Fast and Exact Public Transit Routing with Restricted Pareto Sets

Daniel Delling*

Julian Dibbelt†

Thomas Pajor‡

Abstract

We present a novel exact journey planning approach to computing a reasonable subset of multi-criteria Pareto sets in public transit networks. Our restriction is well defined and independent of the choice of algorithm. In order to compute the restricted Pareto set efficiently, we present *Bounded McRAPTOR*, a new set of algorithms that extend the well-known McRAPTOR algorithm. The fastest variant employs a novel pruning scheme based on carefully computed bounds. Experiments on large metropolitan networks show that a four-criteria restricted Pareto set can be computed faster by a factor of up to 65, while retaining the important journeys of the full Pareto set. This easily enables interactive applications in practice, making multi-criteria Pareto-optimal journey planning scalable without the need of a preprocessing-based speedup technique.

1 Introduction

Nowadays, millions of people use computer-based journey planning systems to obtain public transit directions. Most of these systems use modern algorithms developed by applied algorithms researchers in the last 15 years. In public transit systems, such algorithms typically rely on computing Pareto sets, often optimizing both number of transfers and travel time. Although NP-hard in general [10], computing such Pareto sets turns out to by feasible in public transit networks [12]. Beyond various problem specific extensions [9, 11, 13] of Dijkstra's classic algorithm, the most prominent algorithms are Connection Scan (CSA)[8], RAPTOR [7], Transfer Patterns [1, 3], Trip-Based Public Transit Routing [15], and graph labeling [5, 14]. All of these algorithms yield fast query times when computing two-criteria journeys in public transit networks of metropolitan size. For a detailed overview, see [2].

However, these algorithms fall short as soon as one tries to optimize more than the two mentioned criteria. For example, McRAPTOR (the version of RAPTOR which can optimize more than two criteria) is much slower than plain RAPTOR. The reason for this is two-fold: on the one hand, McRAPTOR has to use more complicated data structures (it is four times slower than RAPTOR, even when only optimizing two criteria [7]), while on the other hand, the size of the Pareto set increases significantly with each additional criterion, slowing down performance even more. Since the running time of all algorithms mentioned above is heavily affected by the size of the solution, all of them suffer from this performance degradation. Nonetheless, in practice we still want to use more than two criteria (such as walking duration, reliability, costs, accessibility, and others) when computing the Pareto set.

One possibility that has been considered to improve the performance is to tighten the domination rule [4]. A journey may already dominate another journey, if it is slightly worse in one criterion, as long as it is much better in another one. Unfortunately, this approach is only exact, if this rule is applied to the Pareto set in the end, after it has been fully computed. If applied at intermediate steps of the algorithm, the correctness guarantee is dropped, and the algorithm may miss important journeys [4]. Even worse, the journeys that are computed (and omitted) depend on the algorithm itself, which makes debugging a real production system very difficult.

In this work, we present a novel approach that computes for the first time a *restricted Pareto set* substantially faster than the full Pareto set, while still being exact. Our notion of the restricted Pareto set is well-defined and independent of the choice of algorithm. The key idea is to only allow journeys in the restricted Pareto set that are not arriving substantially later or with substantially more trips than

*Apple Inc. ddelling@apple.com
†Apple Inc. jdibbelt@apple.com
‡Apple Inc. tpajor@apple.com

journeys in the (smaller but full) Pareto set optimizing arrival time and number of trips. To compute the restricted Pareto set efficiently, we introduce *Bounded McRAPTOR* (BMRAP), a new family of algorithms that utilize RAPTOR and McRAPTOR as building blocks. The fastest variant, Tight-BMRAP, is a sophisticated three stage pruning scheme. First, it executes a series of quick RAPTOR runs to obtain tight bounds at every stop of the network. Then, in a subsequent McRAPTOR run these bounds are used to precisely prune journeys (anywhere in the network) that are provably not contained in the restricted Pareto set.

Our experiments reveal that using our new algorithm, we can find a restricted four-criteria Pareto set (arrival time, number of trips, walking duration, and number of buses) in roughly twice the time of finding a two-criteria Pareto set with plain RAPTOR (arrival time and number of trips). This is still fast enough for practical applications. However, the surprising observation is that Dijkstra's algorithm, which optimizes only one criterion, is also roughly twice as slow as RAPTOR [7]. This brings the performance of Bounded McRAPTOR on four criteria to the same ballpark as computing the earliest arrival time with Dijkstra's algorithm. Finally, as our approach does not utilize a costly preprocessing phase, it can also be used in dynamic scenarios, making it overall very appealing for a production system.

This paper is organized as follows. Section 2 settles preliminaries, defining the problem domain, and reviewing important algorithmic building blocks. Section 3 introduces our notion of a restricted Pareto set. Section 4 introduces Bounded McRAPTOR, a family of new algorithms that compute the restricted Pareto set quickly. Section 5 presents an experimental evaluation of the quality and performance of our new approach on several metropolitan networks. Section 6 concludes with a summary and a discussion of future work.

2 Preliminaries

This section defines preliminary notion required throughout the paper. It also recaps the RAPTOR and McRAPTOR algorithms, which we require as building blocks.

2.1 Input and Output.

In this work the input is represented by *timetables* $\mathbb{T} = (\Pi, \mathcal{P}, \mathcal{R}, \mathcal{T}, \mathcal{F})$, where $\Pi \subseteq \mathbb{N}_0$ is the period of operation (think of it as seconds of the day), $\mathcal{P}$ a set of *stops*, $\mathcal{T}$ a set of *trips*, $\mathcal{R}$ a set of *routes*, and $\mathcal{F}$ a set of *footpaths*. Stops $p \in \mathcal{P}$ represent distinct locations where a vehicle can be boarded or alighted, such as a bus stop or a train platform. Trips represent travel of individual vehicles in the network. Each trip $t \in \mathcal{T}$ consists of a sequence of stops (where the vehicle stops) with associated arrival and departure times $\tau_{\text{arr}}(t,p), \tau_{\text{dep}}(t,p) \in \Pi$. The set of trips is partitioned into a (typically substantially smaller) set of routes, such that each two trips t_i, t_j of the same route $r \in \mathcal{R}$ follow the exact same sequence of stops, and neither trip overtakes the other. (Note that the set of routes $\mathcal{R}$ can be derived from a given set of trips $\mathcal{T}$.) Finally, each footpath $f \in \mathcal{F}$ represents walking between two stops p_1, p_2 with an associated duration $\ell(p_1, p_2) \in \mathbb{N}_0$. If the input data does not contain any footpaths, we programmatically add them between each pair of stops p_i, p_j whose geographic distance is below a certain threshold.

Given a timetable $\mathbb{T}$, source and target stops $p_s, p_t \in \mathcal{P}$, and a departure time $\tau \in \Pi$, a *depart-after* query computes a set of *journeys* $\mathcal{J}$, such that each journey $J \in \mathcal{J}$ is an alternating sequence of footpaths and (partial) trips—sometimes called *journey legs*—in the order of travel, starting at p_s, ending at p_t, and not departing at p_s before τ. Similarly, for an *arrive-by* query, the input specifies an arrival time τ at p_t, and any journey must not arrive at p_t after τ.

