \cdot)$, we look at every trip t of a route at most once, since $\text{et}(r, \cdot)$ can only decrease while scanning a route. Thus, the total running time of our algorithm is *linear* per round. In total, it takes $\mathcal{O}\big(K(\sum_{r \in \mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|)\big)$ time, where K is the number of rounds. Constant access to the stops along routes and the arrival and departure times of specific trips can be achieved by a few arrays (see Appendix A 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 improved version of the algorithm, we *mark* during round $k-1$ those stops p_i for which we improved the arrival time $\tau_{k-1}(p_i)$. 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 $\tau^*(p_i)$ representing the earliest known arrival time at p_i. Since we are only interested in Pareto-optimal paths, we only mark a stop during route traversal at round k when the arrival time with k trips is earlier than $\tau^*(p_i)$. Note that local pruning allows us to drop the first stage (copying the labels from the previous round), since $\tau^*(p_i)$ automatically keeps track of the earliest possible time to get to p_i.

Note that, as described, RAPTOR does not exploit the fact that we are only interested in journeys to a target stop p_t. In fact, it computes journeys to *all* stops of the network. Since this seems wasteful, we use *target pruning*. During round k, there is no need to mark stops whose arrival times are greater than $\tau^*(p_t)$ (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. MLC can be extended to the scenario where one is interested in Pareto-optimal solutions with respect to *strict domination*: one journey only dominates another if it is strictly better in at least one criterion [6]. This leads to bigger Pareto-Sets. The motivation for this extension stems from outputting journeys which use transfers at preferred locations. Then, the best journey is 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. If strict dominance is still necessary, it can be incorporated into our algorithm as well. 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 $\tau_{k-1}(p') \leq \tau_{\text{dep}}(t, p')$ held. 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.

3.3 Parallelization. Our algorithm can be extended to work in parallel. Most of the work is spent processing individual routes, which are scanned 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

Algorithm 1: RAPTOR

Input: Source and target stops p_s, p_t and departure time τ.

```
// Initialization of the algorithm
foreach i do
    τ_i(·) ← ∞
τ*(·) ← ∞
τ_0(p_s) ← τ
mark p_s
foreach k ← 1, 2, ... do
    // Accumulate routes serving marked
       stops from previous round
    Clear Q
    foreach marked stop p do
        foreach route r serving p do
            if (r, p') ∈ Q for some stop p' then
                Substitute (r, p') by (r, p) in Q if p
                comes before p' in r
            else
                Add (r, p) to Q
        unmark p
    // Traverse each route
    foreach route (r, p) ∈ Q do
        t ← ⊥ // the current trip
        foreach stop p_i of r beginning with p do
            // Can the label be improved in
               this round? Includes local
               and target pruning
            if t ≠ ⊥ and
            arr(t, p_i) < min{τ*(p_i), τ*(p_t)} then
                τ_k(p_i) ← τ_arr(t, p_i)
                τ*(p_i) ← τ_arr(t, p_i)
                mark p_i
            // Can we catch an earlier trip
               at p_i?
            if τ_{k-1}(p_i) ≤ τ_dep(t, p_i) then
                t ← et(r, p_i)
    // Look at foot-paths
    foreach marked stop p do
        foreach foot-path (p, p') ∈ F do
            τ_k(p') ← min{τ_k(p'), τ_k(p) + ℓ(p, p')}
            mark p'
    // Stopping criterion
    if no stops are marked then
        stop
```

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 all update attempts on any value $\tau_k(p)$. The logs are then used to correct the labels by the master thread sequentially. The same technique can also be used to keep $\tau^*(p)$ 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 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 $\mathcal{R} = \{r_0, r_1, \ldots, r_{|\mathcal{R}|-1}\}$. 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 thread finishes scanning a route, it grabs 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. 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.

4 Extensions

In this section we show how RAPTOR can be extended to handle additional criteria, such as fare zones. We call the resulting algorithm McRAPTOR. For the special case of bicriteria range queries, we present a tailored extension called rRAPTOR.

