algorithms

MDPI

Article

Journey Planning Algorithms for Massive Delay-Prone Transit Networks

Mattia D'Emidio [1,†], Imran Khan [2,†,*] and Daniele Frigioni [1]

[1] Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Via Vetoio, 67100 L'Aquila, Italy; mattia.demidio@univaq.it (M.D.); daniele.frigioni@univaq.it (D.F.)

[2] Gran Sasso Science Institute (GSSI), Viale Francesco Crispi, 67100 L'Aquila, Italy

* Correspondence: imran.khan@gssi.it

† Parts of this paper appeared in extended abstract form within the proceedings of the 19th International Conference on Computational Science and its Applications (ICCSA2019), Saint Petersburg, Russia, 1–4 July 2019.

Received: 29 September 2019; Accepted: 10 December 2019; Published: 18 December 2019

Abstract: This paper studies the journey planning problem in the context of transit networks. Given the timetable of a schedule-based transportation system (consisting, e.g., of trains, buses, etc.), the problem seeks journeys optimizing some criteria. Specifically, it seeks to answer natural queries such as, for example, "find a journey starting from a source stop and arriving at a target stop as early as possible". The fastest approach for answering to these queries, yielding the smallest average query time even on very large networks, is the Public Transit Labeling framework, proposed for the first time in Delling et al., SEA 2015. This method combines three main ingredients: (i) a graph-based representation of the schedule of the transit network; (ii) a labeling of such graph encoding its transitive closure (computed via a time-consuming pre-processing); (iii) an efficient query algorithm exploiting both (i) and (ii) to answer quickly to queries of interest at runtime. Unfortunately, while transit networks' timetables are inherently dynamic (they are often subject to delays or disruptions), PTL is not natively designed to handle updates in the schedule—even after a single change, precomputed data may become outdated and queries can return incorrect results. This is a major limitation, especially when dealing with massively sized inputs (e.g., metropolitan or continental sized networks), as recomputing the labeling from scratch, after each change, yields unsustainable time overheads that are not compatible with interactive applications. In this work, we introduce a new framework that extends PTL to function in delay-prone transit networks. In particular, we provide a new set of algorithms able to update both the graph and the precomputed labeling whenever a delay affects the network, without performing any recomputation from scratch. We demonstrate the effectiveness of our solution through an extensive experimental evaluation conducted on real-world networks. Our experiments show that: (i) the update time required by the new algorithms is, on average, orders of magnitude smaller than that required by the recomputation from scratch via PTL; (ii) the updated graph and labeling induce both query time performance and space overhead that are equivalent to those that are obtained by the recomputation from scratch via PTL. This suggests that our new solution is an effective approach to handling the journey planning problem in delay-prone transit networks.

Keywords: journey planning; transit networks; dynamic graph algorithms; algorithms engineering; massive datasets; experimental algorithmics

1. Introduction

Computing the "best" journeys in schedule-based transit systems (consisting, e.g., of trains, buses, etc.) is a problem that has been faced at least once by everybody who has ever travelled [1]. In particular, the journey planning problem takes as input a timetable (or schedule), that is, the description, in terms of departure and arrival times, of transits of vehicles between stops within the system, and seeks to answer to natural queries such as "What is the best journey from some stop A to some other stop B if I want to depart at time t?".

Solving (efficiently) such problems constitutes a fundamental primitive in the world of information technologies and intelligent transport systems. Nowadays, in fact, millions of people rely on computer-based journey planning to obtain, accurately and quickly, public transit directions. To this aim, most of the currently deployed journey planners employ algorithms that have been developed and refined in the last couple of decades by researchers in applied algorithmics and algorithm engineering.

In particular, it is known that, despite its simple formulation, the problem is much more challenging than, for instance, the classic route planning problem in road networks [1–3], since schedule-based transit systems exhibit an inherent time-dependent component that requires complex modeling assumptions to obtain meaningful results. For this reason, transit companies in the last decade have invested a lot of resources to develop effective systems called journey planners (like, e.g., Google Transit or bahn.de), that store the schedule via a suitable model/data structure and incorporate algorithms to answer efficiently to various types of queries on such model, seeking best journeys with respect to different metrics of interest. Depending on the considered metric and modelling assumptions, the problem can be in turn specialized into a plethora of optimization problems [1].

The most common type of query in this context is the earliest arrival query, which asks for computing a journey that minimizes the total travelling time from a given departure stop to a given arrival stop, if one departs at a certain departure time. Another prominent type of query is the profile query, which instead asks to retrieve a set of journeys from a given departure stop to a given arrival stop if the departure time can lie within a given range. Further types of queries can be obtained by considering multiple optimization criteria simultaneously (multi-criteria queries) or according to the abstraction at which the problem has to be solved. If, for instance, one wants to optimize the transfer time, that is, the time required by a passenger for moving from one vehicle to another one within a stop, then the journey planning problem is called realistic, while it is referred to as ideal otherwise [2]. In this paper, we focus on the realistic scenario, which is much more meaningful from an application viewpoint.

