Scalable Algorithms for Bicriterion Trip-Based Transit Routing

Prateek Agarwal and Tarun Rambha

Abstract—The paper introduces innovative techniques to enhance Trip-Based Transit Routing (TBTR), a popular bicriterion transit routing approach. Inspired by Hypergraph-based Round-based Public Transit Routing (HypRAPTOR), we present Hypergraph-based TBTR (HypTBTR), a partitioning variant aimed at improving TBTR's query times. However, this improvement in HypTBTR (and HypRAPTOR) comes with increased preprocessing requirements. To address this issue, we propose two novel techniques: a One-To-Many variant of TBTR and multilevel partitioning. Our One-To-Many algorithm efficiently handles profile queries, and integrating multilevel graph partitioning in both HypTBTR and HypRAPTOR significantly reduces preprocessing computations. Extensive experiments on large-scale GTFS datasets of Switzerland, Netherlands, and Sweden, showcase the efficacy of the proposed algorithms. Empirically, HypTBTR outperforms TBTR by 23–37%, the One-To-Many variant shows a speed-up of 90–95%, and multilevel partitioning reduces preprocessing calculations by up to 53%. These improvements have the potential to enhance user experiences in web and mobile navigation applications and expedite problems such as transit assignments that involve repeated calls to routing algorithms.

Index Terms—Transit routing, shortest paths, multi-criteria optimization, hypergraph partitioning.

I. INTRODUCTION

ROUTING in public transit networks is typically aided by mobile apps and backend algorithms. Ideally, these algorithms must be fast and provide details on optimal journeys between origins and destinations of passengers. Conventional approaches model the transit network as a time-expanded or time-dependent graph and run a variant of Dijkstra's algorithm. However, this method turns out to be too slow for large networks [5]. Further, transit users have greater sensitivity to objectives other than travel time, particularly the number of transfers. This makes the problem challenging since users look for journeys that weigh the trade-offs between transfers and travel time. The primary objective of the current study is to improve the efficiency and practicality of bi-objective Public Transit Routing (PTR) algorithms, such as Transfer-Based Transit Routing (TBTR), using the recent advances in graph partitioning techniques.

The underlying principle of partitioning-based methods is to divide the transit networks (either based on routes or stops) into subnetworks and precompute the optimal paths between the boundary nodes. For example, consider the transit network in the left panel of Figure 1, where each route is indicated by a different color. The panel on the right shows the network split into three subnetworks. For origin-destination pairs that lie completely within each subnetwork, we can use regular PTR algorithms on a smaller graph. However, to travel between these subnetworks, a passenger has to pass through the red boundary stops (also known as cutstops). Thus, we can accelerate routing queries by precomputing and storing the shortest paths between the red stops for all departure times. While analogous techniques have been extensively explored in road networks [16], [19], their adaptation to transit networks proves more challenging due to the dynamic nature of transit networks. Additionally, the potential of multilevel partitioning to enhance PTR algorithms, particularly for multi-criterion problems, remains largely unexplored.

An inherent challenge in incorporating partitioning methods into PTR stems from the absence of an efficient One-To-Many algorithm (i.e., an algorithm that computes the optimal journey between a single origin and multiple destinations) required for finding the optimal paths between a given set of cutstops. Existing studies such as [6] and [11] repeatedly apply a One-To-One algorithm (i.e., an algorithm that computes the optimal journey between a single pair of origin and destination) using Round-based Public Transit Routing (RAPTOR) or TBTR for all possible cutstop combinations. This repetitive process is a bottleneck during preprocessing and hinders the scalability and practicality of the partitioning-based approach. To address this, we propose a novel One-To-Many framework that can efficiently handle multi-criteria queries. Besides aiding preprocessing in PTR algorithms, our framework can also benefit several other practical problems like queries involving Points-of-Interest [14]; simulation-based transit assignment [3]; and building isochrones [7], i.e., a set of vertices reachable from a given point within a time or distance limit.

A. Relevant Studies and Research Gaps

Finding the “shortest/best” path efficiently is a widely researched problem in network science. In this section, studies relevant to the current work is summarized. For a more comprehensive review on shortest paths, refer [5].
Popular multi-criteria transit routing frameworks developed in the past decade include — RAPTOR [13], Transfer Patterns [4], and TBTR [36]. RAPTOR operates directly on the timetable to compute Pareto-optimal journeys. It is fully dynamic (no preprocessing) and has been extended to include multiple criteria (e.g., arrival time, transfer, and fare zones). In contrast, the Transfer Patterns algorithm also computes Pareto-optimal journeys but involves a substantial preprocessing step. It relies on the concept that optimal journeys for a origin-destination pair can be derived from a small, fixed subset of nodes, allowing a significant portion of the network to be ignored during the query stage. However, the preprocessing associated with the Transfer Patterns is much higher than its counterparts. TBTR strikes a balance between the extremes represented by RAPTOR and Transfer Patterns. It features a lightweight preprocessing, outperforming RAPTOR while avoiding the extensive preprocessing of Transfer Patterns.

More recent approaches for PTR algorithms explore partitioning-based methods to enhance query times (Scalable Transfer Patterns [6], Hyper-partitioned RAPTOR (HypRAPTOR) [11] and Accelerated Connection Scan Algorithm (ACSA) [15], [29]). Although the preprocessing times of Scalable Transfer Patterns are less than those of Transfer Patterns, they are still higher when compared with other PTR frameworks. Notably, a partitioning framework for trip-based algorithms such as TBTR had not been explored. Furthermore, the potential of multilevel partitioning in enhancing PTR algorithms, especially for multi-criterion problems, received little attention despite its widespread use in road routing. [16]. To the best of our knowledge, ACSA is the only algorithm (relevant to the current study) that exploits a multilevel paradigm. However, its application is limited to the earliest arrival problem, which is significantly easier to solve [30].

Addressing the challenges of integrating partitioning methods into PTR, [26] proposed a One-To-Many algorithm that proposed ULTRA-PHAST that combines UnLimited TRAnsfer technique with Parallel Hardware-Accelerated Shortest path Trees. However, the ULTRA-PHAST framework has not been applied for queries involving a range of departure times, which is required during preprocessing. While several extensions to road routing algorithms have been proposed to handle similar problems [12], [21], literature on One-To-Many PTR algorithms is sparse. Other directions of research that have been explored include online transit routing strategies with uncertainty in trip times [18], [25], uncertain demand [32], [33], and multi-modal PTR [9], [17]. In this paper, we focus on deterministic settings and bicriteria journeys using arrival times and the number of transfers.

Table I highlights the gaps in existing PTR frameworks and the features of the algorithms proposed in this paper. More specifically, the following contributions address these gaps.

- **Hyper-partitioned TBTR (HypTBTR):** Section III presents HypTBTR, a novel algorithm that combines TBTR with a partitioning-based speed-up method. Results demonstrate that HypTBTR significantly outperforms the TBTR algorithm, with a speedup of 23–37% in country-level datasets. In the context of transit routing, where thousands of concurrent queries are executed every second, even slight improvements in both web and mobile routing algorithms lead to a significant enhancement in user experience. These algorithms can quickly run the shortest paths on low-end mobile hardware or edge devices (such as kiosks in transit stations), enabling the development of practical tools like offline maps (which currently exist only for road networks on Google Maps).
- **One-To-Many rTBTR:** Profile queries are a major bottleneck in the preprocessing phase of advanced PTR algorithms. To address this problem, we present a new One-To-Many variant of the rTBTR algorithm in Section IV-A. While many transit routing algorithms can be customized to solve One-To-All queries, the preprocessing phase only requires optimal journeys to a small subset of destinations. We notice a 90–95% speed-up compared to existing approaches using our One-To-Many version.
- **Multilevel variants for HypTBTR and HypRAPTOR:** Reduced query times in HypTBTR (and HypRAPTOR) come at the cost of increased preprocessing which can hinder the practicality of these algorithms. We solve this issue in Section IV-B using multilevel partitioning. Our approach reduces preprocessing calculations by approximately 5–53% compared to standard partitioning. Notably, these advantages are achieved without compromising the quality of solutions or query times.

The rest of the paper is structured as follows. In Section II, we introduce terminology related to PTR, and review RAPTOR and TBTR. In Section III, we introduce the HypTBTR algorithm. Section IV is devoted to reducing preprocessing times using a One-To-Many rTBTR algorithm and multilevel partitioning. Section V compares the performance of the pro-

---

**Fig. 1.** Partitioning techniques used in PTR algorithms.
posed techniques. Finally, Section VI summarizes our findings and presents future research directions.

II. PRELIMINARIES

Terminology used in public transit networks are described in Subsection II-A and are summarized in Table II. Next, Subsection II-B reviews a few algorithms closest to our work.

A. Terminology

In the present context, the problem of journey planning in PTR can be divided into three categories (1) Earliest Arrival Problem: Find the quickest journey to destination (2) Multi-criterion Problem: Find a journey(s) based on multiple criteria such as transfers, wait and walk time, cost, and crowding, etc. The output of this problem is generally a Pareto-optimal set of journeys. Among the possible objective function combinations, the bicriterion problem with travel time and transfers is the most explored setting. (3) Range Problem: Find all optimal journeys departing within a specific time range (e.g., journeys departing between 0700–0730). If the time range is 24 hours, it is also called a Profile Query.

A timetable in a public transit network is defined as a tuple \((S, R, T, F)\), where \(S\) is a set of stops, \(R\) denotes the set of routes, \(T\) represents the set of trips, and \(F\) indicates a set of footpaths. The timetable corresponds to bus/train schedules and contains information related to a transit network and the movement of vehicles on routes. The most common format for representing timetable information is the General Transit Feed Specification (GTFS). GTFS is a standardized format that defines headers for multiple files and rules on how they are related to each other, facilitating the exchange of transit data.