Each journey $J \in \mathcal{J}$ can be associated with a set of *optimization criteria*, such as arrival time, number of trips, walking duration, and more. We say that a journey J_1 *dominates* another journey J_2, written $J_1 \preccurlyeq J_2$, if J_1 is better or equal in every criterion than J_2. A set of pairwise non-dominating journeys is called a *Pareto set*. In this paper we consider the *multi-criteria problem*, whose objective is to compute a Pareto set of journeys—or a well-defined subset thereof (see Section 3).

2.2 Algorithms.

Several algorithms that solve the multi-criteria problem exist. They can be categorized into either classic graph-based algorithms [9, 11, 13], or algorithms that use the timetable data structures directly [7, 8, 15]. Algorithms from the latter category tend to be more efficient, from which two prominent ones are RAPTOR and McRAPTOR [7]. RAPTOR is explicitly designed to optimize arrival time and number of trips, which makes it very fast—about twice as fast as Dijkstra's algorithm, which only considers arrival time. The more general McRAPTOR algorithm can take arbitrary many further criteria into account, however, at the expense of being slower. Our new approach will require both algorithms as building blocks. We therefore recap them in the following.

2.2.1 RAPTOR.

Given an input query (p_s, p_t, τ), the RAPTOR algorithm, herein called *RAP* for short, is a dynamic programming approach, working in discrete *rounds* $k = 1, 2, \ldots, K$. In its basic variant [7], the k-th round computes the earliest arrival time to all stops, which are reachable using exactly k trips. To that extent,

it maintains an arrival time label $\tau_{\text{arr}}(k,p) \in \Pi$ for each round and stop, initialized to ∞, except for $\tau_{\text{arr}}(0,p_s) := \tau$. The entirety of arrival time labels is also called the *search space* $\mathcal{S}$ in the rest of the paper.

Each round k performs two phases: scanning routes and relaxing footpaths. The first phase takes the stops, whose arrival time improved in the previous round (in the first round this is only the source stop), and selects all routes $r \in \mathcal{R}$ that serve those stops. Each route is then scanned by traversing its stops $p \in r$ in order of travel, while maintaining the earliest possible trip $t \in r$ that can be taken subject to the previous round's arrival times $\tau_{\text{arr}}(k-1,p)$. During the same scan, the algorithm minimizes the arrival time labels of the current round $\tau_{\text{arr}}(k,p)$ with the arrival time $\tau_{\text{arr}}(t,p)$ of the currently selected trip. The second phase of the round relaxes footpaths $(p_i,p_j,\ell) \in \mathcal{F}$ from all improved stops p_i of the first phase by setting $\tau_{\text{arr}}(k,p_j) = \min\{\tau_{\text{arr}}(k,p_j), \tau_{\text{arr}}(k,p_i) + \ell(p_i,p_j)\}$.

In order to maintain a proper Pareto set at every stop (the arrival time labels at p must never increase with higher rounds), Delling et al. [7] propose two options. The first, called *label copying* approach, sets $\tau_{\text{arr}}(k,\cdot) := \tau_{\text{arr}}(k-1,\cdot)$ at the beginning of each round k. The second, called *local pruning* approach, instead maintains (across rounds) at each stop the earliest arrival time $\tau^*(p)$ that the algorithm encountered during its execution so far. A label $\tau_{\text{arr}}(k,p)$ is then only updated, if it is a strict improvement over $\tau^*(p)$.

Another optimization is *target pruning*, which only updates a label if it is an improvement over $\tau_{\text{arr}}(k,p_t)$ at the target stop p_t. Moreover, the algorithm may terminate after round k, if no labels have been improved during that round. This condition is often called *stopping condition*.

If one is interested in arrive-by instead of depart-after queries, RAP can be run in reverse with arrival time labels exchanged for labels maximizing departure time. The algorithm is then initialized with $\tau_{\text{dep}}(0,p_t) = \tau$, and routes and footpaths are simply scanned in reverse direction.

An important distinction from [7] is that in our experiments (cf. Section 5) the footpaths are not guaranteed to form full cliques, and computing their transitive closure, as sometimes done in previous works, is not feasible in practice. To not obtain two or more consecutive footpaths on a single journey leg, the second phase of each round k is augmented by a second set of labels $\tau^{\text{fp}}_{\text{arr}}(k,p)$. While relaxing footpaths, values are now read from $\tau_{\text{arr}}(k,p)$ and written to $\tau^{\text{fp}}_{\text{arr}}(k,p)$. Accordingly, the first phase of the subsequent round $k+1$ now reads its values from $\tau^{\text{fp}}_{\text{arr}}(k,p)$. Note that this technique carries over to all algorithms of this paper.

2.2.2 McRAPTOR. The McRAPTOR algorithm, herein called *MRAP* for short, extends RAP by supporting arbitrary many additional criteria on top of arrival time and number of trips. Examples that have been studied previously include walking duration, reliability, fare zones, and cost [4, 7].

The overall approach of working in rounds, each scanning routes and relaxing footpaths, is identical to RAP. The main distinction is that the algorithm now maintains *bags* $B(k,p)$ of (multiple) Pareto-optimal labels L at each round k and stop p. Each label has one value for every optimization criterion that is being considered by the algorithm. Routes are scanned by maintaining a *route bag* B_r, containing all Pareto-optimal labels (with associated trips) that have been picked up from bags $B(k-1,p)$ of the previous round (along the route). At each stop the route bag B_r is merged into $B(k,p)$, thereby removing dominated labels from $B(k,p)$. Similarly, footpaths (p_i,p_j,ℓ) are relaxed by taking all labels $L \in B(k-1,p_i)$, adding ℓ to its appropriate criteria, and then merging L into $B(k,p_j)$.

The local and target pruning techniques as well as the stopping condition naturally carry over from RAP as well. See [7] for details.

Due to the more complicated dynamic data structures in the algorithm (bags instead of scalar values), MRAP has been shown to be about a factor of four slower than RAP when optimizing the same criteria (arrival time and number of trips) [7]. On top of that, each additional criterion may increase the number of labels per bag significantly, which has a direct impact on the running time (the bag merge operation is quadratic in the number of contained labels). As a result, MRAP can quickly become too slow for practical use [4, 7].

3 Restricted Pareto Set

When computing public transit journeys in practice, solving the multi-criteria problem is generally considered to be the approach of choice [2]. The reason is quite simple: when planning a transit journey, factors beyond arrival time are important to the user. These may include the number of trips, the amount of walking, or the cost of the journey. When used as optimization criteria in the multi-criteria problem, the resulting Pareto set is guaranteed to contain journeys for *all* possible tradeoffs in these criteria.