1.1. Related Work

To solve both the ideal and the realistic version of the problems, a great variety of models and algorithms have been proposed in the literature in the last decade, each exhibiting a different performance. In particular, most of them can be broadly classified into two categories: those representing the timetable as an array and those representing it as a graph (see e.g., Reference [1]). Two of the most successful (and effective) examples of the array-based model are the Connection Scan Algorithm (CSA) [4] and the Round-bAsed Public Transit Optimized Router (RAPTOR) [5]. In CSA, all the elementary connections of a timetable are stored in a single array, which is scanned only once per query. An elementary connection represents a vehicle driving from one stop to another without intermediate stops. The acyclic nature of the timetables is then exploited to solve the earliest arrival problem. In RAPTOR, on the other hand, the timetable is stored as a set of arrays of trips and routes, that is suitably defined sets of elementary connections. This representation is used by a dynamic programming algorithm that operates in rounds and extends partial journeys by one trip per round to solve the problem. Several variants of RAPTOR, either incorporating heuristical improvements or

considering more refined modeling strategies, have been presented and experimentally analyzed in the last couple of years (see, e.g., References [6–8]).

The graph-based models, instead, store the timetable as a suitable graph and execute known adaptations of Dijkstra's shortest path algorithm to compute optimal routes [2,9–11]. Alternatives to both plain array-based and graph-based models have been also recently considered [12–14]. Some of them, like the one in Reference [12], directly operate on the timetable. In details, trips are labeled with the stops at which they are boarded and a precomputed list of transfers to other trips is scanned during a query. Newly reached trips are labeled and for a trip reaching a desired destination, a journey is added to the result set. The algorithm terminates only when all optimal journeys have been found.

Some others, like those in References [13,14], combine a graph representation of the timetable with the notion of graph labeling to achieve extremely low query times. In this paper, we focus on this latter category of approaches, since they are the ones that offer the smallest average query times and are hence suited for modern applications of the journey planning problem.

1.2. Motivation

The fastest solutions to the journey planning problem with respect to query time are those in References [13,14]. Of them, the one in Reference [13] relies on an algorithmic framework referred to, in the following, as Public Transit Labeling (PTL). Such a framework employs a heavy preprocessing phase of the input data to speed up the query algorithm at runtime. This allows to obtain query times that have been experimentally observed to be, on average, the smallest among all known techniques, including RAPTOR, CSA and their variants. Such behavior have been observed on all meaningful real-world inputs that have been tested, including continental sized networks and the method has been shown to scale very well with the networks' size [13].

In more details, PTL consists of three main ingredients: (i) the well-known time-expanded graph model to store transit networks (see e.g., Reference [9]); (ii) a labeling that is a compact representation of the transitive closure of the said graph, computed via a (time-consuming) preprocessing step; (iii) an efficient query algorithm exploiting both the graph and the precomputed labeling to answer quickly to queries of interest at runtime.

On the one hand, the approach outperforms all other solutions in terms of query time and it is general and widely applicable, since several variants of the three mentioned components exist to manage a variety of meaningful application scenarios, including being able to answer to both profile and multi–criteria queries. On the other hand, unfortunately, PTL has the major drawback of not being practical in dynamic scenarios, that is when the network can undergo to unpredictable updates (e.g., due to delays affecting the route traversed by a given vehicle). In particular, even after a single update to the network, queries can return arbitrarily incorrect results, since the preprocessed data can become easily outdated and hence may not reflect properly the transitive closure. Note that recomputing the preprocessed data from scratch, after an update occurs, is not a viable option as it yields unsustainable time overheads, up to tens of hours [13]. Since transit networks are inherently dynamic (delays can be very frequent), the above represents a major limitation of PTL.

Dynamic approaches to update graphs and corresponding (compact) representations of transitive closures have been investigated in the past, in other application domains, due to the effectiveness of such structures for retrieving graph properties [15–22]. However, none of these can be directly employed in the PTL case, where time constraints imposed by the time-expanded graph add a further level of complexity to the involved data structures.

1.3. Contribution of the Paper

In this paper, we move forward toward overcoming the above mentioned limitations, by presenting a new algorithm, named Dynamic Public Transit Labeling (D-PTL, for short), that is able to update the information precomputed by PTL whenever a delay occurs in the transit network, without recomputing it from scratch. It is worthy to mention that, although decreases in departure

times are typically not allowed in transit networks [1], hence updating the information in such case would not be necessary, our solution can be easily extended to manage such scenario. It is also worthy to mention that part of the work we present here already appeared in [23].

The algorithm we present here is based on suited combinations of graph update routines inspired to those in Reference [1] and labeling dynamic algorithms, extensions of those in References [15,21,24]. In particular, we present different versions of the algorithm, that are compatible with the different flavors of PTL, namely single criterion and multi-criteria. Furthermore, we discuss on the correctness of D-PTL and analyse its computational complexity in the worst case.