A stop \(s \in S\) is a distinct location in the network where the passengers can board/alight vehicles. For example, bus or train platforms. The origin and destination stops are labeled \(s_o\) and \(s_d\), respectively. A route \(r \in R\) is a sequence of stops followed in a particular order, i.e., a predefined course/path of transit operation. The \(i^{th}\) stop on route \(r\) is denoted by \(r(i)\). The length of route \(r\), denoted by \(l_r\), is the total number of stops in the route.

A trip \(t \in T\) is associated with the movement of a vehicle along a route. A stop event (or simply an event) refers to a trip arrival or departure at a stop. Similar to \(r(i)\) and \(l_r\), we use \(t(i)\) and \(l_t\) to denote \(i^{th}\) stop and the length/number of stops in trip \(t\), respectively. A trip segment \(t(i \to j)\) is used to refer to a section of a trip \(t\) from stop index \(i\) to \(j\), that is from the \(i^{th}\) stop of the trip to the \(j^{th}\) stop. Route ID of trip \(t\) is indicated by \(route(t)\). The arrival and departure times at the \(i^{th}\) stop of trip \(t\) are denoted by \(arr(t, i)\) and \(dep(t, i)\), respectively. Unless stated otherwise, trips along a route are assumed to be non-overtaking, i.e., trips follow the First-In-First-Out (FIFO) property. Additionally, for two trips \(t_1\) and \(t_2\) operating on the same route, i.e., \(route(t_1) = route(t_2)\), the following notation is used to indicate precedence. If \(arr(t_1, i) < arr(t_2, i)\) \(\forall i = 1, \ldots, l_t\), \(t_1 < t_2\). Likewise, if \(arr(t_1, i) \leq arr(t_2, i)\) \(\forall i = 1, \ldots, l_t\), we express this scenario as \(t_1 \leq t_2\).

A footpath \((s_1, s_2) \in F\) (also sometimes called a transfer) is a pedestrian connection between stops \(s_1, s_2 \in S\). For a footpath \((s_1, s_2) \in F\), the time required to travel between stops \(s_1\) and \(s_2\) is denoted by \(f(s_1, s_2)\). We assume that footpaths are transitively closed, i.e., if \((s_1, s_2), (s_2, s_3) \in F\), then \((s_1, s_3) \in F\). Also, footpath times are assumed to follow the triangle inequality, i.e., \(f(s_1, s_3) \leq f(s_1, s_2) + f(s_2, s_3)\). Since in GTFS, the transfers.txt file that contains footpath details between stops is optional, most online sources do not provide it. Thus, to construct the set \(F\), we first define an upper limit on the walking time between a pair of stops. Additional footpaths are then added so that \(F\) is transitively closed. We also define the neighborhood of a stop \(s\), denoted by \(N(s)\), as \(N(s) = \{s' | (s, s') \in F\} \cup \{s\}\). Intuitively, for a stop \(s\), \(N(s)\) is a set of stops directly connected to \(s\) via footpaths in \(F\) and includes \(s\).

For processing shortest path queries efficiently, we need to precompute trip-transfers. A trip-transfer, denoted by \((i_1, i_2, j_1, j_2)\), represents a possible transfer from the \(i^{th}\) stop of trip \(t_1\) to the \(j^{th}\) stop of trip \(t_2\). Note that transfer and trip-transfer are different terms and are not interchangeable. While a transfer just refers to a footpath connection between two stops, a trip-transfer indicates switching between two trips. The switch is possible by either alighting the current trip and waiting at the same stop to board another trip or by walking to a different stop via a footpath and boarding a trip. The set of trip-transfers is denoted by \(J\). Refer to [36] for more details on computing \(J\).

A journey \(y\) is a sequence of trips and footpaths (in the order of traversal). To evaluate a journey \(y\), we define a vector \(g(y) = (g_1(y), g_2(y), \ldots, g_m(y))\), where each element of the right-hand side represents the attributes of various optimization criteria. A journey \(y_1\) dominates journey \(y_2\) if all the elements \(g(y_1)\) are no worse than that in \(g(y_2)\). For example, if the arrival time and the number of transfers are the optimization criteria, a journey \(y_1\) with \(g(y_1) = (1030, 1\) transfer\) dominates a journey \(y_2\) with \(g(y_2) = (1100, 2\) transfers\). The set of \(g(y)\) for different Pareto-optimal journeys is denoted by \(\mathcal{J}\). A set \(\mathcal{J}\) is Pareto-optimal if none of the journeys in \(\mathcal{J}\) are dominated by other feasible journeys. For example, \(\mathcal{J} = \{(1030, 1\) transfer\), \((1100, 0\) transfers\)\} is Pareto-optimal if other feasible journeys do not have earlier arrival times and fewer transfers. We use \(y\) and \(g(y)\) interchangeably to denote a journey when it is clear from the context. Some studies also consider the minimum change time and dwell time at each stop which is the extra time spent by a passenger while transferring between trips and the time duration for which a bus/train waits at a stop before departing, respectively. To keep the pseudocodes simple, both these parameters are assumed to be zero, but extending them to a realistic setting is straightforward.

B. RAPTOR and TBTR

For illustrating the algorithms discussed in this paper, we use the toy network shown in Figure 2. The colored solid edges indicate different routes. Let the travel time between any two stops directly connected by an edge be 10 minutes. The dashed line \(s_3\)–\(s_4\) represents a footpath with a travel time of 40 minutes. Let the origin, destination, and departure time
be \(s_o\), \(s_d\), and 0800, respectively. The table adjacent to the network shows the timetable. Each row represents a route (color-coded) and every cell indicates the trip ID and the trip-time at stop \(s\), which represents the best arrival time at \(s\) so far. (Note that unlike RAPTOR, the TBTR algorithm and the rest of the paper assume that \(\tau_{\text{opt}}(n, s)\) represents the journey times using at most \(n\) transfers.) Each round is divided into two phases: a route phase and a transfer phase. In the route phase, all routes serving a set of marked stops (origin stop or stops whose label improved in the previous round) are collected. For each of these routes, the earliest possible trip that can be boarded from its marked stop is determined, and its subsequent stop labels are improved (if possible). If the label of a stop is updated, it is added to the marked stop list. In the transfer phase, footpath connections of stops in the marked stop list are evaluated. Stops whose arrival time improves with these footpath connections are added to the marked stop list. RAPTOR has also been modified to handle range queries (rRAPTOR) and multi-criteria queries (McRAPTOR).

The following example illustrates the RAPTOR algorithm.

In Figure 2, in Round 0, the label \(\tau_{\text{opt}}(0, s_o)\) is updated to the departure time \(\tau\) and \(s_o\) is added to the marked stop list. Thus, the first round involves boarding the earliest possible trips on the pink (\(r_1\)) and green (\(r_2\)) routes at 0800 and updating its subsequent stop labels. Stops whose labels are updated (i.e., \(s_2\), \(s_3\), \(s_5\), and \(s_8\)), are added to the marked stop list. In the transfer phase of Round 1, footpath connections are evaluated, and the label of stop \(s_d\) is updated to 0900, and \(s_d\) is added to the marked stop list. Next, Round 2 begins its route phase using the marked stop list from the transfer phase of Round 1. The algorithm proceeds similarly and terminates when the maximum transfer limit is reached or no stop labels are updated. For the current example, Round 1 explores the pink route (\(r_1\)), green route (\(r_2\)), and the footpath; Round 2 scans the grey (\(r_3\)) and orange route (\(r_4\)); and finally, Round 3 explores the red route (\(r_5\)) and terminates. The final Pareto-optimal journeys are (0900, 0 transfers) and (0850, 2 transfers).

2) TBTR: TBTR [36] solves the bicriteria (travel time and transfer) optimization problem using trips and trip-transfers as building blocks. The core idea in TBTR is similar to a breadth-first search on a graph in which trips are modeled as nodes and trip-transfers are edges. The first level can be imagined to represent the earliest trips that can be boarded from the origin stop, and each subsequent level corresponds to a transfer. TBTR’s search pattern is similar to RAPTOR, but unlike RAPTOR, it does not maintain multi-labels for all stops. The preprocessing phase and query phase of TBTR are described below.

a) Preprocessing phase: TBTR has a two-stage preprocessing phase. Using the GTFS set, the first stage collects all possible trip-transfers in a set \(T\) (referred to as the Trip-transfers set). This may result in a large collection of trip-transfers, most of which are never part of optimal journeys for any origin-destination pair. Hence, the second stage aims to reduce \(T\) to only contain useful trip-transfers [36].
b) Query phase: Given an input (GTFS, $T$, $s_o$, $s_d$, $\tau$, $\lambda$), the query stage maintains a set $\mathcal{J}$ of Pareto-optimal tuples of the form (arrival time, number of transfers) and a queue $Q_n$ of trips-segments to be scanned for each allowed number of transfers $n$, where $n \in \{0, 1, \ldots, \lambda\}$ and $\lambda$ is the maximum transfer limit. $Q_0$ is initialized with the earliest trip-segments that can be boarded on all routes passing through stops in $\mathcal{N}(s_o)$. Additionally, for every trip $t$, we maintain a variable, $\text{ind}(t)$, which denotes the index of the first stop of trip $t$ that can be reached. To efficiently check if a trip passes through $s_d$, the algorithm initializes a set $L$ which keeps track of all routes that pass through at least one of the stops in $\mathcal{N}(s_d)$. The best-known arrival time at the destination stop $s_d$ using at most $n$ transfers is denoted by $\tau_{\text{opt}}(n)$. TBTR, like RAPTOR, also works in rounds. In the $n^{th}$ round, trip-segments in the queue $Q_n$ are scanned. While scanning a trip-segment $t(h \rightarrow k)$ in round $n$, the following operations are performed:

- Using $L$, we check if the trip $t$ passes through $s_d$ (or stops in its neighborhood) and results in a non-dominated label. If so, $\tau_{\text{opt}}(m)$ is updated $\forall m = n, \ldots, \lambda$. Reducing the labels for all subsequent rounds ensures that the optimality check in later rounds is done against the destination’s best arrival time.
- Using the trip-transfers $(t, i, i', j)$ available from the $i^{th}$ stop of $t$ within a trip-segment $t(h \rightarrow k)$, we add trip-segments $t'(j \rightarrow \text{ind}(t'))$ to the queue $Q_{n+1}$ if the arrival time at the $(h+1)^{th}$ stop on $t$ is less than the destination’s best-known arrival time (i.e., $\text{arr}(t, h+1) < \tau_{\text{opt}}(n)$) and the trip $t'$ from $j$ onwards has not already been explored (i.e., $j < \text{ind}(t')$).