However, computing entire Pareto sets has downsides as well. First, with every additional criterion, the number of Pareto optimal journeys may grow exponentially [10]. This slows down the algorithm, often to an extent that it becomes impractical [2, 4]. To make things worse, most of the journeys may not end up being shown to the user after all. Firstly, only a limited number can

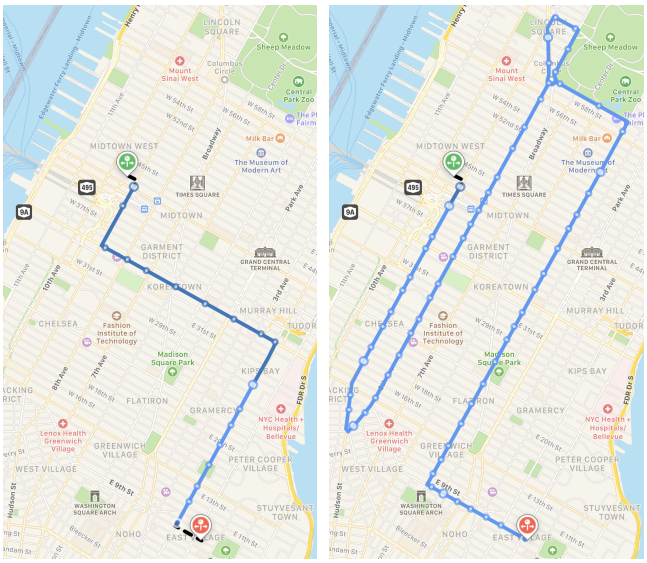

Figure 1: Two Manhattan bus journeys that are in the same Pareto set. The left one has 2 trips, takes 46 minutes, and requires 543 m of walking, while the right one has 7 trips, takes 135 minutes, and requires 241 m of walking. The full Pareto set often contains journeys with undesirable tradeoffs like these.

be reasonably presented. But more crucially, it can be easily seen that many Pareto-optimal journeys are quite undesirable outliers, which a user is very unlikely to take. For example, when considering walking duration, the Pareto set may contain journeys with an excessive number of trips (and very late arrival) in order to save an insignificant amount of walking. See Figure 1 for an example. In practice, a ranking algorithm is typically used to identify the best journeys from the Pareto set in a post-processing step [4].

In order to avoid computing undesirable journeys in the first place, previous approaches have suggested to remove labels from the (intermediate) Pareto sets during the execution of the algorithm, if their tradeoffs are particularly bad. For example a label L_1 may lead to the removal of a label L_2, if L_2 has just one minute earlier arrival at the expense of walking two hours longer. Unfortunately, in the context of journey planning, this technique has no provable guarantee on which labels (journeys) are computed at the destination; see [4] for details. Even worse, results depend on the choice of algorithm and the order in which it processes the stops [4].

To introduce a more principled approach, this paper proposes the notion of *restricted Pareto sets*, which is both well-defined and independent of the choice of algorithm. In other words, any algorithm that computes a restricted Pareto set outputs the same set of journeys.

Our motivation is to formally exclude journeys that have undesirable tradeoffs in their criteria (outliers), which would most likely not be presented to the user anyway (as verified by our experiments in Section 5). For that, we start with the (much simpler) Pareto set of journeys $\mathcal{J}^*$, which only optimizes arrival time and number of trips, and which can be very quickly computed by RAP. In the following we will also call these journeys *anchor journeys*. Each anchor journey $J^* \in \mathcal{J}^*$ represents the earliest arrival time $\tau_{\text{arr}}(J^*)$ at which the destination can be reached with exactly $\text{tr}(J^*)$ trips.

We argue that any other Pareto-optimal journey that takes further criteria into account, should neither arrive substantially later than $\tau_{\text{arr}}(J^*)$, nor use a significant number of additional trips compared to $\text{tr}(J^*)$.

More formally, let $\mathcal{J} \supseteq \mathcal{J}^*$ be the entire Pareto set of journeys that optimizes arbitrary criteria, but always including arrival time and number of trips. For any given journey $J \in \mathcal{J}$, let now J^* be the *corresponding* anchor journey from $\mathcal{J}^*$ that has the maximum number of trips smaller or equal to $\text{tr}(J)$. Note that J^* is unique and always exists (it may coincide with J). Also, let $\sigma_{\text{arr}} \in \mathbb{N}_0$ be an *arrival time slack* value (in number of seconds), and $\sigma_{\text{tr}} \in \mathbb{N}_0$ be a *trip slack* value (in number of trips). Then, the *restricted Pareto set* of journeys $\mathcal{J}^{\text{R}}$ is a subset $\mathcal{J}^* \subseteq \mathcal{J}^{\text{R}} \subseteq \mathcal{J}$ defined as

$$\begin{aligned} \mathcal{J}^{\text{R}} := \{J \in \mathcal{J} \mid \text{its anchor journey} J^* \in \mathcal{J}^* \text{ has} \\ \tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J^*) + \sigma_{\text{arr}} \text{ and} \\ \text{tr}(J) \leq \text{tr}(J^*) + \sigma_{\text{tr}}\} \end{aligned} \tag{3.1}$$

Note that if we set $\sigma_{\text{arr}} = 0$ and $\sigma_{\text{tr}} = 0$, we exactly obtain $\mathcal{J}^{\text{R}} = \mathcal{J}^*$; similarly, for $\sigma_{\text{arr}} = \infty$ and $\sigma_{\text{tr}} = \infty$, we obtain $\mathcal{J}^{\text{R}} = \mathcal{J}$. This allows us to control the size of the restricted Pareto set and the kind of journeys it contains. Figure 2 illustrates our definition of the restricted Pareto set.

4 Bounded McRAPTOR

This section presents our new family of algorithms for computing restricted Pareto sets $\mathcal{J}^{\text{R}}$ (as introduced in Section 3). All algorithms share the common approach of extending MRAP (cf. Section 2.2.2) to use additional *time bounds* for pruning labels in the network. We call each algorithm a variant of *Bounded McRAPTOR*, or *BMRAP* for short.

The first algorithm (Section 4.1) is called *Self-BMRAP* and is the simplest one: it extends MRAP to prune labels based on the earliest arrival times as they are being computed at the target stop. Being quite simple, this approach is limited in that it may actually compute a superset of $\mathcal{J}^{\text{R}}$ and does not handle trip slacks. (It assumes $\sigma_{\text{tr}} = \infty$.)

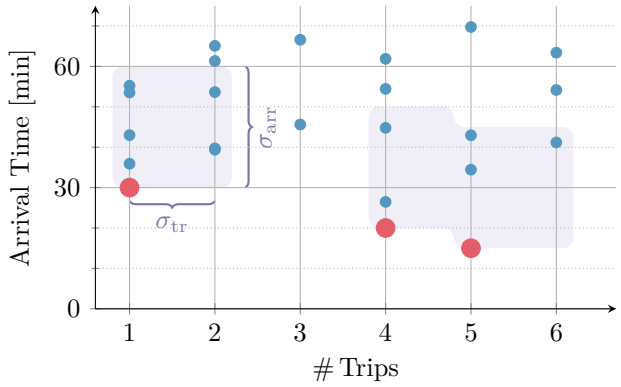

Figure 2: Illustrating the solution space of a restricted Pareto set with $\sigma_{\text{arr}} = 30\,\text{min}$ and $\sigma_{\text{tr}} = 1$. The thick red dots represent the Pareto-optimal (for arrival time and number of trips) anchor journeys $\mathcal{J}^*$. According to the slack values, each such journey induces a subspace of the full Pareto set $\mathcal{J}$ with further criteria. The restricted Pareto set $\mathcal{J}^{\text{R}}$ is precisely formed of the journeys (small blue dots) that fall within the shaded areas.