Asymptotically speaking, the proposed solution is not better than the recomputation from scratch. However, we present an extensive algorithm-engineering based experimental study, conducted on real-world networks of large size, that shows that D-PTL always outperforms the from scratch computation in practice. In particular, our results show that (i) D-PTL is able to update both the graph and the labeling structure orders of magnitude faster than the recomputation from scratch via PTL; (ii) this behavior is amplified when networks are massive in size, thus suggesting that D-PTL scales well with the networks' size. Our data also highlight that the updated graph and labeling structure induce both query time performance and space overhead that are equivalent to those that are obtained by the recomputation from scratch, thus suggesting the use of D-PTL as an effective approach to handle the journey planning problem in delay-prone transit networks.

1.4. Structure of the Paper

The paper is organized as follows. Section 2 gives the basic notation and the definitions used throughout the paper. Sections 3 and 4 describe the PTL approach [13] in its two flavours, namely single and multi-criteria, respectively. Sections 5 and 6 present our new dynamic algorithms, in the basic and multi-criteria versions, respectively and discusses correctness and complexity of the new methods. Section 7 presents our experimental study. Finally, Section 8 concludes the paper and outlines possible future research directions.

2. Background

The journey planning problem takes as input a timetable that contains data concerning stops, vehicles (e.g., trains, buses or any means of transportation) connecting stops and departure and arrival times of vehicles at stops. More formally, a timetable $\mathcal{T}$ is defined by a triple $\mathcal{T} = (\mathcal{Z}, \mathrm{S}, \mathcal{C})$, where $\mathcal{Z}$ is a set of vehicles, S is a set of stops (often in the literature also referred to as stations) and $\mathcal{C}$ is a set of elementary connections whose elements are 5-tuples of the form $c = (Z, s_i, s_j, t_d, t_a)$. Such a tuple is interpreted as vehicle $Z \in \mathcal{Z}$ leaves departure stop $s_i \in \mathrm{S}$ at departure time t_d, and the immediately next stop of vehicle Z is stop $s_j \in \mathrm{S}$ at time t_a (i.e., t_a is the arrival time of Z at arrival stop $s_j \in \mathrm{S}$). Departure and arrival times are integers in $\{0, 1, \ldots, t_{max}\}$ representing times in minutes after midnight, where t_{max} is the largest time allowed within the timetable (typically $t_{max} = (n \cdot 1440 - 1)$, where n is the number of days that are represented by the timetable). We assume $|\mathcal{C}| \geq \max\{|\mathrm{S}|, |\mathcal{Z}|\}$, that is we do not consider vehicles and stops that do not take part to any connection. In the realistic scenario, each stop $s_i \in \mathrm{S}$ has an associated minimum transfer time, denoted by MTT_i, that is the time, in minutes, required for moving from one vehicle to another inside stop s_i.

Definition 1 (Trip). *A trip is a sequence* $\mathrm{TRIP} = (c_1, c_2, \ldots, c_k)$ *of k connections that: (i) are operated by the same vehicle; (ii) share pairwisely departure and arrival stop, that is, formally, we have* $c_{i'-1} = (Z, s_i, s_j, t_d, t_a)$ *and* $c_{i'} = (Z, s_j, s_k, t'_d, t'_a)$ *with* $t'_d > t_d$ *for any* $i' \in [2, k]$.

Clearly, connections in a trip are ordered in terms of the associated departure times, hence we say connection c_j follows connection c_{j-1} in a trip TRIP whenever the departure time of the former is larger than that of the latter. Similarly, we say connection c_j precedes connection c_{j+1} in a trip TRIP.

Definition 2 (Journey). *A journey $J = (c_1, c_2, \ldots c_n)$ connecting two stops s_i and s_j is a sequence of n connections that: (i) can be operated by different vehicles; (ii) allows reaching a given target stop starting from a distinguished source stop at a given departure time $\tau \geq 0$, that is, the departure stop of c_1 is s_i, the arrival stop of c_n is s_j and the departure time of c_1 is larger than or equal to τ; (iii) is formed by connections that satisfy the time constraints imposed by the timetable, namely that if the vehicle of connection c_i is different with reference to that of c_{i+1} at a certain stop s_h, then the departure time of c_{i+1} must be larger than the arrival time of c_i plus* MTT_h.

As well as trips, journeys are implicitly ordered by time according to departure times of the connections. The traveling time of a journey is given by the difference between arrival time of its last connection and τ.

An earliest arrival query $\text{EA}(s_i, s_j, \tau)$ asks, given a triple s_i, s_j, τ consisting of a source stop s_i, a target stop s_j, and a departure time $\tau \geq 0$, to compute a quickest journey, that is, a journey that starts at any $t \geq \tau$, connects s_i to s_j, and minimizes traveling time. In what follows we provide two useful definitions that are necessary to introduce the notion of profile query.