The algorithm stops when the maximum transfer limit $\lambda$ is reached, i.e., $n = \lambda$ or when $Q_0$ is empty for $n < \lambda$. The pseudocode for the TBTR’s bicriterion query is shown in Algorithm 1. Lines 1–5 represent the initialization phase where $\tau_{\text{opt}}(n)$ is initialized to $\infty$ for all $n \leq \lambda$. Lines 6–8 generate the set $L$, which is used during the query phase to check if a journey reached the destination stop. Lines 9–14 add the trip-segments that can be boarded from stops in $\mathcal{N}(s_o)$ to $Q_0$. Lines 15–26 scan every trip-segment $t(h \rightarrow k)$ as described above. Connections from the trip-segment (assuming the condition in Line 21 is satisfied) are collected in a list $\text{clist}$. Next, Line 24 calls the function $\text{ENQUEUE}$, which iterates over the elements of $\text{clist}$ and adds unexplored trip-segments to $Q_{n+1}$. Figure 3 illustrates the main while-loop of the TBTR algorithm.

Consider the example from Figure 2 to illustrate the TBTR algorithm and assume $T = \{(t_1, 2, f_8, 1), (t_2, 3, t_{14}, 1), (t_{14}, 2, t_{20}, 1), (f_8, 3, t_{20}, 1)\}$. Note that this is the smallest $T$ required to solve the example query. Round 0 starts by scanning the trip-segments in queue $Q_0 = \{t_1(1 \rightarrow 3), t_2(1 \rightarrow 3)\}$. Since trip $t_1$ is connected to $s_d$ via a footpath, $\tau_{\text{opt}}(0)$ is updated to 0900. Next, we scan $T$ to get trip-transfers from trips $t_1$ and $t_2$ and add the corresponding trip-segments to queue $Q_2$. Round 1 starts with $Q_1 = \{t_8(1 \rightarrow 3), t_{14}(1 \rightarrow 2)\}$ and proceeds in a similar fashion. For Round 2, $Q_3 = \{t_{20}(1 \rightarrow 3)\}$. The final output of the algorithm is $\tau_{\text{opt}}(0) = 0900$, $\tau_{\text{opt}}(1) = 0900$, and $\tau_{\text{opt}}(2) = 0850$.

Similar to rRAPTOR, TBTR can also handle range queries. The idea behind this algorithm is to run the main query for different departure times while preserving labels between runs. Later departure times are processed first. Another version of TBTR, Trip-based Routing using Condensed Search Trees or TBTR-CT [37], exploits the observation that the optimal journey for an origin-destination pair for any departure time can be constructed using only a fixed set of routes instead of the complete network. It precomputes routes for each origin-destination pair, and thus, the query phase explores a much smaller graph, resulting in faster queries. Like Transfer Patterns [4], faster query times in TBTR-CT come at the cost of high preprocessing time and increased memory usage.

Reference [24] proposed two new variants of TBTR: the Walking TBTR, which additionally minimizes the walking time along a journey, and Fare Zone TBTR, which optimizes fares as a third criterion. Their results have been compared with McRAPTOR. Reference [27] extended TBTR to handle multimodal bicriteria routing problems by combining it with ULTRA [8].

III. REDUCING QUERY TIMES

This section integrates partitioning-based methodologies in the TBTR environment and presents a new faster bi-criteria algorithm HypTBTR. Subsection III-A starts with the relevant background, followed by the development of HypTBTR in Subsection III-B.

A. Background

Reference [11] proposed HypRAPTOR for faster queries by introducing a preprocessing phase that partitions routes into $p$ disjoint sets $R_1, R_2, \ldots, R_p$, also known as route cells. The central idea is to construct a hypergraph $G$ in which nodes represent routes, and hyperedges between subsets of nodes represent intersecting routes. Two routes intersect if they have at least one stop in common. Footpaths are also treated as routes with two stops. A partitioning algorithm (based on a min-cut approach) is then used to generate route cells. By definition, route cells are mutually exclusive and exhaustive (i.e., $R_i \cap R_j = \emptyset$ for all $i$ and $j$, and $\bigcup_{i=1}^{p} R_i = R$).

In the partitioning algorithm, suitable edge and node weights can be used to influence the partitions. Multi-edges resulting from routes intersecting at more than one stop are reduced to a single edge by summing their weights. Every boundary edge of the partition (an edge with nodes in different cells) represents a stop in the original transit network and is referred to as a cutstop. Consider the example in Figure 4. Subfigure (b) shows the corresponding hypergraph with each route as
Algorithm 1: TBTR: Bicriterion Query (GTFS, \( T \), \( s_o \), \( s_d \), \( \tau \), \( \lambda \))

1: \( n \leftarrow 0 \)
2: \( J, L \leftarrow \emptyset \)
3: \( \tau_{opt}(n) \leftarrow \infty \) for all \( n = 0, 1, \ldots, \lambda \)
4: \( Q_n \leftarrow \emptyset \) for all \( n = 0, 1, \ldots, \lambda \)
5: \( \text{ind}(t) \leftarrow l_t \) for all \( t \in T \)
6: for \( s \in \mathcal{N}(s_o) \) do
7: for \( (r, i) \) such that \( r(i) = s \) do
8: \( L \leftarrow L \cup \{(r, i, f(s, s_d))\} \)
9: \( \text{clist} \leftarrow \emptyset \)
10: for \( s \in \mathcal{N}(s_o) \) do
11: for \( (r, i) \) such that \( r(i) = s \) do
12: \( t \leftarrow \) earliest trip on \( r \) such that \( \tau + f(s_o, s) \leq \text{dep}(t, i) \)
13: \( \text{clist} \leftarrow \text{clist} \cup \{(t, i, n)\} \)
14: ENQUEUE(\( \text{clist, ind, } Q_n \))
15: while \( Q_n \neq \emptyset \) and \( n \leq \lambda \) do
16: for \( r(h \rightarrow k) \in Q_n \) do
17: for \( (\text{route}(t), i, \Delta \tau) \in L \) such that \( h < i \leq k \) and \( \text{arr}(t, i) + \Delta \tau < \tau_{opt}(n) \) do
18: \( \tau_{opt}(m) \leftarrow \text{arr}(t, i) + \Delta \tau \)
20: \( \text{clist} \leftarrow \emptyset \)
21: if \( \text{arr}(t, h + 1) < \tau_{opt}(n) \) then
22: for \( (i, i', j) \in T \) such that \( h < i \leq k \) do
23: add \( (i', j, n + 1) \) to \( \text{clist} \) if not already present
24: ENQUEUE(\( \text{clist, ind, } Q_{n+1} \))
25: add \( (\tau_{opt}(n), n) \) to \( J \) if it is non-dominated
26: \( n \leftarrow n + 1 \)
1: procedure ENQUEUE(list \( \text{clist, vector } \text{ind, queue } Q )\)
2: for \( (t, i, n) \in \text{clist} \) do
3: if \( i < \text{ind}(t) \) then
4: \( Q \leftarrow Q \cup \{t(i \rightarrow \text{ind}(t))\} \)
5: for trip \( t' \) such that \( i \leq t' \text{androute}(t) = \text{route}(t') \) do
6: \( \text{ind}(t') \leftarrow \min \{\text{ind}(t'), i\} \)

A node. A hyperedge with three nodes is added between the red, orange, and grey routes. The black node represents the footpath. Assuming \( p = 3 \), a partitioning algorithm creates three route cells \( R_1 \) (dark blue), \( R_2 \) (cyan), and \( R_3 \) (purple), as shown in Subfigure (c). Subfigure (d) shows the cutstops derived from the route cells in (c).

Similar to route cells, stops cells \( S_0, S_1, \ldots, S_p \) is a partition of stops. To construct a stop cell \( S_i \), we start by including all the stops belonging to the routes in the corresponding route cell \( R_i \). However, the resulting stop cells are not mutually exclusive due to cutstops. Thus, define \( S_0 \) to contain all the cutstops and update stop cell \( S_i \) as \( S_i \leftarrow S_i \setminus S_0 \). For example, in Figure 2, the stop cells are \( S_0 = \{s_0, s_2, s_3, s_5\} \), \( S_1 = \emptyset \), \( S_2 = \{s_5, s_6\} \), and \( S_3 = \{s_3, s_4, s_7\} \). Note that for \( p \) partitions, there are \( p \) route cells and \( p + 1 \) stop cells. The next step is to find and store optimal routes between these cutstops, referred to as fill-in. To this end, a profile query (using rRAPTOR) is run for all possible pairs of cutstops in \( S_o \). If a route belongs to an optimal journey between a pair of cutstops, it is added to the fill-in set.