We improve upon this with a two-staged approach called *Target-BMRAP* (Section 4.2). It first runs RAP to obtain the precise earliest arrival times at the target stop, which are then passed to MRAP as input. This approach no longer computes a superset of $\mathcal{J}^{\text{R}}$, however, it is still unable to handle trip slacks. (It again assumes $\sigma_{\text{tr}} = \infty$.) Also, the bounds passed to MRAP are only tight for the target stop, which limits their effectiveness for pruning.

Finally, our most sophisticated three-staged approach, called *Tight-BMRAP* (Section 4.3), is a highly effective pruning scheme that results in tight bounds at every stop and every round. In addition, it can handle both arrival time and trip slacks. The first stage runs a regular forward RAP query. The second stage then runs a series of reverse RAP queries on a common search space in order to build a set of tight bounds at every stop and round. Moreover, the reverse RAPs use the search space of the first stage for pruning, which makes them very quick. The bounds produced by the second stage are then finally used for pruning a final single forward MRAP query in the third stage.

As the bounds are tight, MRAP is sped up significantly in the third stage. Indeed, our experiments in Section 5 show that computing the restricted Pareto set on four criteria with Tight-BMRAP takes only about twice the time of a single plain RAP query.

4.1 One Stage: Self-Bounded McRAPTOR. This section presents *Self-BMRAP*, our simplest approach for pruning journeys that are not part of the restricted Pareto set $\mathcal{J}^{\text{R}}$. It runs a single MRAP stage.

For that, recall that MRAP works in rounds $k = 1, 2, \ldots, K$. After each round k, the union of the bags $\cup_{k' \leq k} B(k', p_t)$ contains exactly the partial Pareto set of journeys $\mathcal{J}_{\leq k}$ with at most k trips. In particular, it also contains all anchor journeys from $\mathcal{J}^*_{\leq k}$. Recall that these induce the restricted Pareto set $\mathcal{J}^{\text{R}}$ (cf. Section 3).

To leverage this fact, the algorithm now maintains an additional earliest arrival time value τ^* of the target stop p_t, initially set to ∞. After each round k, the value is updated by calculating the minimum of itself with the arrival times of all labels $L \in B(k, p_t)$ at the target stop. By doing so, the value τ^* now represents the earliest arrival time of any journey contained in $\mathcal{J}^*_{\leq k}$.

Since we are not interested in journeys J with an arrival time $\tau_{\text{arr}}(J) > \tau^* + \sigma_{\text{arr}}$ (by definition of the restricted Pareto set), MRAP may from now on prune (at any stop) any label L with $\tau_{\text{arr}}(L) > \tau^* + \sigma_{\text{arr}}$.

Note that this technique may compute more journeys at the target stop than the actual restricted Pareto set $\mathcal{J}^{\text{R}}$ contains. The reason is that MRAP by design propagates entire bags instead of individual labels. Therefore, when merging a bag into $B(k, p_t)$ before τ^* gets updated, some of the labels may end up with an arrival time higher than $\tau^* + \sigma_{\text{arr}}$.

Moreover, this approach is unable to handle trip slacks (it assumes $\sigma_{\text{tr}} = \infty$), since each intermediate round must be processed in full. Later rounds may still produce journeys that are part of $\mathcal{J}^{\text{R}}$, and we do not know on which intermediate solutions they may be based.

4.2 Two Stages: Using RAPTOR as Input. Our second approach, called *Target-BMRAP*, has two stages. The first stage runs the regular RAP algorithm to compute an earliest arrival time label $\tau_{\text{arr}}(k, p_t)$ for each round k at the target stop p_t. The second stage then runs MRAP, taking the arrival times at the target stop $\tau_{\text{arr}}(\cdot, p_t)$ from the first stage as an additional input. Consequently, round k of MRAP has the precise earliest arrival time $\tau_{\text{arr}}(k, p_t)$ at the target stop available from the beginning.

A label in round k at stop p is now pruned, if its arrival time exceeds $\tau_{\text{arr}}(k, p_t) + \sigma_{\text{arr}}$. Note that if RAP was run with local pruning enabled (cf. Section 2.2.1), the MRAP stage must instead prune using $\tau_{\text{arr}}(k', p_t)$ for the largetst $k' \leq k$ that yields a non-infinite value.

While this technique no longer outputs a superset of $\mathcal{J}^{\text{R}}$, it is still unable to handle trip slacks (it again assumes $\sigma_{\text{tr}} = \infty$). Moreover, the given arrival time bounds are only tight at the target stop, which may result in unnecessarily computed labels at intermediate stops and limits the achievable speedup.

4.3 Three Stages: A Tight Pruning Scheme. To obtain tight bounds for MRAP at every stop and round, we propose our most sophisticated three-stage pruning scheme, called *Tight-BMRAP*. It first runs forward RAP, followed by a series of reverse RAPs, and finally one forward MRAP. The approach outputs the exact restricted Pareto set $\mathcal{J}^{\mathrm{R}}$ and can handle arrival time and trip slacks.

Given a (p_s, p_t, τ) depart-after query, the first stage runs the forward RAP algorithm between stops p_s and p_t at departure time τ. This results in the set of anchor journeys $\mathcal{J}^*$. Moreover, as described in Section 2.2.1, the search space $\overrightarrow{\mathcal{S}}$ now contains the earliest arrival time $\tau_{\mathrm{arr}}(k, p)$ at every stop p and round k. Also recall that stops that have not been visited in round k have their earliest arrival time set to ∞.

The second stage now has the goal of building tight bounds at *every* stop and round, which are eventually used by MRAP for pruning in the third stage.

In order to understand how the bounds work, consider the following example of $\mathcal{J}^*$ (from the first stage) containing two journeys: the first has three trips and a 10 am arrival; the second has four trips and a 9 am arrival. Assume that the arrival time slack is 30 min and (for simplicity) the trip slack is 0. Suppose MRAP (third stage) is currently in round 2. A label L that is considered at some stop p corresponds to a partial journey to p using exactly two trips. From J^* we know that according to the slack values, any arrival at the target p_t must be before 10:30 am with three trips (i.e., one additional trip from p), or before 9:30 am with four trips (i.e., two additional trips from p).

Now let $\tau^{(3)}_{\mathrm{dep}}(1, p)$ be the *latest departure time* at p for which p_t can still be reached before 10:30 am with one onward trip (from p), and $\tau^{(4)}_{\mathrm{dep}}(2, p)$ the latest departure time for which p_t can still be reached before 9:30 am with two onward trips (from p). This means, using

$$\tau^*_{\mathrm{dep}}(p) := \max\{\tau^{(3)}_{\mathrm{dep}}(1, p), \tau^{(4)}_{\mathrm{dep}}(2, p)\} \tag{4.2}$$

is a viable (and tight) upper bound that may be used for pruning the label L.

The crucial observation here is that for any anchor journey J^*, the aforementioned latest departure time values $\tau_{\mathrm{dep}}(k, p)$ exactly correspond to the search space $\overleftarrow{\mathcal{S}}$ of a reverse RAP query from p_t at time $\tau_{\mathrm{arr}}(J^*) + \sigma_{\mathrm{arr}}$. This motivates the following construction scheme for the bounds.