Definition 3 (Time-Dominated Journey). *Let J' and J'' be two journeys, both connecting two stops s_i and s_j. Then journey J'' is time-dominated by journey J' if and only if both the following conditions hold:*

- *the departure time of the first connection of J' is larger than the departure time of the first connection of J'';*
- *the arrival time of the last connection in J' is smaller than the arrival time of the last connection in J''.*

By the above, if we let $\mathcal{J}$ be the set of all journeys connecting two stops s_i and s_j in a transit network, then trivially a journey $J \in \mathcal{J}$ is non-time-dominated if and only if either one of the two following conditions hold: (i) the departure time of the first connection of J is larger than the departure time of the first connection of all other journeys in $\mathcal{J}$; (ii) the arrival time of the last connection of J is smaller than the arrival time of the last connection of all other journeys in $\mathcal{J}$.

Hence, we define a profile query $\text{PQ}(s_i, s_j, \tau, \tau')$ as the one that asks for the set of non-time-dominated journeys between stops s_i and s_j in the time range $\langle \tau, \tau' \rangle$, subject to $\tau < \tau'$, that is, the set of journeys connecting stops s_i and s_j that start at any time in $[\tau, \tau']$ and are non-time-dominated journeys.

Finally, we define a multi-criteria query $\text{MC-EA}(s_i, s_j, \tau)$ as the one asking to compute the set of Pareto-optimal journeys. Informally, such journeys simultaneously optimize more than one criterion (e.g., traveling time and number of vehicle transfers), departing in s_i at some time $t \geq \tau \geq 0$ and arriving at stop s_j. More precisely, given a set of criteria, a journey is in the Pareto-optimal set S if it is non-dominated by any other journey. A journey J_1 dominates a journey J_2 if it is better with respect to every criterion, while it is non-dominated otherwise. Note that, most commonly considered optimization criteria are traveling time and number of vehicle transfers, although other optimization can be found in the literature, for example, monetary cost.

It is long known that the problem of computing the mentioned Pareto-optimal set is (weakly) NP-hard [25], since such journeys can be exponential in number. However, if some degree of importance of the optimization criteria is imposed then the problem is polynomially solvable, by using a simple multi-criteria modification of the Dijkstra's algorithm, based on lexicographical optimality [25]. An example of this scenario is when one wants to compute the set of quickest journeys between two stops s_i and s_j and then, among them, to choose the one minimizing the number of transfers between vehicles. In this paper, we focus on this latter realistic variant of the journey planning problem. As a final remark, observe that profile queries are a special case of multi-criteria ones using arrival and departure times as criteria.

3. Basic Public Transit Labeling

The state-of-the-art method (in terms of query time) to solve the journey planning problem is commonly referred to as Public Transit Labeling (PTL) [13]. The technique comes in two flavors: a basic version to answer to earliest arrival and profile queries only, and an extended, more general version to incorporate generic criteria of optimization, for example, to seek for earliest arrival journeys that also minimize the number of transfers. The basic version essentially consists of three main ingredients:

1. a reduced time-expanded graph, a well-known data structure for storing transit networks (see e.g., Reference [9]);
2. a reachability labeling, a compact labeling-based representation of the transitive closure of the said graph, computed via a (time-consuming) preprocessing step;
3. an efficient query algorithm exploiting both the graph and the labeling to answer quickly to queries of interest at runtime.

In what follows we describe such ingredients in detail.

3.1. Reduced Time-Expanded Graph

The input timetable $\mathcal{T} = (\mathcal{Z}, \mathrm{S}, \mathcal{C})$ associated to the transit network is modelled via a reduced time-expanded graph (RED-TE) [2]. In the case of an aperiodic timetable, the RED-TE graph is a directed acyclic graph (DAG) $G = (V, A)$ [13]. Starting from initially empty sets of vertices V and arcs A, the DAG G associated with the aperiodic timetable $\mathcal{T}$ is built as follows. For each elementary connection $c = (Z, s_i, s_j, t_d, t_a)$:

- two vertices are added to V, namely a departure vertex v_d^c and an *arrival vertex* v_a^c, respectively, each having an associated time $time(v_d^c)$ and $time(v_a^c)$, respectively, such that $time(v_d^c) = t_d$ and $time(v_a^c) = t_a$. Departure and arrival vertices are logically stored within the corresponding stop, that is each vertex v_d^c (v_a^c, respectively) belongs to the set of departure (arrival, respectively) *vertices* $\mathrm{DV}[i]$ ($\mathrm{AV}[j]$, respectively) of stop s_i (s_j, respectively);
- a directed connection arc (v_d^c, v_a^c) is added to A, connecting the corresponding departure and arrival vertices.

Furthermore:

- for each trip $\mathrm{TRIP} = (c_0, c_1, \ldots, c_k)$, and for each connection $c_i \in \mathrm{TRIP}$, $0 \leq i < k$, a bypass arc $(v_a^{c_i}, v_a^{c_{i+1}})$ is added to A, connecting the two arrival vertices of c_i and c_{i+1}.
- for each pair of vertices $u, v \in \mathrm{DV}[i]$, a waiting arc (u, v) is added to A if $time(v) \geq time(u)$, and there is no w in $\mathrm{DV}[i]$ such that $time(v) \geq time(w) \geq time(u)$.
- for each $u \in \mathrm{AV}[i]$ and for each $v \in \mathrm{DV}[i]$, a transfer arc (u, v) is added to A if $time(v) \geq time(u) + \mathrm{MTT}_i$, and there is no $w \in \mathrm{DV}[i]$ such that $time(w) < time(v)$ and $time(w) \geq time(u) + \mathrm{MTT}_i$.