4.1 More Criteria: McRAPTOR. Recall that plain RAPTOR stores exactly one value $\tau_k(p)$ per stop and round. To extend the algorithm to more criteria, we keep multiple nondominating 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 relaxing a route r, we first create an empty *route bag* B_r. 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 $L \in B_r$ according to 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. Moreover, 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 can also adapt local and target pruning. Similarly to τ^* in RAPTOR, we keep for every stop p a *best bag* $B^*(p)$ that—informally—represents the non-dominated 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 $B^*(p_t)$ (recall that p_t 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.

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

An example: 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 $\mathcal{Z}$. The price of a journey is then determined by which fare zones it touches. However, handling exact prices during the algorithm often is complicated. 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 $z(L) \subseteq \mathcal{Z}$ is the set of touched fare zones so far. 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 $B_0(p_s)$ with a label $(\tau, z(p_s))$. 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.

4.2 Range Queries: rRAPTOR. For the special case of range queries, RAPTOR can be extended to

rRAPTOR similarly to SPCS [10].

Let $\Delta \subseteq \Pi$ be the input time range. First, we accumulate into a set Ψ all departure times of trips t at the source stop p_s that depart within Δ. Now, we 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 τ, and round k. However, not all journeys from Ψ 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 Ψ from latest to earliest, and then run RAPTOR for every $\tau \in \Psi$ in order. However, we keep the labels $\tau_k(p)$ between rounds instead of reinitializing them. To see why this is correct, note that this value of $\tau_k(p)$ corresponds to an intermediate journey departing from p_s *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, when $|\Psi|$ is larger than the number P of CPU cores, we can use the same approach as for SPCS [10]: We partition Ψ into subsets of equal size $\Psi_0, \ldots, \Psi_{P-1}$. Then, each core i runs rRAPTOR on Ψ_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 [26]. It includes tube (subway), buses, tram, and Dockland Light Rail (DLR). We extracted a Tuesday from the periodic summer schedule of 2011, which is publicly available [18]. The network has 20 843 stops, 2 225 routes served by 133 011 trips, and a total of 5 132 672 distinct departures (a trip departing at a stop). Moreover, there are 45 652 footpaths in the network. Each tube and DLR station is also assigned to one of 11 fare zones. In London a tube ticket automatically includes any bus ride. Thus, we assign bus stops to a special fare zone z 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 100 878 vertices and 283 587 edges.

All experiments were done on a dual 6-core Intel Xeon X5680 machine clocked at 3.33 GHz, with 96 GiB of DDR3-1333 RAM. We implemented all algorithms in C++ (with OpenMP for parallelization), and compiled them with Microsoft Visual C++ 2010 (64 bit) with full optimization. To evaluate performance, we ran 10 000 queries with source/target stops and departure time selected uniformly at random. Results for more realistic distributions are similar.

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. All algorithms are fully optimized: RAPTOR makes use of marking, local, and target pruning; 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. The results are presented in Table 1. We report the average number of visits and label comparisons per stop, the size of the Pareto-sets (number of journeys) output, and the 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 relaxed.

We observe that, on average, RAPTOR performs 8.4 rounds before it can stop (i. e., no labels can be improved) and scans each route 3 times. 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

Table 1: Evaluation of the base variant of RAPTOR on the London instance, compared to Time-Dijkstra (TD), Layered Dijkstra (LD), and Multi-Label-Correcting (MLC). "Tr" indicates whether the algorithm is (•) minimizing the number of transfers besides arrival time or not (◦).

Algorithm	Tr	# Rnd.	# Relax. p. Route	# Visits p. Stop	# Comp. p. Stop	# Jn.	Time [ms]
RAPTOR	•	8.4	3.0	11.1	22.2	1.9	7.3
TD	◦	—	—	7.4	7.4	0.9	14.2
LD [7]	•	—	—	17.3	39.5	1.9	44.5
MLC [15]	•	—	—	12.8	28.7	1.9	67.2

Table 2: Comparison of 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, the criteria we may consider are range (R), number of transfers (Tr), and fee zones (Fz).