Formally, let K denote the maximum number of trips of any journey in $\mathcal{J}^*$. We first initialize a blank search space $\overleftarrow{\mathcal{S}}$ for a fixed number of $m := K + \sigma_{\mathrm{tr}}$ rounds, with all latest departure time values $\tau_{\mathrm{dep}}(k, p)$ set to $-\infty$. (Note that adding σ_{tr} to K ensures we handle trip slacks correctly.)

The stage now considers each anchor journey $J^* \in \mathcal{J}^*$ in the order of earliest to latest arrival time (i.e., most to fewest trips). For each anchor journey J^* in that order, the stage runs one reverse RAP query from the target stop p_t at time $\tau_{\mathrm{arr}}(J^*) + \sigma_{\mathrm{arr}}$. Importantly, the search space $\overleftarrow{\mathcal{S}}$ is not cleared between the second-stage RAP runs, and we stop the current reverse RAP execution after exactly $n := \mathrm{tr}(J^*) + \sigma_{\mathrm{tr}}$ rounds. (Note that if $n >= \mathrm{tr}(J'^*)$, where $J'^* \in \mathcal{J}^*$ is the journey with the next higher number of trips, we have to set $n := \mathrm{tr}(J'^*) - 1$ instead.)

Recall that MRAP (third stage) is supposed to prune labels at any stop p by the time bound corresponding to the *remaining* number of trips from p towards the target stop. If different numbers of remaining trips from p may yield feasible solution in $\mathcal{J}^{\mathrm{R}}$, we must guarantee that the value used by MRAP for pruning corresponds to the *weakest* (i.e., highest) bound in order to not lose any journeys in $\mathcal{J}^{\mathrm{R}}$. This corresponds to Equation 4.2 in our example above.

This guarantee can be achieved by carefully overlapping the labels in $\overleftarrow{\mathcal{S}}$ that are read and written by the reverse RAP runs during the second stage. More precisely, the departure time labels in round k of the current reverse RAP are read from $\tau_{\mathrm{dep}}(k+(m-n)-1, p)$ and written to $\tau_{\mathrm{dep}}(k+(m-n), p)$. (Recall that n is the number of executed rounds of the current reverse RAP.)

Since the search space $\overleftarrow{\mathcal{S}}$ is not re-initialized between RAP runs, labels at the beginning of each round must be carried over from the previous round using the maximum operation. In particular, local pruning may not be used, and target pruning has to be disabled as well. Otherwise, some required $\tau_{\mathrm{dep}}(k, p)$ values may remain untouched at $-\infty$, which in turn may falsely prune required solutions from $\mathcal{J}^{\mathrm{R}}$ during MRAP in the third stage.

See Figure 3 for an illustration on how the overlapping labels in $\overleftarrow{\mathcal{S}}$ are gradually built by the second stage.

As mentioned, the third stage of our scheme runs forward MRAP at the original departure time τ_{dep}. Having computed the labels of $\overleftarrow{\mathcal{S}}$ in the staggered way as explained above, now enables the following simple pruning rule. During round k at stop p, any label L may be discarded in MRAP, if $\tau_{\mathrm{arr}}(L) > \tau_{\mathrm{dep}}(m-k, p) \in \overleftarrow{\mathcal{S}}$ holds. By construction these bounds are tight at every stop and for any round, precisely resulting in the journeys of the restricted Pareto set $\mathcal{J}^{\mathrm{R}}$.

Recall from above that in order to obtain these tight bounds, Tight-BMRAP runs $1 + |\mathcal{J}^*|$ RAP executions (one forward RAP, followed by $|\mathcal{J}^*|$ reverse RAPs)

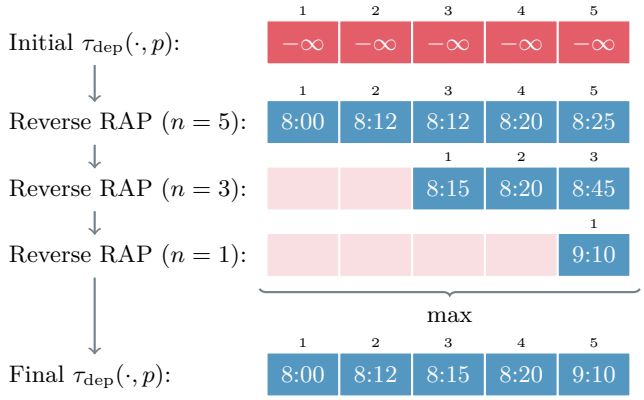

Figure 3: Illustrating the search space at a stop p being gradually built in the second stage of Tight-BMRAP. Given anchor journeys of 1, 3, and 5 trips (assuming $\sigma_{\text{tr}} = 0$ for simplicity), the search space is initialized to $-\infty$ for $m = 5$ rounds. Then, a reverse RAP is run for each anchor journey, in the order of most to fewest trips. Round k of each reverse RAP writes labels at position $k + (m - n)$, resulting in the staggered alignment as illustrated by this figure. Maximizing the departure times across RAP runs finally results in the tight bounds that are used for pruning at p in the third stage.

in the first two stages. In particular, running such many reverse RAPs in stage two can become costly. To accelerate those runs, a similar pruning scheme can be applied to the reverse RAPs as well. It uses the earliest arrival time labels $\tau_{\text{arr}}(k, p) \in \overrightarrow{\mathcal{S}}$ from the first stage for pruning during the second stage.

More precisely, in round k of any reverse RAP (computing journeys of up to n trips; see above), a departure time label $\tau_{\text{dep}}(k, p)$ can be discarded at stop p, if $\tau_{\text{dep}}(k, p) < \tau_{\text{arr}}(n-k, p)$ holds. Intuitively, this avoids computing latest departure time labels that arrive at p before the corresponding earliest arrival time, at which point any journey in the MRAP stage would be pruned anyway.

Additionally, if the first-stage RAP has target pruning enabled, its target pruning rule must be relaxed in order to incorporate the arrival time slack σ_{arr} into the labels that are used by the second-stage reverse RAPs for pruning. More precisely, consider the best arrival time label $\tau^*(p_t)$ at the target stop during the first-stage RAP. The algorithm may only prune a label $\tau_{\text{arr}}(k, p)$, if $\tau_{\text{arr}}(k, p) > \tau^*(p_t) + \sigma_{\text{arr}}$ holds.

With the second-stage RAPs using the first stage for pruning, and the third-stage MRAP using the second stage for pruning, we obtain a sophisticated pruning scheme that is able to compute the restricted Pareto set very efficiently. In fact, our experiments in Section 5 show that the running time of the complete Tight-BMRAP scheme optimizing four criteria is only a factor of two slower over computing the bi-criteria Pareto set with the plain RAP algorithm. In other words, the time of running stages two and three takes about as long as running the single RAP execution of the first stage.

5 Experiments

This section presents an experimental study to evaluate the performance of our new Bounded McRAPTOR (BM-RAP) algorithms as well as the quality of the restricted Pareto sets they compute.

For that we consider the following four minimization criteria to compute (restricted) Pareto sets: the arrival time, the number of trips, the total walking duration, and the number of buses.