An example of construction of RED-TE graph is given in Figure 1, built using the timetable of Table 1 as input.

Table 1. An example of timetable with three stops X, Y, Z and five vehicles $\alpha, \beta, \gamma, \phi, \theta$.

Departure Stop	Arrival Stop	Departure Time	Arrival Time	VehicleID	Minimum Transfer Time
–	X	–	00:05	α	5
–	X	–	00:07	β	5
X	Y	00:10	00:15	α	5
X	Y	00:15	00:20	β	5
Y	–	00:20	–	α	–
Y	Z	00:25	00:30	β	5
Y	Z	00:30	00:39	γ	5
X	Y	00:35	00:42	ϕ	5
Z	–	00:40	–	θ	5
Y	–	00:50	–	ϕ	–
Z	–	00:50	–	γ	–

In the remainder of the paper, given a directed graph $G = (V, A)$: we denote by $N_{out}(v) = \{u \in V : (v, u) \in A\}$ ($N_{in}(v) = \{u \in V : (u, v) \in A\}$, respectively) the set of *outgoing* (*incoming*, respectively) neighbors of a vertex $v \in V$. We say a vertex u *is reachable from* (*reaches*, respectively) another vertex v if and only if there exists a path from v to u (from u to v, respectively) in G, that is, a sequence of arcs $((v, v_1), (v_1, v_2), \ldots, (v_k, u))$.

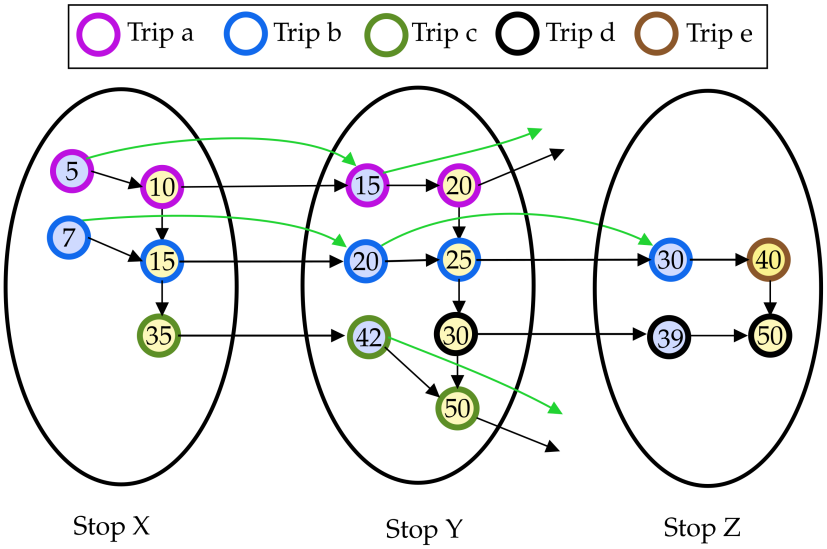

Figure 1. An example of RED-TE graph for a timetable with three stops X, Y and Z, five trips $\{a, b, c, d, e\}$. Details of the timetable are reported in Table 1. Each ellipse groups together vertices in $\text{DV}[i] \cup \text{AV}[i]$ for each stop $i \in \{X, Y, Z\}$, where vertices on the left side of each group, filled in light blue, are arrival vertices, while vertices on the right side, filled in light yellow, are departure vertices. The latter are connected to the former via transfer arcs. The numbers within vertices show the associated time, where the minimum transfer time is assumed to be, for the sake of simplicity, $\text{MTT}_X = \text{MTT}_Y = \text{MTT}_Z = 5$ min for all stops. Departure and arrival vertices of connections of the same trip are highlighted via a same border color. Bypass arcs are drawn in green while consecutive departure vertices of a same set $\text{DV}[i]$ are connected via waiting arcs.

Moreover, we say a path P connecting two vertices u and v is a shortest path between u and v in G if P is a path from u and v in G whose length is minimum among all paths connecting u and v in G. Such length is given by the sum of the weights of the arcs in the path, it is typically denoted by $d_G(u, v)$ and it is often referred to as the distance from u to v in G.

3.2. Reachability Labeling

Given a graph $G = (V, A)$, any approach for computing a so–called *2-Hop-Cover* reachability labeling (2HC-R labeling, for short) L of G associates two labels to each vertex $v \in V$, namely a backward label $\text{L}_{in}(v)$ and a forward label $\text{L}_{out}(v)$, where a label is a subset of the vertices of G [26]. In particular, for any two vertices $u, v \in V$, $\text{L}_{out}(u) \cap \text{L}_{in}(v) \neq \emptyset$ if and only if there exists a path from u to v in G. Therefore, any query on the reachability between two vertices $u, v \in V$ can be answered by a linear scan of the two labels of u and v only [26]. Vertices $\{h : h \in \text{L}_{out}(u) \cap \text{L}_{in}(v)\}$ are called *hub vertices* for pair u, v, and each element in said set is a vertex lying on a path from u to v in G.