Algorithm	R	Tr	Fz	# Rnd.	# Relax. p. Route	# Visits p. Stop	# Comp. p. Stop	# Jn.	Time [ms]
rRAPTOR	•	•	○	138.5	36.6	124.7	346.4	16.3	87.0
McRAPTOR	•	•	○	9.5	3.8	15.1	2062.7	16.3	259.8
McRAPTOR	○	•	•	10.8	4.5	17.9	396.4	9.0	107.4
MLC [15]	○	•	•	—	—	48.1	930.3	9.0	399.5
SPCS [11]	•	○	○	—	—	76.2	76.2	7.8	183.6

benefits from its simpler data structures, better locality, and lack of a priority queue: with an average query time of 7.3 ms, it is 9 times faster than MLC, and 6 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 [5, 11, 16], these models would make multi-criteria queries more complicated [5].

Extensions of RAPTOR. We evaluate McRAPTOR and rRAPTOR (cf. Section 4). For rRAPTOR, we fix the time range to 2 hours, and consider two variants of McRAPTOR. The first emulates a two-hour range query by using departure time as an additional criterion; the second uses fare zones, as discussed in Section 4. We compare our algorithms to MLC (using arrival time, transfers, and fare zones) and SPCS (using a range of 2 hours as well). Note that SPCS is a range query minimizing only arrival time (regardless of transfers). The results are presented in Table 2. Note that columns R (range), Tr (transfers), and Fz (fare zones) indicate which criteria each method takes into account.

Recall that rRAPTOR repeatedly runs RAPTOR (without reinitializing labels). Its performance reflects this: it runs 16 times as many rounds, and takes 87 ms on average. Using McRAPTOR to emulate the same range queries reduces the number of rounds (relative to rRAPTOR), but running times 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 4.7 times more reported journeys. Using McRAPTOR, we achieve a running time of 107 ms, a factor of 3.7 faster than MLC. This is less than the factor of 9 for RAPTOR (cf. Table 1): unlike RAPTOR, McRAPTOR also uses costly bags.

Figure 2 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. 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.

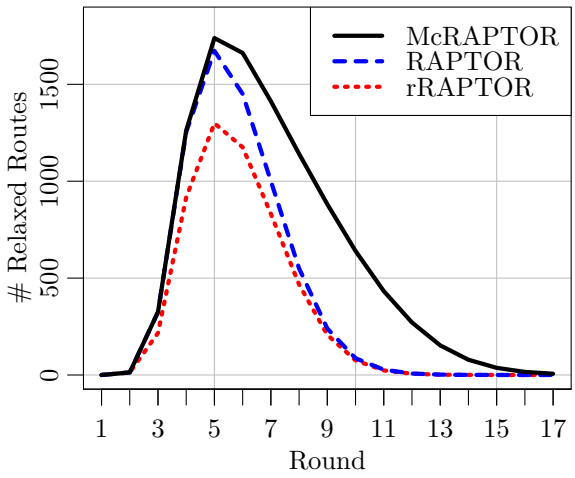

Figure 2: Number of relaxed routes per round.

Parallelization. Table 3 shows the parallel performance of our algorithms. Since writes to $\tau_k(p)$ 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. We ran each algorithm on one, three, six, and 12 cores, pinning thread i to core i.

Comparing the parallel implementations (Table 3) with the sequential ones (Tables 1 and 2), we observe a slow-down of less than 10% for all algorithms. This is expected because we introduce additional work for our parallel implementations (see Section 3.3). On six cores, RAPTOR achieves a speedup of only 1.9. Recall that

Table 3: Parallel performance of RAPTOR, McRAPTOR, rRAPTOR, and SPCS in a multi-core setup.

				1 core		**3 cores**		**6 cores**		**12 cores**	
Algorithm	R	Tr	Fz	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]
RAPTOR	○	●	○	21.5	7.7	21.7	5.0	21.8	4.1	21.8	3.7
rRAPTOR	●	●	○	346.4	92.3	357.7	39.5	374.0	26.8	404.6	21.6
McRAPTOR	●	●	○	2098.6	280.2	2101.2	113.1	2098.4	66.1	2098.2	50.1
McRAPTOR	○	●	●	410.0	118.6	410.6	49.4	408.4	29.9	408.3	26.1
SPCS [11]	●	○	○	76.2	183.6	79.9	69.1	85.2	44.9	95.5	38.9

we only parallelize scanning routes, which limits the speedup due to Amdahl's law (it makes up 75 % of the total sequential running time). Because McRAPTOR spends more time on each route (due to the costly manipulation of bags), it has better speedups when run in parallel (a factor of 4 with either fare zones or range query emulation). Finally, rRAPTOR achieves a speedup of 3.4 on six cores, which is consistent with SPCS. Using 12 cores hardly pays off for any algorithm. Compared to six cores, the additional speedup is limited; increased memory contention is a factor in this case.