In the query phase, given the origin and destination, the first step is to identify \( S_o \) and \( S_d \), the stop cells to which \( s_o \) and \( s_d \) belong to, respectively. Let the corresponding route cells be \( R_o \) and \( R_d \). If the origin stop is a cutstop, then \( S_o \) and \( R_o \) are set to \( S_o \) and \( \emptyset \), respectively. The same convention is used for the destination stop. The algorithm scans a route only if it belongs to the fill-in set or if all its stops are in the origin or destination cells. Building on this idea, we construct the HypTBTR algorithm. Changes to the query phase and hypergraph weighting scheme are suggested for the trip-based setting. In a subsequent section, we discuss methods to make HypTBTR preprocessing more efficient.

Outputs from partitioning-based PTR algorithms are known to miss Pareto-optimal solutions occasionally. To see why, consider the two-route example in Figure 5, which is similar to that discussed in [6]. Let the travel time on all links (except \( s_2+s_3 \)) be 10 minutes. Suppose the travel time on segment \( s_2+s_3 \) along route \( r_1 \) and \( r_2 \) is 30 and 40 minutes, respectively. Assume that trips run on both routes at a 10-minute frequency starting from 0800. For a traveler departing at 0800, the Pareto-optimal journeys from \( s_o \) to \( s_d \) according to RAPTOR are \((0900, 0 \text{ transfers})\) and \((0850, 2 \text{ transfers})\). In the partitioning-version, let \( R_1 = \{r_1\} \) and \( R_2 = \{r_2\} \), which implies \( S_0 = \{s_2, s_3\} \). Thus, the fill-in set only contains route \( r_1 \). Since the query phase of HypRAPTOR is restricted to using trips from the fill-in, the output contains only one journey \((0850, 2 \text{ transfers})\).

To address this issue, [6] suggested extending the set of cutstops by including stops that are connected to the original cutstops via some trip. For the example in Figure 5, setting \( S_0 = \{s_2, s_3\} \cup \{s_0, s_1, s_2, s_4\} \) will help discover all Pareto-optimal journeys. However, this increases the size of \( S_0 \), resulting in a larger fill-in set and, consequently, slower query times. Notice that route \( r_2 \) could be in the fill-in if the travel times and departure times were different. Empirically, we rarely found the Pareto-optimal sets of the partitioning and the non-partitioning versions to be different and hence we chose to stick to the definition of cutstops proposed in [11].

B. HypTBTR

1) Preprocessing Phase: Preprocessing in HypTBTR can be grouped into two phases: trip-transfer phase and fill-in computation phase. The trip-transfer phase generates a trip-transfer set \( T \) and is similar to TBTR's preprocessing.

The fill-in computation phase can be further divided into two steps. The first step is similar to HypRAPTOR and generates a hypergraph by modeling each route as a node. Hyperedges between routes that have common stops are then added. To generate the partitions and cutstops, we use the state-of-the-art KahyPar algorithm [28]. KahyPar works on a min-cut principle and finds nearly equal-sized cells such that the total weight of hyperedges in the cut is minimized. The size of a cell is defined as the sum of the weights of nodes belonging to it. The nodes (i.e., routes in the original transit network)
are weighted by the total number of events in the route. The weights of nodes corresponding to footpath connections are set to zero. The weight of a hyperedge corresponding to a stop \( s \) is set to the logarithm of the sum of events associated with all stops in \( \mathcal{N}(s) \).

For a pre-determined partition size \( p \), the output of the first step includes \( p \) route cells \( R_1, R_2, \ldots, R_p \) and \( p + 1 \) stop cells \( S_0, S_1, \ldots, S_p \), where \( S_0 \) is the set of cutstops. The second step in the fill-in computation phase is to find the fill-in trip set \( \mathcal{F} \), i.e., the set of trips required for all optimal journeys between the cutstops. For this purpose, [11] uses a profile query for all possible cutstop permutations by repeatedly applying rRAPTOR. To speed-up this process, we propose a new accelerated version of rTBTR, a One-To-Many rTBTR in Section IV-A. Our implementation significantly outperforms the existing approach.

2) Query Phase: Depending on the fill-in representation, the query phase can be implemented in multiple ways. The simplest approach is to scan a route only if it belongs to either the origin or destination cell, or is a part of the fill-in. Other methods include marking every event that is a part of fill-in computations instead of marking the whole route or generating a new overlay graph by copying the events of the origin and destination cells and fill-in.

These methods present a trade-off between speed and memory usage. Experiments by [11] show that overlay graphs are slightly better, but other methods have comparable performance and the exact benefits depend on the network and partition structure. In this paper, we use the simplest form of fill-in representation, i.e., for every trip \( t \), we initialize a one-bit variable known as a trip flag denoted by \( \text{flag}(t) \).

A trip's flag is true if it belongs to the origin or destination cell or is a part of the fill-in. The pseudocode for the query phase is described in Algorithm 2.

Given \( s_o \) and \( s_d \), we first identify \( S_o \) and \( S_d \), the stop cells to which the origin and destination stop belong, respectively. The corresponding route cells are identified as \( R_o \) and \( R_d \). Lines 1–5 are used to initialize the variables and are similar to Algorithm 1. Lines 6–8 sets the trip flags for every trip by checking if it belongs to the fill-in set or its route belongs to \( R_o \) or \( R_d \). Lines 9–11 define the set \( \mathcal{L} \), which stores information on all the routes and footpaths leading to the destination stop. Lines 12–17 consider the first trip on different routes that can be bored from \( \mathcal{N}(s_o) \) and adds them to \( Q_0 \). Lines 18–29 contain the main iterations of the algorithm. Trip segments are added to a queue and are scanned as before with an extra condition that checks if the \( \text{flag}(t) \) variable is True.

a) Proof of correctness: Suppose for a given fill-in set \( \mathcal{F} \), there exists an optimal journey \( y \) between \( s_o \) and \( s_d \) whose earliest arrival time and number of transfers are not captured by Algorithm 2. The following cases can arise: (i) \( S_o = S_d = \emptyset \), i.e., \( s_o \) and \( s_d \) are cutstops, (ii) \( S_o = S_d (\neq \emptyset) \), i.e., \( s_o \) and \( s_d \) belong to the same stop cell, or (iii) \( S_o \neq S_d \), i.e., \( s_o \) and \( s_d \) belong to different stop cells.

In Case (i), as both \( s_o \) and \( s_d \) are cutstops, HypTBTR will only scan trips in the fill-in set \( \mathcal{F} \) (since Lines 6–8 set \( \text{flag}(t) = \text{True} \) \( \forall t \in \mathcal{F} \)). Thus, if \( y \) is optimal, the fact that \( \mathcal{F} \) contains all the optimal trips between the cutstops is contradicted. In Case (ii), optimal journeys are either fully contained in the route cell or exit the route cell at some cutstop \( s_i \) and enter again at another cutstop \( s_j \). The former scenario reduces to a simple TBTR query without partitioning and in the latter instance, all the trips from the route cell and \( \mathcal{F} \) are scanned in Lines 6–8 and hence it again contradicts the definition of \( \mathcal{F} \). The arguments for Case (iii) are similar. Further, note that Line 28 restricts \( \mathcal{F} \) to only contain Pareto-optimal labels and hence it does not contain any extra labels corresponding to sub-optimal journeys.

IV. REDUCING PREPROCESSING TIMES

In this section, we speed-up preprocessing using a One-To-Many rTBTR which reduces the time required for profile queries (Section IV-A) and multilevel partitioning which reduces the calculations required for the fill-in set computation (Section IV-B).

A. One-To-Many rTBTR

As discussed in Section III-B.1, preprocessing involves solving profile queries between all possible cutstop pairs. Existing approaches use a range algorithm (rRAPTOR or rTBTR). This section proposes a One-To-Many version to make this step efficient.

For a give origin stop \( s_o \), let \( tlist \) represent a list of all departure times of trips (sorted in descending order) from \( s_o \).
Algorithm 2 HypTBTR: Bicriterion Query (GTFS, $T, s_o, s_d, \tau, F, \lambda, R_o, R_d$)