The size of a 2HC-R labeling is given by the sum of the sizes of the label entries and it is known that computing a 2HC-R labeling of minimum size is NP-Hard [26]. However, numerous approaches have been presented to heuristically improve both the time to compute the labeling and its size [24,27,28]. Among them, the one in Reference [24], called BUTTERFLY, has been shown to exhibit superior performance for DAGs and is suited for dynamic graphs, that is, the authors also provide a dynamic algorithm that is able to update the 2HC-R labeling L of a graph G to reflect changes occurring on

G itself. In particular, given a graph G, a 2HC-R labeling L of G, and an update operation occurring on G, the algorithm is able to compute another labeling L$'$ that is a 2HC-R labeling for G', where G' is the graph obtained by applying the update on G.

Note that, in this scenario, updates can be incremental (decremental, respectively), if they are additions (removals, respectively) of a vertex/arc. Throughout the paper, we denote by INC-BU(G, L, v) (DEC-BU(G, L, v), respectively) the result of the application of the dynamic algorithm of Reference [24] to the labeling L of a graph G to handle an incremental (decremental, respectively) operation occurring on vertex v (we refer the reader to Reference [24] for more details on the above dynamic algorithms).

3.3. Query Algorithm

It has been shown that the above described RED-TE graph model, combined with 2HC-R labeling, can be used to answer efficiently to both earliest arrival and profile queries on a timetable [13]. In particular, for answering an earliest arrival query of the form EA(s_i, s_j, τ), the algorithm is as follows. First a vertex $c \in$ DV$[i]$ satisfying the following conditions is computed:

1. $time(c) \geq \tau$;
2. there is no arc (c', c) adjacent to c such that $c' \in$ DV$[i]$ and $time(c') \leq time(c)$.

Then, vertices in AV$[j]$ are scanned to search for vertices $v \in$ AV$[j]$ such that $\text{L}_{out}(c) \cap \text{L}_{in}(v) \neq \emptyset$ (i.e., that are reachable). If such a vertex v exists, meaning that there exists at least a journey connecting the two stops, then $time(v)$ is returned as the earliest arrival time, only if there is no other $v' \in$ AV$[j]$ such that $\text{L}_{out}(c) \cap \text{L}_{in}(v') \neq \emptyset$ and $time(v') < time(v)$ (this is easy to check as vertices are typically sorted by time). Note that the structure of the journey can be easily retrieved by applying a recursive query procedure [1,13,21].

To answer a profile query PQ(s_i, s_j, τ, τ'), instead, the algorithm needs to compute a set of non-dominated journeys, let us call it PROFILE, by proceeding as follows. First, the routine computes a vertex $c \in$ DV$[i]$ such that:

1. $time(c) \geq \tau$;
2. there is no arc (c', c) adjacent to c such that $c' \in$ DV$[i]$ and $time(c') \leq time(c)$.

Then, vertices of AV$[j]$ are scanned to search for a vertex $v \in$ AV$[j]$ such that:

1. $\text{L}_{out}(c) \cap \text{L}_{in}(v) \neq \emptyset$ (i.e., v is reachable from c);
2. there is no other vertex $v' \in$ AV$[j]$ having $time(v') < time(v)$ and $\text{L}_{out}(c) \cap \text{L}_{in}(v') \neq \emptyset$.

Note that $time(v)$ is the earliest time to arrive at stop s_j given the departure time τ from s_i. Now, since the set PROFILE must contain all non-dominated journeys, the latest departure time that allows to reach s_j is also necessary to complete the computation of the query. To this aim, the algorithm computes another vertex $c'' \in$ DV$[i]$ such that there is no arc (c'', c'''), with $c''' \in$ DV$[i]$, having $time(c'') \leq time(c''')$ and $\text{L}_{out}(c''') \cap \text{L}_{in}(v) \neq \emptyset$. The computed time $time(c'')$ is the latest departure time that allows to reach s_j. Hence, the algorithm adds pair $(time(c''), time(v))$ to PROFILE (or alternately the corresponding journey, both versions of the set PROFILE are equivalent [13]), and repeats the process above by setting the value of τ to $time(c'') + 1$.

Stop Labeling

In order to obtain a very fast query time compatible with modern applications, in Reference [13] a customization of the general query approach, tailored for RED-TE graphs, was proposed. In particular, the main idea underlying the PTL query algorithm is to compact labels and to associate them to stops, rather than to vertices. To this aim, PTL builds a RED-TE (There is a one-to-one correspondence between RED-TE graphs and classic time–expanded graphs [2]) graph G, computes a 2HC-R labeling L of G, and compresses it into a set of stop labels SL of L [13]. In detail, we have a forward stop label $\text{SL}_{out}(i)$ and a backward stop label $\text{SL}_{in}(i)$ for each stop $s_i \in$ S. A forward (backward, respectively)

stop label is a list of pairs of the form $(v, stoptime_i(v))$ where v is a hub vertex reachable from (that reaches, respectively) at least one vertex in $\text{DV}[i]$ ($\text{AV}[i]$, respectively) and $stoptime_i(v)$ encodes the latest departure (earliest arrival, respectively) time from s_i to reach hub vertex v (from the stop, say s_v, of vertex v to reach s_i, respectively).