Additional inputs. We now consider three more test cases: Los Angeles [19], New York [21], and Chicago [8]. We generated these instances from General Transit Feeds (GTFS) that are publicly available [17]. We use an extract of August 10, 2011 (a Wednesday). The timetables consist of 15 003/17 894/12 137 stops, 1 099/1 393/710 routes followed by 16 376/45 299/20 303 trips, and 931 864/1 825 221/1 194 571 departure events. When building the route model graphs, the resulting instances have 81 657/66 124/47 561 vertices and 214 369/193 159/118 452 edges. Unfortunately, footpath data was not available with these networks, so we generated synthetic foot-paths with a known heuristic [11], resulting in 15 482/49 858/12 888 inserted footpaths. Since no fare zone data is available for these networks, we did not run our multi-criteria algorithms that include fare zones. Note that while these are the biggest publicly available GTFS networks at the time of writing, they are all smaller than the London instance. Table 4 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 3.4 ms on Los Angeles, 3.1 ms on New York, and 1.8 ms on Chicago. Parallelizing RAPTOR shows almost no effect: speedups are below a factor of 1.7 for 6 cores on all instances. Running rRAPTOR results in query times of around 20 ms for all instances. Parallelizing rRAPTOR pays off more than for RAPTOR, though speedups are still limited. The best speedup of 2.7 on 6 cores is achieved on New York. Finally, we observe that rRAPTOR again outperforms SPCS by a factor of 2.

Table 4: Comparison of base RAPTOR, rRAPTOR, LD, MLC, and SPCS on other instances. A trailing "6" in the algorithm description refers to a parallel execution on 6 cores.

			Los Angeles		**New York**		**Chicago**	
Algorithm	R	Tr	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]
RAPTOR	○	●	24.0	3.4	14.6	3.1	14.4	1.8
RAPTOR-6	○	●	25.0	2.0	14.0	2.0	14.6	1.2
rRAPTOR	●	●	128.0	16.2	159.5	24.3	143.7	14.6
rRAPTOR-6	●	●	141.8	7.1	173.9	9.0	154.3	5.5
LD [7]	○	●	37.5	24.9	25.4	21.8	22.6	13.0
MLC [15]	○	●	21.7	38.1	16.8	32.2	9.8	17.6
SPCS [11]	●	○	28.4	37.9	29.6	53.7	29.3	36.3
SPCS-6 [11]	●	○	33.0	14.8	34.5	15.0	32.6	8.8

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. Experiments on the transit network of London reveal that RAPTOR is more than an order of magnitude faster than previous approaches. Moreover, 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. 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.

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

References

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *SEA*, LNCS 6630, pp. 230–241. Springer, 2011.

[2] H. Bast. Car or Public Transport – Two Worlds. In *Efficient Algorithms*, LNCS 5760, pp. 355–367. Springer, 2009.

[3] 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, pp. 290–301. Springer, 2010.

[4] R. Bauer, D. Delling, and D. Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 57(1):38–52, January 2011.

[5] A. Berger, D. Delling, A. Gebhardt, and M. Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *ATMOS*, OASIcs, 2009.

[6] A. Berger, M. Grimmer, and M. Müller-Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In *SEA*, LNCS 6049, pp. 35–46. Springer, 2010.

[7] G. Brodal and R. Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *ATMOS*, ENTCS 92, pp. 3–15, 2004.

[8] Chicago Transit Authority. `www.transitchicago.com`.

[9] D. Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, May 2011.

[10] D. Delling, B. Katz, and T. Pajor. Parallel Computation of Best Connections in Public Transportation Networks. In *IPDPS*, pp. 1–12. IEEE, 2010.