1: $n \leftarrow 0$
2: $t_{opt}(n) \leftarrow \infty$ for all $n = 0, 1, \ldots, \lambda$
3: $\mathcal{L}, \mathcal{J} \leftarrow \emptyset, \emptyset$
4: $Q_n \leftarrow \emptyset$ for all $n = 0, 1, \ldots, \lambda$
5: $ind(t), flag(t) \leftarrow \emptyset$, False for all $t \in T$
6: for $t \in T$ do
7: if $t \in \mathcal{F}$ or $\text{route}(t) \in R_o \cup R_d$ then
8: $flag(t) \leftarrow \text{True}$
9: for $s \in \mathcal{N}(s_d)$ do
10: for $(r, i)$ such that $r(i) = s$ do
11: $\mathcal{L} \leftarrow \mathcal{L} \cup\{(r, i, f(s, s_d))\}$
12: $clist \leftarrow \emptyset$
13: for $s \in \mathcal{N}(s_o)$ do
14: for $(r, i)$ such that $r(i) = s$ do
15: $t \leftarrow \text{earliest trip on } r \text{ such that } \tau + f(s, s) \leq \text{dep}(t(i))$
16: $clist \leftarrow clist \cup \{(t, i, n)\}$
17: ENQUEUE(clist, ind, $Q_n$)
18: while $Q_n \neq \emptyset$ and $n \leq \lambda$ do
19: for $r(h \rightarrow k) \in Q_n$ do
20: for $(\text{route}(t), i, \Delta \tau) \in \mathcal{L}$ such that $h < i \leq k$ and
21: $\text{arr}(t, i) + \Delta \tau < t_{opt}(n)$ do
22: for $m = n, n + 1, \ldots, \lambda$ do
23: $\tau_{opt}(m) \leftarrow \text{arr}(t, i) + \Delta \tau$
24: $clist \leftarrow \emptyset$
25: if $\text{arr}(t, h + 1) < t_{opt}(n)$ then
26: for $(t', j, n + 1)$ do
27: $\text{enqueue}(clist, ind, Q_{n+1})$
28: $n \leftarrow n + 1$
1: procedure ENQUEUE(list clist, vector ind, queue $Q$)
2: for $(t, i, n)$ in clist do
3: if flag(t) and $i < \text{ind}(t)$ then
4: $Q \leftarrow Q \cup\{(i \rightarrow \text{ind}(t))\}$
5: for trip $t'$ such that $t \preceq t'$ and $\text{route}(t') = \text{route}(t)$ do
6: $\text{ind}(t') \leftarrow \min\{\text{ind}(t'), i\}$

(or $\mathcal{N}(s_d)$ if walking from the origin is allowed). Let $dlist$ be list of destination stops. Similar to $\mathcal{L}, \mathcal{J}$, and $\tau_{opt}(n)$ in TBTR, we define $\mathcal{L}(s), \mathcal{J}(s)$, and $\tau_{opt}(n, s)$ for each $s \in dlist$ where $\mathcal{L}(s)$ keeps track of routes through $\mathcal{N}(s)$, $\mathcal{J}(s)$ stores the attributes of optimal journeys to $s$, and $\tau_{opt}(n, s)$ is the optimal label of stop $s$ using at most $n$ transfers, respectively.

Lastly, to preserve labels between runs, we also store the index of the first stop on trip $r$ that can be reached within $n$ transfers, $\text{ind}(n, t)$.

In addition to $dlist$, we also maintain a dummy list $dlist'$ and a variable scope whose purpose is to prune the list of destination stops as the algorithm proceeds. To understand the idea behind pruning the destination list, consider the network shown in Figure 2. For simplicity, imagine that only one trip on the pink and green routes departs from $s_o$ at 0800 and suppose $dlist = \{s_1, s_d\}$. Thus, $\text{list} = \{\text{arr}(t_1, 1), \text{arr}(t_2, 1)\}$. As discussed in Section II-B.2, for Round 0, $Q_0 = \{t_1(1 \rightarrow 3), t_2(1 \rightarrow 3)\}$. Round 1 starts with $Q_1 = \{t_8(1 \rightarrow 3), t_4(1 \rightarrow 2)\}$ and updates $\tau_{opt}(1, s_6)$ to 0820. Since all trip-transfers ($t, i, t', j$) from trips $t_8$ and $t_4$ are such that $\text{arr}(t, i) > \tau_{opt}(1, s_6)$, there are no other optimal journeys to $s_6$. Hence, we can safely remove $s_6$ from $dlist$. Thus, as the algorithm proceeds, the destination list is progressively pruned, resulting in faster queries. Compared to the rTBTR algorithm, the One-To-Many version scans trip-segments fewer times. In the above example, repeated application of rTBTR scans the trip-segments $t_1(1 \rightarrow 3), t_2(1 \rightarrow 3), t_4(1 \rightarrow 3)$, and $t_4(1 \rightarrow 2)$ twice (once for each destination) whereas the One-To-Many rTBTR does it only once.

The pseudocode for the One-To-Many rTBTR is presented in Algorithm 3. Lines 1–7 initialize the variables $\mathcal{L}(s), \mathcal{J}(s), \text{ind}(n, t)$, and $\tau_{opt}(n, s)$. Line 8 iterates over all possible departure times $\tau$ in decreasing order. Lines 8–15 initialize $n$ and $Q_n$ as before. We start by copying all the elements of $dlist$ into a dummy list $dlist'$ in Line 17. In the main while-loop, for each allowed number of transfers $n (\leq \lambda)$, Line 19 first defines an empty set scope. While scanning a trip-segment $t(h \rightarrow k)$, a for-loop (Line 22) is introduced, which iterates over all the stops in $dlist'$. Lines 23–32 scan the trip-segments as described in the TBTR algorithm. An additional step is introduced in Line 29 that adds stops whose labels could improve to scope. Lastly, Lines 35–36 increment $n$ and update $dlist'$ using scope.

1) Proof of Correctness: Observe that if $dlist'$ was the same as $dlist$, i.e., if the destination list was not pruned, then Algorithm 3 is equivalent to repeated application of rTBTR. Also, $s_d$ is deleted in some round $n < \lambda$ only when the if-condition in Line 28 is violated, i.e., “for all trip-segments $t(h \rightarrow k) \in Q_n$, their first stops $t(h + 1)$ are reached later than the best-known arrival times at $s_d$”. Thus, to establish the algorithm’s correctness, it is enough to show that the updates to $\tau_{opt}$ labels are identical in the following two cases: (i) $dlist$ is pruned (ii) $dlist$ is not pruned in round $n$ even when Line 28 is violated for all trip-segments.

In Case (i), given that $s_d$ is removed in round $n$, $\tau_{opt}(m, s_d)$ will not be updated for $n + 1 \leq m \leq \lambda$ because the for-loop in Lines 22–32 will not iterate over $s_d$. In Case (ii), for round $n + 1$, we can show that there does not exist any $t(h \rightarrow k) \in Q_{n+1}$ that satisfies the arrival time condition in Line 23 and hence the algorithm will consequently not update $\tau_{opt}(n + 1, s_d)$. To see why, note that for all trip segments $t(i \rightarrow j) \in Q_n$, since Line 28 was violated, $\text{arr}(t', i + 1) > \tau_{opt}(n, s_d)$. Because every trip segment $t(h \rightarrow k) \in Q_{n+1}$ was added using some trip segment $t'(i \rightarrow j) \in Q_n$ in the previous round, we can conclude that $\text{arr}(t, h) \geq \text{arr}(t', i + 1) > \tau_{opt}(n, s_d)$. At the start of round $n + 1$, the value of $\tau_{opt}(n + 1, s_d)$ is same as $\tau_{opt}(n, s_d)$ since it was set in Lines 24–26 in the previous round. Thus, $\text{arr}(t, h) > \tau_{opt}(n, s_d) = \tau_{opt}(n + 1, s_d)$ and the if-condition in Line 23 is violated. Hence, the $\tau_{opt}$ label for round $n + 1$ is not updated. A similar argument holds for later rounds.
Algorithm 3 One-To-Many rTBTR (GTFS, $T$, $s_o$, $dlist$, $\lambda$, tlist)

1: $L(s), J(s) \leftarrow \emptyset, \emptyset \forall s \in dlist$
2: $ind(n,t) \leftarrow t \forall t \in T, \forall n = 0, 1, \ldots, \lambda$
3: for $s_d \in dlist$ do
4:     $\tau_{opt}(s, s_d) \leftarrow \infty \forall n = 0, 1, \ldots, \lambda$
5: for $s \in N(s_d)$ do
6:     for $(r, i)$ s.t. $r(i) = s$ do
7:         $L(s_d) \leftarrow L(s_d) \cup \{(r, i, f(s, s_d))\}$
8: for $\tau \in tlist$ do
9:     $n \leftarrow 0$
10: $Q_n \leftarrow \emptyset \forall n = 0, 1, \ldots, \lambda$
11: $clist \leftarrow \emptyset$
12: for $s \in N(s_o)$ do
13:     for $(r, i)$ s.t. $r(i) = s$ do
14:         $t \leftarrow$ earliest trip on $r$ s.t. $\tau + f(s_o, s) \leq dep(t, i)$
15:     $clist \leftarrow clist \cup \{(t, i, n)\}$
16: ENQUEUE(clist, ind, $Q_n$)
17: dlist $\leftarrow dlist$
18: while $Q_n \neq \emptyset$ and $n \leq \lambda$ do
19:     scope $\leftarrow \emptyset$
20: for $(h \rightarrow k) \in Q_n$ do
21:     clist $\leftarrow \emptyset$
22: for $s_d \in dlist$ do
23:     for $(route(t), i, \Delta \tau) \in L(s_d)$ do
24:         if $h < i \leq k \text{ and } arr(t, i) + \Delta \tau < \tau_{opt}(s, s_d)$ then
25:             for $m = n, n+1, \ldots, \lambda$ do
26:                 if $\tau_{opt}(m, s_d) > arr(t, i) + \Delta \tau$ then
27:                     $\tau_{opt}(m, s_d) \leftarrow arr(t, i) + \Delta \tau$
28:             if $arr(t, h + 1) < \tau_{opt}(n, s_d)$ then
29:                 add $s_d$ to scope if not already present
30: for $(t, i, j, \lambda) \in T$ s.t. $h < i \leq k$ do
31:                 add $(t', j, n + 1) \rightarrow clist$ if not already present
32: ENQUEUE(clist, ind, $Q_{n+1}$)
33: for $s \in dlist$ do
34:     add $(\tau_{opt}(n, s), n)$ to $J(s)$ if it is non-dominated
35: $n \leftarrow n + 1$
36: dlist $\leftarrow$ scope

1: procedure ENQUEUE(list clist, vector ind, queue $Q$)
2: for $(t, i, n) \in clist$ do
3:     if $i < \text{ind}(n, t)$ then
4:         $Q \leftarrow Q \cup \{(i \rightarrow \text{ind}(n, t))\}$
5: for trip $t'$ s.t. $t \leq t'$ and $\text{route}(t) = \text{route}(t')$ do
6:     for $j = n, n+1, \ldots, \lambda$ do
7:         $\text{ind}(j, t') \leftarrow \min \{\text{ind}(j, t'), i\}$

B. Multilevel Version: MHypTBTR

This section introduces MHypTBTR, i.e., HypTBTR combined with multilevel partitioning. While we propose this concept using HypTBTR, a similar approach can be used for HypRATOR. Consider the example in Figure 6. The base network is an abstraction of a transit network before partitioning. Nodes in white indicate stops, and those in blue show the origin and destination.