Our inputs are realistic public transit networks from the following metropolitan regions: the greater Berlin region (VBB), Budapest, Melbourne, New York City, Paris, Prague, and Rome. Each input is openly available from the respective transit agency via a GTFS feed, which has been downloaded in August 2017. From the GTFS feeds we extracted two consecutive days of timetable data, representing 14 and 15 August 2017 (a Monday and Tuesday).

Since footpaths were not available for all inputs, we programmatically generated them (for all instances) as follows. We add a footpath between every pair of stops whose straight-line distance is not longer than 500 m. The duration of the footpath is derived from its length by assuming a walking speed of 3 kph. Note that this scheme does not guarantee that the footpaths form full cliques—a requirement in some previous papers [7, 8]. To still restrict transfers between trips to a single footpath, we adapt all considered algorithms to execute two rounds per trip, the first only scanning routes, and the second only relaxing footpaths. See Section 2.2.1 for details. The numbers of stops, routes, trips, departure events, and footpaths for all instances are presented in Table 1.

We implemented all algorithms in C++ and compiled them on LLVM 9.1 with full optimization. All experiments were conducted on a Mac Pro with a quad-core Intel Xeon E5-1620 v2 CPU with 64 GiB of RAM, running macOS 10.13.6. All running times are sequential.

In our evaluation, we ran for each algorithm the same set of 1 000 queries. The source and target stops of each query are uniformly selected with probability proportional to the number of trips serving the stop. This models that users are more likely to start and end their journeys at stops that have more transit service.

Table 1: Size figures for our input instances.

Instance	Stops	Routes	Trips	Departure Events	Foot Paths
Berlin (VBB)	41,302	6,819	120,204	2,494,242	289,110
Budapest	7,462	6,349	504,906	9,268,386	106,144
Melbourne	27,209	2,940	69,642	2,501,490	250,594
New York City	18,211	2,478	122,916	4,100,054	695,408
Paris	52,390	6,755	262,524	5,246,264	774,733
Prague	6,704	2,479	66,570	1,219,576	34,752
Rome	8,590	2,694	247,338	7,246,274	107,988

All queries are depart-after queries with the departure time selected uniformly at random from within the first 24 hours of the timetable data. We limit the maximum number of trips per journey to 8, which is reasonable for metropolitan networks.

5.1 Quality of Restricted Pareto Sets. The first experiment evaluates the quality of our proposed restricted Pareto set definition (see Section 3). Recall that the motivation of the restricted Pareto set (which is a subset of the full Pareto set) is to not contain outlier journeys with bad tradeoffs. For example, a tradeoff that saves one minute of walking time by arriving two hours later might most likely be considered undesirable. To control the tradeoffs that make up the restricted Pareto set, we introduced the notion of arrival time and trip slacks (cf. Section 3).

By design, our algorithms compute the restricted Pareto set exactly (cf. Section 4). However, often one is interested to (additionally) rank the journeys by an algorithm that is independent of the way the journeys have been computed. This allows one to select the "best" k journeys (from any set) and then send only those to the user.

This section therefore evaluates how the restricted Pareto set $\mathcal{J}^{\mathrm{R}}$ retains the top-ranked journeys from the corresponding full Pareto set $\mathcal{J}$, using a known ranking technique. More precisely, we use the *fuzzy dominance* algorithm, introduced by Delling et al [4]: given any set of journeys as input, it assigns each journey a score in the range $[0, 1]$ by leveraging fuzzy logic to tighten the notion of Pareto dominance. The fuzziness of each criterion is specified by a pair (χ, ϵ), which defines a Gaussian (centered at $x = 0$) with value χ for $x = \epsilon$. The journeys are then ordered by their score, and the top k ones are returned. Following [4], we set the fuzziness parameters (χ, ϵ) for our criteria as follows: $(0.8, 5)$ for walking duration, $(0.8, 1)$ for arrival time, $(0.1, 1)$ for the number of trips, and $(0.8, 2)$ for the number of buses.

To finally define the *quality value* for a restricted Pareto set $\mathcal{J}^{\mathrm{R}}$, we also compute the corresponding full Pareto set $\mathcal{J}$, and apply the fuzzy dominance scoring algorithm to both sets independently. We now compare the top five journeys ranked journeys from the full Pareto set (denoted $\mathcal{J}_{(5)}$) to the top five journeys from the restricted Pareto set (denoted $\mathcal{J}^{\mathrm{R}}_{(5)}$). More precisely, we calculate the fraction of journeys $J \in \mathcal{J}_{(5)}$ that are also contained in $\mathcal{J}^{\mathrm{R}}_{(5)}$, weighted by their score (in $\mathcal{J}_{(5)}$). Intuitively, missing a journey in $\mathcal{J}^{\mathrm{R}}_{(5)}$ that has a higher score in $\mathcal{J}_{(5)}$ is emphasized more by the quality value.

Table 2: Quality evaluation of the restricted Pareto set on Paris for varying arrival time and trip slack values. The criteria optimized are arrival time, number of trips, walking duration, and number of buses. See 5.1 for details.

Arr.-Slack	Trip Slack 0	1	2	4	8
5	77.8 %	86.9 %	87.4 %	87.5 %	87.5 %
10	79.1 %	88.5 %	89.2 %	89.3 %	89.3 %
15	80.5 %	90.0 %	90.8 %	90.8 %	90.8 %
30	82.4 %	92.3 %	93.2 %	93.4 %	93.4 %
60	84.1 %	94.3 %	95.3 %	95.5 %	95.5 %
120	84.8 %	95.3 %	96.4 %	96.6 %	96.6 %

To evaluate the quality value for different slack values, we first focus on our Paris instance. Table 2 presents the average quality figure for arrival time slacks of 5, 10, 15, 30, 60, and 120 minutes, as well as trip slack values of 0, 1, 2, 4, and 8 trips. The (restricted) Pareto sets were computed using arrival time, number of trips, walking duration, and the number of buses as criteria. (Recall that in our experiments we compute at most 8 trips.) Each entry in the table is the average value over the same set of 1 000 depart after queries.

We observe that with the strongest setting of $\sigma_{\mathrm{arr}} = 5$ and $\sigma_{\mathrm{tr}} = 0$, the quality is 77.8 %. Increasing the trip slack to 1 significantly improves the quality to 86 %. Further increasing the trip slack to 2 yields a smaller improvement (87.4 %), and going above 2 has almost no gain. Thus, fixing the trip slack to 2 and varying the arrival time slack, we observe that the quality steadily increases up to 95.3 % (for $\sigma_{\mathrm{arr}} = 60$). While we do not observe a plateau like for trips, further doubling the arrival time slack to 120 minutes yields only diminishing (sublinear) returns.

Since the sweet spot is between 1 and 2 regarding the trip slack value and between 30 and 60 minutes regarding the arrival time slack value, we restrict the following experiments to those values.

Note that results on the other instances are similar (not shown here).

Table 3: Evaluating our new algorithms on Paris. We report the criteria each algorithm optimizes, followed by the slack values for arrival time and number of trips (where applicable). The remaining columns report the avg. number of rounds, scanned routes, relaxed footpath, computed journeys; the performance as the avg. running time (in ms) and its standard deviation; and quality values for the top and the top five journeys (cf. Section 5.1).