For the sake of the efficiency, entries in $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) are stored as sorted arrays, in increasing order of hub vertices (according to distinct ids assigned to vertices). The set of stop labels is usually referred to as stop labeling of G (or of L). Similarly to the general 2HC-R labeling case, queries on the timetable can be answered via stop labels by scanning the entries associated to source and target stops only. The query algorithms exploit the information in the stop labeling to discard time-dominated journeys toward the stored hubs and to achieve query times of the order of milliseconds [13].

In particular, the routine for answering to an earliest arrival query $\text{EA}(s_i, s_j, \tau)$ using stop labels is as follows. Since $\text{SL}_{out}(i)$ and $\text{SL}_{in}(j)$ are arrays sorted with respect to ids, the algorithm as a first step finds the vertex v in $\text{SL}_{out}(i)$ ($\text{SL}_{in}(j)$ whose time is greater than or equal to τ. Assume that said vertex is in position p (q, respectively) in such arrays.

Then, a linear sweep, starting from location p, is performed on $\text{SL}_{out}(i)$ to find the first entry $(v, stoptime_i(v))$ satisfying the condition that $stoptime_i(v) \geq \tau$. Let us assume this entry is stored in location $p' \geq p$. This part of the computation is known as the process of computing relevant hubs and it is followed by the computation of all hubs that are both $\text{SL}_{out}(i)$ and $\text{SL}_{in}(j)$, stored at locations greater than p' and q in $\text{SL}_{out}(i)$ and $\text{SL}_{in}(j)$, respectively. Finally, the earliest arrival time among all such common hubs, computed in the previous step, is returned as an answer to the query $\text{EA}(s_i, s_j, \tau)$.

The query algorithm for answering a profile query $\text{PQ}(s_i, s_j, \tau, \tau')$ using stop labels works as follows. First, the algorithm performs the computation of relevant hubs, which returns p' and q as the locations in $\text{SL}_{out}(i)$ and $\text{SL}_{in}(j)$, respectively. Then, all hubs in both $\text{SL}_{out}(i)$ and $\text{SL}_{in}(j)$, stored at locations greater or equal to p' in $\text{SL}_{out}(i)$, and q in $\text{SL}_{in}(j)$ are computed. Finally, among all such common hubs, all non-dominated journeys satisfying the condition that the departure is less than or equal to τ' are added to PROFILE, which is initially an empty set. At the end of the procedure, PROFILE is returned as the answer to the profile query $\text{PQ}(s_i, s_j, \tau, \tau')$.

4. Multi-Criteria Public Transit Labeling

The basic PTL approach is not naturally designed for answering to multi-criteria queries, which require a more careful design to achieve lexicographical optimality for generic optimization criteria. However, besides optimizing arrival time, many users also prefer journeys with fewer transfers. To this aim, in Reference [13], the authors show how the basic approach can be modified to handle general multi-criteria queries by modifying its constituents. Briefly, a weighted reduced time-expanded graph (WRED-TE, for short) is used in place of the RED-TE graph (again note that there is a one-to-one correspondence between RED-TE graphs and classic time–expanded graphs [2]), a shortest path labeling replaces the reachability labeling, and the query algorithm is modified accordingly. In what follows, without loss of generality, we describe the modification designed to manage, as optimization criteria, the traveling time and the number of transfers, both to be minimized. However, both PTL and our approach can be extended to handle other criteria (e.g., monetary cost) [13].

4.1. Weighted Reduced Time-expanded Graph

In the specific case, when the additional criterion to be considered is the number of transfers, the weighted reduced time-expanded graph is obtained as follows: each transfer arc (u, w) in the graph is assigned a weight of value equal to 1. By interpreting weights of 1 as "leaving a vehicle", we can count the number of trips taken along any path. To model staying in the vehicle, consecutive connection vertices of the same trip are linked by zero-weight arcs. In such a way, the weight of paths encodes the number of transfers taken during a journey while the duration of the journey itself can still be deduced from the time difference of the vertices. Thus, if one prefers paths in the graph

having minimum weight, besides optimizing the time criterion as shown in the previous section, then the sought journey will be the one exhibiting minimum arrival time and minimum number of transfers between vehicles. To this aim, a shortest path labeling [26], instead of a reachability labeling, is employed to accelerate the computation of shortest paths. Notice that, differently from the case of basic PTL, to the best of our knowledge no compact version of the shortest path labeling is known, that is, there is no analog of stop labelings (see Section 3.3) for the multi-criteria setting.