1) Standard Partitioning: The figure in the middle shows the network partitioned into six parts (as discussed in Section III). Red nodes represent cutstops. The labels show the stop cell ID for each partition (S1, S2, . . . , S6) and S0 is assumed to contain all the red cutstops. Thus, using standard partitioning, HypTBTR’s fill-in trip set will require finding optimal journeys between 3!2! = 42 origin-destination permutations of the cutstops.

2) Multilevel Partitioning: The figure on the right depicts multilevel partitioning with two levels. In Level 1, the network is partitioned into three parent partitions (S1, S2, S3). Green nodes show the cutstops of Level 1. Next, in Level 2, each parent partition is divided into two subparts (i.e., child partitions). E.g., S1 is divided into S11 and S12. Cutstops are shown using pink nodes here. Finding the fill-in set F can then be divided into two steps:

1) Compute the trips required to travel between cutstops at the topmost level, i.e., 3!2! = 12 origin-destination permutations of the four green cutstops.

2) For each parent partition, determine the trips required to travel between its cutstops and the cutstops of its children. E.g., in Figure 6, arrows between Levels 1 and 2 indicate 4 origin-destination permutations for S1 and its children S11 and S12. Similarly for S2 and its children S21 and S22, we have 8 permutations. Thus, we get a total of 4 + 8 + 4 = 16 permutations between the two levels.

The overall number of origin-destination pairs in the multilevel scheme is 12 + 16 = 28 compared to 42 in the standard scheme (a 33% benefit). Note that if the parent partition is split into two, the optimal trips between the cutstops of child partitions are not required. However, if the split is performed differently, F would also include the optimal trips required to travel between child partitions.

The fill-in computation can be implemented in multiple ways. A straightforward scheme is to store the fill-in trips in a single set F, i.e., there is no distinction between trips required to travel between successive levels and within a level. In this case, MHypTBTR starts from S12 (origin stop cell) in Level 2 and travels up to Level 1 through a trip in F. It then explores a much sparser graph and again transfers down to Level 2 using another trip in F. A Pareto-set is obtained since all the optimal trips between the cutstops are in F. Alternately, a pointer for each trip indicating the level and cutstops for which it is optimal can be maintained (similar to multilevel arc-flags in road networks). This allows additional pruning because, for every trip in F, we have some extra information on the origin-destination pairs for which the trip was optimal. However, the differences between these methods are likely to be pronounced in the query phase. Since the main goal here is to decrease preprocessing, we adopt the former fill-in scheme due to its simplicity.
V. EXPERIMENTS

This section presents the experiments used to evaluate our algorithms. We start with a description of the experimental setup, followed by an overview of the transit networks used. Next, we analyze the performance of non-partitioning algorithms—specifically Dijkstra, RAPTOR, and TBTR—which function as benchmarks for subsequent comparisons. Subsection V-C showcases the performance of various HypTBTR under standard and multilevel partitioning scenarios. Lastly, Subsection V-D analyses the efficacy of our One-To-Many framework.

A. Experimental Setup

All algorithms used in our experiments were implemented in Python 3.7, and the source codes are available on GitHub at transnetlab/transit-routing. The query phase codes were run on an Intel Core i7-8700 CPU @ 3.2 GHz with 32 GB RAM. The more intensive algorithms related to preprocessing were evaluated in parallel using a 128-core Intel Xeon Gold CPU @ 3.0 GHz with 512 GB RAM. The metrics obtained using the parallel method have been marked with an asterisk (e.g., $F$-time$^*$). Query times for One-To-One queries reported in Table III were averaged over 10,000 randomly selected origin and destination stops while capping the maximum runtime at 24 hours. In other words, the runtimes for a few time-consuming instances were averaged over the number of queries successfully executed within 24 hours. To make fair comparisons with existing algorithms (such as RAPTOR, TBTR, and HypRAPTOR), all algorithms were re-implemented and tested using Python.

B. Dataset Description

We tested our algorithms on six transit networks: Switzerland, Netherlands, Sweden, Israel, Taichung, and Bangalore. Except for Bangalore, all are open data sets (Source: transitroutes.com). Bangalore’s network is derived from the Bangalore Metropolitan Transport Corporation (BMTC) timetable. While the first five networks are country-level networks, the last two datasets correspond to city-level networks. Table III summarizes statistics for these networks for a single day. Columns hypedges and hypnodes represent the number of hyperedges and nodes in the corresponding hypergraph, respectively. A significant amount of preprocessing was done to generate these datasets. For example, country-level datasets often contain overtaking trips since they integrate timetables from multiple agencies and modes. As the algorithms discussed in the present study require trips to follow the FIFO property, overtaking trips were discarded. Also, none of the above-mentioned datasets provide footpath details. This information was extracted from OpenStreetMaps (OSM) (geofabrik.de). To do so, all stops were snapped to the nearest OSM coordinate and the corresponding distance matrix was calculated. Next, assuming a constant walking speed of 1 m/s, we obtained footpath edges. Using a threshold on walking time, footpath edges are filtered and subsequently adjusted to satisfy transitivity and triangle inequality. The values in parenthesis in the footpaths column of Table III indicate the threshold on walking time in seconds. Note that these values represent the initial threshold used in the OSM network. Since more footpaths are added to ensure transitivity, the final footpath graph can contain edges that take longer to walk.

Tables III show the query performance, in milliseconds, of the non-partitioning algorithms. For comparison, we also include results from Dijkstra’s algorithm on a Time-Expanded graph (labeled as TED). A maximum transfer limit of four was used for all the algorithms (except TED). Running Dijkstra’s algorithm on the time-expanded graph proved intractable for Bangalore and Israel. As evident from the literature, the following trends can be observed.

- TED vs. others: The query times reported for TED are from NetworkX’s implementation of Dijkstra’s algorithm. The results reconfirm that modern PTR algorithms such as RAPTOR and TBTR perform better than conventional Dijkstra-based approaches.
- RAPTOR vs. TBTR: TBTR outperforms RAPTOR in all instances mainly because it uses the trip-transfer set $T$ to directly switch between trips as opposed to RAPTOR’s method of finding the earliest trip to board a route.

These algorithms will be used as a baseline for evaluating the performance of the partitioning-based algorithms in the next section.

C. Standard and Multilevel Partitioning Algorithms

This subsection analyzes the performance of partitioning-based algorithms on three networks that best illustrate the benefits of the proposed techniques: Switzerland, Netherlands, and Sweden. The trip-transfer computation phase of HypTBTR is the same as that of TBTR. For the fill-in computation phase, we first generate a hypergraph using the GTFS data. Table III contains the details of these hypergraphs. The parameters used for the KaHyPar algorithm are seed $-1$, epsilon $0.2$, and cut_kKaHyPar_sea20.ini configuration.
The top panel in Figure 7 shows the stops when the test networks are partitioned into four cells (blue, green, yellow, and purple) using the standard method. The multilevel partitions are shown in the bottom panel, where shades of the same color depict the children of a parent partition. The weight of a hyperedge corresponding to a stop $s$ is set to the logarithm of the total stop events associated with the stops in $N(s)$. The weight of a node is determined by the total number of events along the corresponding route. Table IV shows results for each of the three networks, from both standard and multilevel
partitioning. A column with label 6 (3-2) implies that the standard partitioning approach splits the network into six parts, and the multilevel partitioning method creates three parent partitions at the upper level and each parent is divided into two child partitions.

The metric \(scut\) indicates the number of cutstops and the percentage of cutstops with respect to the total stops. For example, in the 6 (3-2) scheme for Sweden, the total number of cutstops or \(scut\) is 123 (0.4\%) in standard and 137 (0.4\%) in multilevel partitioning. In the multilevel version, the metric \(scut-p\) and \(scut-c\) indicate the cutstop count at the parent and the child level. In the above example, \(scut-p\), i.e., the number of cutstops in Level 1 is 77 and \(scut-c\), i.e., the number of cutstops in Level 2 are (6, 41, 54). Note that for multilevel partitioning, the sum of \(scut-p\) and \(scut-c\) is not necessarily equal to \(scut\) as some stops act as cutstops in both levels. The metric \(pqueries\) indicates the number of profile queries required for the fill-in set \(F\) computation. With an increase in the number of partitions, both \(scut\) and \(pqueries\) increase as expected. \(F size\) and \(F size^e\) indicate the percentage of trips that are part of the fill-in set \(F\) and the time required to compute \(F\) (in seconds), respectively. The following observations are noteworthy with respect to the above-mentioned metrics.