Algorithm	Arr.	Trp.	Wlk.	Bus	Slack Arr.	Slack Trip	# Rnd.	# Scan. Routes	# Relax. Footpaths	# Jn.	Time [ms] Avg.	Time [ms] Sd.	Quality Top 1	Quality Top 5
RAP [7]	•	•	○	○	—	—	14.8	12,567	325,895	2.0	17.9	8.8	—	—
MRAP [7]	•	•	•	○	—	—	17.9	30,834	1,409,157	29.7	578.9	313.7	100.0 %	100.0 %
Self-BMRAP	•	•	•	○	30	—	17.6	21,701	716,889	10.6	230.5	185.4	100.0 %	97.6 %
Self-BMRAP	•	•	•	○	60	—	17.8	23,901	865,748	14.7	287.9	207.1	100.0 %	98.3 %
Target-BMRAP	•	•	•	○	30	—	32.4	33,051	1,327,054	8.7	268.1	192.3	100.0 %	93.3 %
Target-BMRAP	•	•	•	○	60	—	32.6	35,726	1,442,082	13.4	330.9	214.0	100.0 %	95.3 %
Tight-BMRAP	•	•	•	○	30	1	46.6	18,658	427,169	6.6	25.4	9.7	99.7 %	92.1 %
Tight-BMRAP	•	•	•	○	30	2	50.4	20,131	462,299	7.9	29.9	11.9	100.0 %	93.1 %
Tight-BMRAP	•	•	•	○	60	1	47.1	23,172	522,098	8.9	36.1	17.7	99.7 %	94.0 %
Tight-BMRAP	•	•	•	○	60	2	50.9	26,019	607,266	11.2	48.4	23.2	100.0 %	95.2 %
MRAP [7]	•	•	•	•	—	—	18.0	34,080	1,513,277	48.6	1,690.3	1,251.6	100.0 %	100.0 %
Self-BMRAP	•	•	•	•	30	—	17.6	23,209	736,155	15.2	459.1	447.4	100.0 %	97.8 %
Self-BMRAP	•	•	•	•	60	—	17.9	26,327	911,069	21.8	629.9	548.3	100.0 %	98.5 %
Target-BMRAP	•	•	•	•	30	—	32.4	34,379	1,380,531	12.2	541.1	474.0	100.0 %	93.4 %
Target-BMRAP	•	•	•	•	60	—	32.7	38,102	1,527,425	19.6	731.6	579.7	100.0 %	95.5 %
Tight-BMRAP	•	•	•	•	30	1	46.6	18,661	427,191	9.0	26.2	10.7	99.7 %	92.3 %
Tight-BMRAP	•	•	•	•	30	2	50.4	20,138	462,384	10.9	32.0	14.2	100.0 %	93.2 %
Tight-BMRAP	•	•	•	•	60	1	47.1	23,181	522,226	12.8	39.8	25.5	99.7 %	94.3 %
Tight-BMRAP	•	•	•	•	60	2	51.0	26,044	607,654	16.4	58.6	36.7	100.0 %	95.3 %

5.2 Performance of Bounded McRAPTOR. To evaluate the performance of our algorithms, we again first focus on our largest instance, Paris. (Figures for other instances are shown later.) Table 3 presents figures for RAP, MRAP, Self-BMRAP, Target-BMRAP, and Tight-BMRAP. The first columns indicate the criteria optimized by each algorithm, denoted by the •-symbol. These are arrival time (Arr.), number of trips (Trp.), walking duration (Wlk.), and number of bus trips (Bus).

As concluded in Section 5.1, we evaluate Self-BMRAP, Target-BMRAP, and Tight-BMRAP for arrival time slack values of 30 min and 60 min, and trip slack values of 1 and 2 (where applicable).

The table reports the number of rounds (Rnd.), the number of scanned routes, the number of relaxed footpaths, the number of computed journeys (Jn.). Note that for our multi-stage approaches, these values are sums over all RAP and MRAP executions. The table also reports the average (Avg.) running time and its standard deviation (Sd.), as well as the quality value regarding the top and the top five journeys.

We observe that RAP, which only considers arrival time and number of trips, is the fastest algorithm, taking 17.9 ms on average. However, it typically only outputs 2 journeys, which severely limits the presented choices to the user. Using MRAP to add further criteria to the Pareto set, increases the number of journeys to 29.7 (adding walking) and 48.6 (also adding buses). However, this comes with a substantial performance cost of MRAP taking 0.6 sec and 1.7 sec on average, respectively. This clearly motivates the computation of restricted Pareto sets (cf. Sections 3 and 4).

Our simplest approach, Self-BMRAP (which uses the earliest arrival times at the target for pruning), helps to bring the running time down by a factor between 2 and 3.6, depending on the considered criteria and slack values. Note that these factors closely correspond to the reduction in the number of computed journeys.

Recall that Self-BMRAP can neither handle trip slacks nor does it compute the restricted Pareto set exactly (cf. Section 4.1). The latter is overcome by Target-BMRAP, which runs a RAP stage to obtain the precise anchor journeys, which are then input to the following MRAP stage (cf. Section 4.2). While this results in the precise restricted Pareto set (albeit still not supporting trip slacks), the time of running the additional RAP stage is not offset by the time savings due to computing fewer journeys. As a result, Target-BMRAP is between 15 and 18 % slower than Self-BMRAP, which does not pay off.

Finally, we evaluate Tight-BMRAP, our three stage algorithm that uses a sophisticated pruning scheme,

Table 4: Query performance on several metropolitan public transit networks for all considered algorithms (except Self-BMRAP, which does not pay off; cf. Table 3). For each algorithm and instance, we report the average running time in milliseconds as well as the quality value of the top five journeys (where applicable). RAP optimizes arrival time and number of trips. All other algorithms also optimize walking duration and the number of buses, where indicated (Bus). The slack values for Self-BMRAP and Tight-BMRAP are 30 min arrival time and two trips.

		Berlin (VBB)		Budapest		Melbourne		New York City		Prague		Rome	
Algorithm	Bus	Time [ms]	Qual. Top 5	Time [ms]	Qual. Top 5	Time [ms]	Qual. Top 5	Time [ms]	Qual. Top 5	Time [ms]	Qual. Top 5	Time [ms]	Qual. Top 5
RAP [7]	○	12.9	—	10.2	—	11.1	—	11.2	—	2.8	—	7.2	—
MRAP [7]	○	66.1	100.0 %	235.3	100.0 %	323.0	100.0 %	544.8	100.0 %	53.7	100.0 %	171.8	100.0 %
Self-BMRAP	○	51.0	99.4 %	117.2	98.6 %	144.3	98.2 %	222.3	99.2 %	27.4	98.5 %	118.6	99.2 %
Tight-BMRAP	○	19.4	97.1 %	30.8	95.7 %	17.8	95.7 %	27.6	97.3 %	7.0	96.1 %	24.5	96.6 %
MRAP [7]	•	105.5	100.0 %	987.5	100.0 %	910.9	100.0 %	1,459.6	100.0 %	220.4	100.0 %	675.6	100.0 %
Self-BMRAP	•	71.1	99.3 %	257.7	98.8 %	280.6	98.2 %	437.9	99.2 %	58.4	98.3 %	280.5	99.2 %
Tight-BMRAP	•	19.9	96.8 %	37.5	96.0 %	18.7	95.7 %	30.8	97.4 %	7.8	95.8 %	30.2	96.5 %

resulting in tight bounds at every stop and round. It supports both arrival time and trip slacks, and computes the restricted Pareto set exactly. Having tight bounds available, dramatically accelerates the algorithm. Depending on the slack values, the running time is a factor of 12 to 23 faster than MRAP when optimizing walking duration. When additionally considering the number of buses as criterion, the speedup increases to a factor of 29 to 65. As a result, Tight-BMRAP computes a four-criteria restricted Pareto set of reasonable slack values (30 min for arrival time and two trips) in 32 ms.