[11] D. Delling, B. Katz, and T. Pajor. Parallel Computation of Best Connections in Public Transportation Networks. Journal version. Submitted for publication. Online available at `i11www.iti.uni-karlsruhe.de/~pajor/paper/dkp-pcbcp-11.pdf`, 2011.

[12] D. Delling, T. Pajor, and D. Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Robust and Online Large-Scale Optimization*, LNCS 5868, pp. 182–206. Springer, 2009.

[13] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, LNCS 5515, pp. 117–139. Springer, 2009.

[14] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[15] Y. Disser, M. Müller-Hannemann, and M. Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In *WEA*, LNCS 5038, pp. 347–361. Springer, 2008.

[16] R. Geisberger. Contraction of Timetable Networks with Realistic Transfers. In *SEA*, LNCS 6049, pp. 71–82. Springer, 2010.

[17] General Transit Feed Specification. `code.google.com/transit/spec/transit_feed_specification.html`

[18] London Data Store. `data.london.gov.uk`

[19] Los Angeles County Metropolitan Transportation Authority. `www.metro.net`

[20] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel Shortest Path Algorithms for Solving Large-Scale Instances. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74, pp. 249–290. AMS, 2009.

[21] Metropolitan Transportation Authority of the State of New York. `www.mta.info`

[22] U. Meyer and P. Sanders. Δ-Stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[23] M. Müller-Hannemann and M. Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, LNCS 4359, pp. 246–263. Springer, 2007.

[24] M. Müller-Hannemann, M. Schnee, and L. Frede. Efficient On-Trip Timetable Information in the Presence of Delays. In *ATMOS*,OASIcs, 2008.

[25] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2007.

[26] Transport for London. `www.tfl.gov.uk`

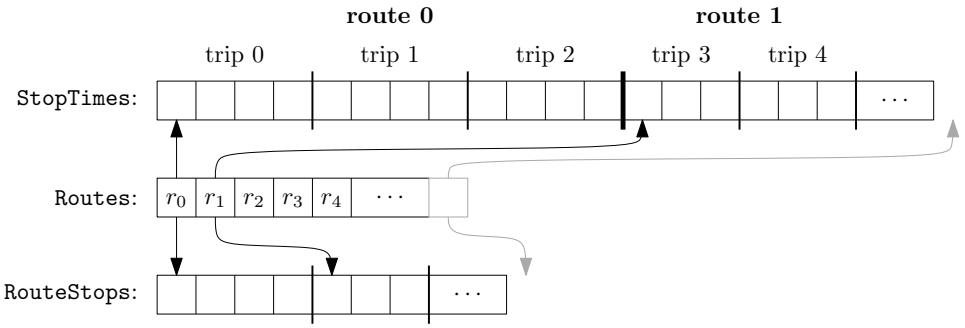

Figure 3: Illustration of the adjacency structure of routes.

A Data Structures

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 enable this operation, 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`$[i]$ points 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 3.

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 3.) 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 `RouteStops` associated with r_i. To find the earliest trip departing from some stop p along the route after some time τ, 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 `StopTimes`. In particular, when processing r_i with trip t, the arrival time of the next stop is determined by the subsequent entry in `StopTimes`. Furthermore, to check if an earlier trip improves r_i, we can jump $|r_i|$ (the length of the route r_i as stored in `Routes`) entries to the left to retrieve the departure time of the next earlier trip.

Other Operations. We still need to support some operations outside the main loop of the algorithm. For those, we need an array `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—this is important for our marking improvement. 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. `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, `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 `Stops` points to the first entries in `StopRoutes` and `Transfers` associated with stop p_i. See Figure 4.

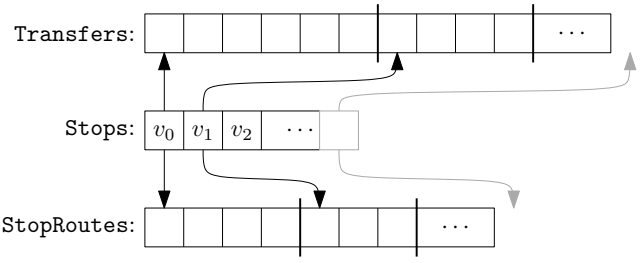

Figure 4: Illustration of the adjacency structure of stops.