- The teal-colored values inside the parenthesis indicate the \% benefit of the multilevel partition over its standard version. For example, in Sweden’s 6 (3-2) partitioning case, the number of \(pqueries\) needed are 15006 (standard) and 6922 (multilevel), which translates to the benefit of 53.9\%. A significant reduction in \(pqueries\) and \(F time\) can be seen across standard and multilevel versions in all test cases. An advantage of comparing \(pqueries\) is that it is a language-agnostic metric.

- While multilevel partitioning was consistently better in all test cases, the percentage benefit varies. This is mainly because the partitions (and hence the cutstops) generated depend on the KaHyPar configuration such as \textit{seed} and \textit{epsilon}. However, since the aim here is to compare the benefits of multilevel partitioning over standard partitioning, the configuration parameters used were kept same in all the experiments.

Next, Table IV contains the average query times for HypRAPTOR, HypTBTR, and their multilevel versions MHPRAPTOR, MHypTBTR in milliseconds. Based on these results, the following conclusions can be drawn.

- HypRAPTOR vs. RAPTOR: As expected, HypRAPTOR performs better than RAPTOR in all the test cases. The benefits were found to be in the range of 12–32\%.

- HypTBTR vs. TBTR: HypTBTR consistently performs better than TBTR on all networks. The gain observed was in the range of 23–37\%.

- HypRAPTOR vs. HypTBTR: In all three networks, we observe that the average gains from using HypTBTR over TBTR are more than that between HypRAPTOR and RAPTOR. In other words, the partitioning-based scheme performed better in the TBTR setting than RAPTOR.

A possible reason could again be that TBTR uses trips as building blocks instead of routes. For example, imagine a route with \(m\) trips of which only one is part of the fill-in set. While HypRAPTOR adds the route (and all \(m\) trips) to its fill-in, HypTBTR adds only one trip.

- MHPRAPTOR vs. HypTBTR and MHPRAPTOR vs. HypRAPTOR: Query times of the multilevel versions are in the same range as their standard counterparts. This is expected since the size of the fill-in trips in multilevel partitioning is approximately the same as that of standard partitioning. Multilevel partitioning mainly helped reduce preprocessing times.

We also experimented with alternate hypergraph partitioning tools such as hMETIS, an integer program-based set partitioning method, and studied the effect of different weighting schemes for hypergraph partitioning. As anticipated, the integer program model, solved using CPLEX, exhibited significantly longer partitioning times than hMETIS and KaHyPar. KaHyPar showed a slight edge in performance compared to hMETIS. The proposed weighting scheme, as outlined in Section III-B, proved to be the most effective in the test cases.

### D. One-To-Many rTBTR

The calculation of the fill-in set can be accelerated using faster One-To-Many range queries. The standard approach for obtaining these journeys is to either apply a One-To-One variant repeatedly or to use a One-To-All algorithm and extract the required journeys in a post-processing step. In this section, we evaluate the performance of our proposed One-To-Many rTBTR algorithm. To do so, experiments were carried out by varying the number of randomly selected destination stops from 10 to 1000 (see Column 2 of Table V). This covers the range of cutstop sizes we encountered in the fill-in set calculations. Column 3, OTO, contains runtimes for repeated

![Figure 7. Illustration of standard 4-way and multilevel (2-2) partitioning.](image)
application of rTBTR in [36]. Next, Column 4, OTM without pruning, shows the average runtime for Algorithm 3. Column 5, OTM with pruning, shows Algorithm 3’s performance when destination-based pruning is turned off, i.e., Lines 17, 29, and 36 that implement destination-based pruning by removing the stop from the $d_{\text{list}}$ once it is optimal are removed. While the results in this subsection are for the Switzerland, Netherlands, and Sweden networks, similar trends were observed across the other test cases.

- OTM vs. OTO: In all the test cases, our OTM implementation significantly beats the repeated application of rTBTR by 90–95%, demonstrating the potential of the One-To-Many version in reducing the preprocessing time of HypTBTR and similar algorithms (HypRAPTOR, Transfer Patterns, Scalable Transfer Patterns, etc.). As expected, the runtime for both approaches increases with destination list size.

- OTM with vs. without pruning: The benefits of the pruning scheme increase as the destination list grows. Empirically, the benefits were in the range of 0–10%. Note that when computing the fill-in set $\mathcal{F}$, OTM-rTBTR queries are performed multiple times (once for each origin cutstop), so a small benefit in individual runs can significantly reduce the overall preprocessing computation time.

We also benchmarked the proposed OTM-rTBTR against a One-To-All (OTA) approach, using OTA-rRAPTOR. We observed that the average runtime for OTA-rRAPTOR was much lower than OTA-rTBTR. For example, OTA-rRAPTOR took 7.8, 4.3, and 0.8 seconds on an average for the Switzerland, Netherlands, and Sweden networks, respectively. Although TBTR is generally faster, RAPTOR (like a label-correcting algorithm) works in a One-To-All fashion and has no destination-based calculations. On the other hand, TBTR uses the set $\mathcal{L}$ (Line 17 Algorithm 1) to check if the destination is reachable from the current trip-segment. Hence, TBTR’s performance deteriorates for the One-To-Many case as the number of destination stops increases, as seen in Table V.

For example, with regard to the Sweden network, in queries with 250 destination stops, OTM-rTBTR outperforms OTA-rRAPTOR. However, for a destination list of size 500, OTA-rRAPTOR is faster than OTM-rTBTR. On the other hand, OTM-rTBTR is faster than the Switzerland and Netherlands networks only when the destination sizes are between 100 and 250. Therefore, for One-To-Many queries, the TBTR approach is only beneficial up to a certain threshold; beyond that, it is advisable to switch to the RAPTOR environment.

Recall that the motivation for using OTM algorithms is to speed up fill-in set computation. For standard partitioning used in HypTBTR the number of cutstops in the networks discussed in Table IV varies in the range of 77–567. Thus, depending on the network and the number of partitions, either OTA-rRAPTOR or OTM-rTBTR can be used, but on an average, the advantages of using OTM-rTBTR are relatively marginal.

However, the OTM-rTBTR framework proves beneficial in the multilevel partitioning case because the destination-list sizes in One-To-Many queries are significantly lower due to the distinction between parent-parent and parent-child queries. For example, in Netherlands’ 6 (3-2) scheme of Table IV, the size of the destination list in parent-parent queries is 240 (as opposed to 329 destinations in the standard partitioning). The destination-list sizes from cutstops of the parent to that of the child are 13, 64, and 65. Likewise, we also need to calculate the fill-in trips from each of the cutstops of the child partitions to those in the parent-level. In this example, these destination lists turn out to have sizes 115, 181, and 191 (not shown in the table). In summary, for most instances, the destination list sizes for fill-in calculations of the multilevel scheme were small.
A hybrid approach that uses OTM-rTBTR for smaller number of cutstops and OTA-rRAPTOR for larger destination lists could potentially save additional time during preprocessing.

VI. CONCLUSION AND FUTURE CHALLENGES

A. Conclusion

Seamless journey planning plays an essential role in improving the attractiveness of public transit. Given the widespread use of mobile applications for journey planning, the algorithms used in the backend must be fast and efficient. To achieve this goal, the paper makes the following contributions:

- A novel One-To-One bicriteria transit routing algorithm, HypTBTR, designed to improve the efficiency of backend algorithms for mobile journey planning applications. By leveraging graph partitioning-based methodologies, HypTBTR achieved a significant reduction in query times, ranging from 23% to 37%.

- While the above optimization focuses on query times, it comes with an associated increase in preprocessing efforts. To address this trade-off, we introduced a One-To-Many version tailored for range queries. This extension not only expedited preprocessing times but also enhanced the practicality of the TBTR approach, accommodating user scenarios where multiple stops near a geographical location are involved. Empirical experiments show benefits in the range of 90–95% compared to repeated application of One-To-One queries.

- Additionally, the exploration of a multilevel partitioning paradigm compatible with HypTBTR and HypRAPTOR demonstrated a notable reduction, up to 53%, in fill-in computation calculations. By alleviating preprocessing bottlenecks, these proposed extensions empower TBTR to handle queries in large-scale networks, significantly enhancing scalability and enabling mobile/web systems to manage user traffic adeptly.

B. Future Work

The performance of partitioning-based methods is contingent on network topology. Recall that the benefits in query times are inversely related to the size of the fill-in set (since the network explored during the query phase comprises fill-in and origin/destination regions). For some transit networks, such as Israel, Taichung, and Bangalore, the fill-in set was relatively large. Table VI highlights a few metrics that distinguish these networks. Columns spr and eps denote the average number of routes and events (arrival/departure) per stop. KaHyPar was used for partitioning and the results reported are for four partitions. Figure 8 shows the locations of cutstops for these networks. As can be seen from the table and the figure, due to the large % of cutstops in Israel, Taichung, and Bangalore, the size of fill-in trips is drastically higher. While we can influence the partitions to some extent using different weighting schemes and partitioning algorithms, our empirical tests indicated that the % of cutstops mainly depended on the network topology, particularly the density of footpath graph and the extent of overlapping routes (represented by spr and eps). For this reason, we even set the initial walking threshold (Table III Column footpaths) to 120 s instead of 180 s for these three networks. Since footpaths are connected to complete the graph under transitivity, a larger threshold in a dense network such as Bangalore might form a full clique, resulting in unrestricted walking. As the number of partitions increase, the number of cutstops (and hence the size of the fill-in set) increases.