We like to highlight that this is within a factor of 2 of running a single RAP execution that only optimizes two criteria. When taking into account that according to [7], RAP is about a factor of two faster than a single-criteria query using Dijkstra's algorithm, our experiments indicate that with our three stage Tight-BMRAP algorithm, optimizing a four-criteria restricted Pareto set can be done in about the same time as computing the earliest arrival time with Dijkstra.

At the same time, the quality of the computed journeys does not degrade significantly: the top journey has always a value (close to) 100 %, and the top five journeys have values above 92 % in all considered scenarios.

5.3 Other Instances. Our final experiment, reported in Table 4, evaluates our algorithms on additional metropolitan instances. For that we consider the complete public transit networks of the greater Berlin region (VBB), Budapest, Melbourne, New York City, Prague, and Rome. Recall that size figures for all networks are available from Table 1.

As concluded by the previous experiments, slack values resulting in a reasonable tradeoff between quality and running time are 30 min for arrival time and two trips. We therefore solely focus on these values in Table 4. For each instance and algorithm, we report the running time in milliseconds and the quality value of the top five journeys. Note that we omit reporting figures for Target-BMRAP, which seems to compute fewer journeys than Self-BMRAP, while being slower at the same time (cf. Table 3).

All reported algorithms optimize arrival time and number of trips. MRAP, Self-BMRAP, and Tight-BMRAP also optimize walking duration and potentially the number of buses. The latter is indicated in the table by the bus column with the •-symbol.

We generally observe a similar pattern as in the previous experiment on Paris. RAP is the fastest algorithm, optimizing the fewest criteria resulting in the fewest journeys (the latter is not reported in the table). Optimizing more criteria with MRAP comes with a huge running time impact—the largest in New York City, where MRAP (optimizing all four criteria) takes 1.5 sec, which is a factor of 130 slower than RAP.

Again, using our simplest of the new algorithms, Self-BMRAP, helps to improve the running time by small factors. Here, the largest improvement that we observe is in Melbourne, where the running time of Self-BMRAP using all four criteria is 280.6 ms compared to 910 ms of MRAP—a factor of 3.8. On our slowest instance, New York City, Self-BMRAP achieves 437 ms on average (again using all criteria), compared to 1.5 sec for MRAP.

When switching to our three phase Tight-BMRAP algorithm, which exactly computes the restricted Pareto set and handles arrival time and trip slacks, the running

times can be additionally reduced by significant amounts. Computing three-criteria restricted Pareto sets with Tight-BMRAP takes less than 31 ms on all instances—the slowest one being Budapest. We also observe that the running time impact of adding adding another criterion (number of buses) is marginal with Tight-BMRAP. Its slowdown is below 24 % on all instances, whereas the running time of MRAP may increase by a factor of four in the same scenario. For the case of optimizing all four criteria, the running time of Tight-BMRAP thus stays below 38 ms on all instances.

Examining the quality, we observe that the quality value of the top five journeys remains above 95 % on all instances in both optimization scenarios (above 93 % when including Paris).

From these experiments we conclude that Tight-BMRAP is a fast approach to computing multi-criteria restricted Pareto sets that retaining almost all important journeys of the full Pareto set. Moreover, it scales very well when additional optimization criteria are added.

6 Conclusion

In this work we presented a new efficient approach to computing multi-criteria transit journeys. By introducing the notion of arrival time and trip slacks, we obtained a sound definition of a restricted Pareto set—a subset of the entire Pareto set that we have shown to retain the most important journeys without containing undesirable outliers. To compute the restricted Pareto sets, we developed *Bounded McRAPTOR* (BMRAP), an extension of the well-known McRAPTOR algorithm. Its fastest variant, called Tight-BMRAP, is a sophisticated three-stage pruning scheme that computes the restricted Pareto set precisely and efficiently.

Our experiments on large realistic metropolitan transit networks have shown that Tight-BMRAP on four criteria (arrival time, number of trips, walking duration, and number of buses) is up to 65 times faster than full McRAPTOR. In fact, for reasonable slack values, running times of Tight-BMRAP remain below 38 ms on all evaluated instances, which is within a small factor of running a simple RAPTOR query. This, for the first time, enables four-criteria Pareto-optimal journey planning in interactive scenarios. Moreover, this is achieved without any preprocessing, which allows to handle dynamic updates like delays and cancellations easily.

For future work we would like to evaluate the robustness of the quality of the restricted Pareto sets under varying parameters of the ranking algorithm. Also, in this work we focused on metropolitan-sized networks, and the arrival and trip slack values that we experimentally determined to yield good results are tailored to that scenario. We are therefore interested in applying our approach to the long-distance scenario as well. We imagine this may require setting the slack values adaptively based on the properties of the anchor journeys to yield good results. Finally, regarding our algorithms, we are interested in combining them with speedup techniques, such as HypRAPTOR [6], to further accelerate query times.

References

[1] 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 *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, volume 6346 of *Lecture Notes in Computer Science*, pages 290–301. Springer, 2010.

[2] H. Bast, D. Delling, A. V. Goldberg, M. Müller–Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

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

[4] D. Delling, J. Dibbelt, T. Pajor, D. Wagner, and R. F. Werneck. Computing multimodal journeys in practice. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

[5] D. Delling, J. Dibbelt, T. Pajor, and R. F. Werneck. Public transit labeling. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*, Lecture Notes in Computer Science, pages 273–285. Springer, 2015.

[6] D. Delling, J. Dibbelt, T. Pajor, and T. Zündorf. Faster transit routing by hyper partitioning. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'17)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:14, 2017.

[7] D. Delling, T. Pajor, and R. F. Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2015.

[8] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Connection scan algorithm. *ACM Journal of Experimental Algorithmics*, 23(1.7):1–56, October 2018.

[9] Y. Disser, M. Müller–Hannemann, and M. Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of

Lecture Notes in Computer Science, pages 347–361. Springer, June 2008.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness*. W. H. Freeman and Company, 1979.

[11] M. Müller–Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2007.

[12] M. Müller–Hannemann and K. Weihe. Pareto shortest paths is often feasible in practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[13] 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, 2008.

[14] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 967–982. ACM Press, 2015.

[15] S. Witt. Trip-based public transit routing. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, volume 9294 of *Lecture Notes in Computer Science*, pages 1025–1036. Springer, 2015. Accepted for publication.