For future research, methods to generate partitions using past origin-destination query data can be explored to make the proposed algorithms more practical for mobile and real-time applications. Other algorithms, such as CSA and ACSA, were not used for comparison as they only evaluate earliest arrival time queries. Also, Transfer Patterns-based algorithms (Transfer Patterns and Scalable Transfer Patterns) may perform better, but the preprocessing associated with them is likely to be much higher than algorithms in the current study. We defer these comparisons and multilevel partitioning variants of Transfer Patterns for future research. Beyond expediting computational processes, the quality of the output journeys is also equally important. Often, the output set comprises numerous journeys, many of which might be similar. To address this issue, future research can explore efficient post-processing techniques [10], [34]. Furthermore, the timetables in the current study are assumed to be deterministic. However, real-world transit operations are often prone to delays and cancellations, which may require changes to the fill-in set.

The paper focuses on the bicriteria version (travel time and transfers) of TBTR [36]. This approach works well for transit systems that operate under a flat fare policy, that is, the fares are fixed regardless of the origin/destination (e.g., Germany [23], New York [22], and Toronto [35]) or where transit is free (e.g., Luxembourg [1], for women passengers in Delhi [2] and Bengaluru [20]). However, other criteria, such as fares and reliability, are also crucial. Reference [24] expanded TBTR to incorporate a third criterion (travel time or fare zones). Despite this extension, a generalized multi-criteria framework for TBTR remains unexplored. Investigating this extension and examining the implications of partitioning are promising avenues for future research.

Research on complexity analysis of the PTR algorithms is sparse and is another open topic for future research. Multi-objective problems are generally NP-Hard since the efficiency frontier can have exponential solutions (see [31] for details). Reference [13] provided a loose bound of $O(\lambda(\sum_{r \in R}(l_r) + |T| + |F|))$ on the worst-case complexity of RAPTOR using their naive variant (which assumes that the local and destination-based pruning are not used). In such a setting, every route is scanned exactly once per round. However, the above expression does not reflect the true complexity of the RAPTOR algorithm because its success comes mainly from the marking and pruning methods. TBTR’s pruning methods are more involved, which makes a worst-case complexity result an interesting topic for future research.

REFERENCES

[1] Public Transport Fast And Free, The Best Way to Explore The Country. Accessed: Dec. 12, 2023. [Online]. Available: https://luxembourg.public.lu/en/living.html

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
[1] A. Raj. Delhi's 100 Crore Question: What Does A Free Bus Ride Mean For A Woman. Accessed: Dec. 12, 2023. [Online]. Available: https://indianexpress.com/article/cities/delhi/delhis-100-crore-question-what-does-a-free-bus-ride-mean-woman-8519082/

[2] M. Balac and S. Hörl, “Simulation of intermodal shared mobility in the san Francisco bay area using MATSim,” in Proc. IEEE Int. Conf. Transp. Syst. Conf. (ITSC), Sep. 2021, pp. 3278–3283.

[3] H. Bast et al., “Fast routing in very large public transportation networks using transfer patterns,” in Proc. Eur. Sym. Algorithms, 2010, pp. 290–301.

[4] H. Bast et al., “Route planning in transportation networks,” in Algorithm Engineering. Cham, Switzerland: Springer, 2016, pp. 19–80.

[5] H. Bast, M. Hertel, and S. Storandt, “Scalable transfer patterns,” in Proc. 18th Workshop Algorithm Exp. (ALENEX), Jan. 2016, pp. 15–29.

[6] H. Bast, J. Camer, R. Loperfido, S. Proferman, S. Putzer, and L. Timko, “Computing isochrones in multi-modal, schedule-based transport networks,” in Proc. 16th SIGSPATIAL Int. Conf. Adv. Geographic Inf. Syst., Nov. 2008, pp. 1–2.

[7] M. Baum, V. Buchhold, J. Sauer, D. Wagner, and T. Zündorf, “Unlimited transfers for multi-modal route planning: An efficient solution,” 2019, arXiv:1906.04832.

[8] P. Benchimol, A. Amrani, and M. Khouadjia, “A multi-criteria multi-modal predictive trip planner: Application on Paris metropolitan network,” in Proc. IEEE Int. Smart Cities Conf., Sep. 2021, pp. 1–4.

[9] D. Delling, J. Dibbelt, T. Pajor, D. Wagner, and R. F. Werneck, “Computing multimodal journeys in practice,” in Proc. Interface Symp. Experiment. Algorithm. Cham, Switzerland: Springer, 2013, pp. 260–271.

[10] D. Delling, J. Dibbelt, T. Pajor, and T. Zündorf, “Faster transit routing by hyper partitioning,” in Proc. 17th Workshop Algorithmic Approaches for Transp. Model., Opt., Syst., 2017, pp. 1–12.

[11] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, “PHAST: Hardware-accelerated shortest path trees,” J. Parallel Distrib. Comput., vol. 73, no. 7, pp. 940–952, Jul. 2013.

[12] D. Delling, T. Pajor, and R. F. Werneck, “Round-based public transit routing,” Transp. Sci., vol. 49, no. 3, pp. 591–604, Aug. 2015.

[13] D. Delling and R. F. Werneck, “Customizable point-of-interest queries in road networks,” IEEE Trans. Knowl. Data Eng., vol. 27, no. 3, pp. 686–698 Mar. 2015.

[14] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, “Intriguingly simple and fast transit routing,” in Proc. Int. Symp. Experiment. Algorithm. 2013, pp. 43–54.

[15] L. Gottesbüren, M. Hamann, T. N. Uhl, and D. Wagner, “Faster and better nested dissection orders for customizable contraction hierarchies,” Algorithm. vol. 12, no. 9, p. 196, Sep. 2019.

[16] P. He, G. Jiang, S.-K. Lam, Y. Sun, and F. Ning, “Exploring public transport transfer opportunities for Pareto search of multicriteria journeys,” IEEE Trans. Intell. Transp. Syst., vol. 23, no. 12, pp. 22895–22908, Dec. 2022.

[17] A. Khani, “An online shortest path algorithm for reliable routing in schedule-based transit networks considering transfer failure probability,” Transp. Res. Part B, Methodol., vol. 126, pp. 549–564, Aug. 2019.

[18] M. S. Thesis, Univ. Appliance. Inst. Theor. Inform. Sci. Karlsruhe Institut fUr Technologie (KIT), Karlsruhe, Germany, 2020.

[19] T. Rambha, S. D. Boyle, and S. T. Waller, “Adaptive transit routing in stochastic time-dependent networks,” Transp. Sci., vol. 50, no. 3, pp. 1043–1059, Aug. 2016.

[20] J. Dauer, D. Wagner, and T. Zündorf, “An efficient solution for one-to-many multi-modal journey planning,” in Proc. 20th Symp. Algorithmic Approaches Transp. Model., Opt., Syst., 2020, pp. 1–14.

[21] J. Dauer, D. Wagner, and T. Zündorf, “Integrating ultra and trip-based routing,” in Proc. 20th Symp. Algorithmic Approaches Transp. Model., Opt., Syst., 2020, pp. 1–17.

[22] S. Slag, “High-quality hypergraph partitioning,” M.S. thesis, Faculty Comput. Sci. Karlsruhe Institut fUr Technologie (KIT), Karlsruhe, Germany, 2020.

[23] B. Strasser and D. Wagner, “Connection scan accelerated,” in Proc. 16th Workshop Algorithm Exp. Eng. (ALENEX), 2014, pp. 125–137.

[24] B. A. L. Strasser, “Algorithm engineering for adaptive route planning,” Ph.D. thesis, Faculty Comput. Sci., Karlsruhe Institut fUr Technologie (KIT), Karlsruhe, Germany, 2017.

[25] Z. Tarapata, “Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms,” Int. J. Appl. Math. Comput. Sci., vol. 17, no. 2, pp. 269–287, Jun. 2007.

[26] E. B. Tirkolaee, A. Goli, A. Faridinia, M. Soltani, and G.-W. Weber, “Multi-objective optimization for the reliable pollution-routing problem with cross-dock selection using Pareto-based algorithms,” J. Cleaner Prod., vol. 276, Dec. 2020, Art. no. 122927.

[27] E. B. Tirkolaee, A. Goli, S. Gutmen, G.-W. Weber, and K. Szwedzka, “A novel model for sustainable waste collection arc routing problem: Pareto-based algorithms,” Ann. Operations Res., vol. 324, nos. 1–2, pp. 189–214, May 2023.

[28] E. Babaei Tirkolaee, A. Goli, and G.-W. Weber, “Fuzzy mathematical programming and self-adaptive artificial fish swarm algorithm for just-in-time energy-aware flow shop scheduling problem with outsourcing option,” IEEE Trans. Fuzzy Syst., vol. 28, no. 11, pp. 2772–2783, Nov. 2020.

[29] TTC. Fares & Passes. Accessed: Dec. 12, 2023. [Online]. Available: www.ttc.ca

[30] E. Winn, “Trip-based public transit routing,” in Algorithms-ESA 2015. Cham, Switzerland: Springer, 2015, pp. 1025–1036.

[31] E. Winn, “Trip-based public transit routing using condensed search trees,” in Proc. 16th Workshop Algorithmic Approaches for Transp. Model., Opt., Syst., 2016, pp. 1–17.

Prateek Agarwal is currently pursuing the Ph.D. degree in civil engineering with Indian Institute of Science (IISc). His research interests include modeling reliable routing algorithms and journey planning in public transit networks.

Taran Rambha received the Ph.D. degree from UT Austin. He worked on network equilibrium, congestion pricing, and adaptive routing in stochastic networks at UT Austin. He was a Post-Doctoral Researcher with Cornell University, where he studied hospital evacuations and evacuation demand estimation during hurricanes. He is currently an Assistant Professor with Indian Institute of Science (IISc). His current research interests include public and electric mobility systems, logistics, and intelligent parking solutions.