4.2. Shortest Path Labeling

Given a directed graph G, a *2-Hop-Cover shortest path labeling* (shortly, 2HC-SP labeling) L of G associates two labels $\mathrm{L}_{in}(v)$ and $\mathrm{L}_{out}(v)$ to each vertex v in V, called backward label and forward label, respectively. Differently from reachability labelings, in this case, each label contains additional information, namely each entry in $\mathrm{L}_{in}(v)$ ($\mathrm{L}_{out}(v)$, respectively) is of the form (h, δ_{hv}) ((h, δ_{vh}), respectively), where:

- (h, δ_{hv}) represents a vertex h in G from which v can be reached via a shortest path of length δ_{hv};
- (h, δ_{vh}) represents a vertex h in G reachable from v via a shortest path of length δ_{vh};
- label entries satisfy the so–called *cover property* that is, for any pair of vertices $u, v \in V$, the distance $d(u, v)$ (i.e., the weight of the shortest path) from u to v in G can be retrieved by a linear scan of the two labels of u and v only.

In details, a query on the distance is defined as follows:

$$\mathrm{QUERY}(u, v, \mathrm{L}) = \begin{cases} \min_{h \in V} \{\delta_{uh} + \delta_{hv} \mid (h, \delta_{uh}) \in \mathrm{L}_{out}(u) \wedge (h, \delta_{hv}) \in \mathrm{L}_{in}(v)\} & \textbf{if } \mathrm{L}_{out}(u) \cap \mathrm{L}_{in}(v) \neq \emptyset \\ \infty & \textbf{otherwise.} \end{cases}$$

It can be shown that, for any 2HC-SP labeling, $\mathrm{QUERY}(u, v, \mathrm{L})$ always equals $d(u, v)$ [26], that is for any two connected vertices $u, v \in V$, we have $\mathrm{L}_{out}(u) \cap \mathrm{L}_{in}(v) \neq \emptyset$ and the minimum value of the sums equals the weight of a shortest path in the graph.

In particular, in this case we call hub vertices for pair u, v the vertices in

$$\{k : k \in \arg\min_{h \in V} \{\delta_{uh} + \delta_{hv} \mid (h, \delta_{uh}) \in \mathrm{L}_{out}(u) \wedge (h, \delta_{hv}) \in \mathrm{L}_{in}(v)\}\},$$

where each element in said set is a vertex lying on a shortest path from u to v in G. In the above definition we slightly overload our notation by saying that h belongs to $\mathrm{L}_{out}(v)$ ($\mathrm{L}_{in}(v)$, respectively) whenever $(h, \delta_{vh}) \in \mathrm{L}_{out}(v)$ ($(h, \delta_{hv}) \in \mathrm{L}_{in}(v)$, respectively). Note that, despite the same nomenclature, the notion of hub vertex here is more restrictive with respect to reachability labelings, as it requires the vertex to be on a shortest path rather than on any path. To this regard, for 2HC-SP labelings, the following definition can be given. We refer the reader to Reference [29] for more details.

Definition 4 (Induced Path). *Given a graph $G = (V, A)$, a pair $s, t \in V$ and a* 2HC-SP *labeling L of G, a shortest path P is induced by L for pair $s, t \in V$ if, for any two vertices u and v in P, there exists a hub h of pair (u, v) such that $h \in P$, or $h = u$, or $h = v$. The set of shortest paths between vertices s and t induced by L is denoted by* $\mathrm{PATH}(s, t, L)$.

Finally, note that, as well as reachability labelings, also for shortest path labelings the size of the labeling is given by the sum of the sizes of the label entries and it is known that computing a 2HC-SP covering of the graph of minimum size is NP-Hard, by a simple reduction to the 2HC-R case [26]. However, numerous approaches have been presented to heuristically improve both the time to compute the labeling and its size [24,27,28].

A reference approach for DAGs (as the WRED-TE is) is that of Reference [24] which, in the case of shortest path labelings, relies on computing a topological order over the vertices of G. A topological

order Ton the vertex set V for a DAG is a total ordering defining a precedence relationship among the vertices such that for any arc (u, v) in G we have $t(u) < t(v)$, where $t(w)$ is the position in the ordering of the generic vertex $w \in V$. Note that a topological order can be computed in linear time with respect to the size of the graph. For details about the mentioned approach, we refer the reader to [24] and references therein.

Note also that, for a pair $u, v \in V$, the shortest path between u and v in G can be also retrieved from the labeling L by deploying a recursive procedure that builds the path by repeatedly combining hub vertices of pairs of vertices belonging to the path. This is possible due to the optimal sub-structure of shortest paths, where each sub-path of a shortest path is itself a shortest path. We refer the reader to Reference [29] for more details. Such a path between u and v is commonly known as the path induced by the labeling, which in our case is L.

4.3. Multi-Criteria Query Algorithm

It is known that the above described WRED-TE graph model, combined with a 2HC-SP labeling, can be used to answer efficiently to both earliest arrival and profile queries on a timetable (see Reference [13] for more details). In particular, the routine for answering a multi-criteria query MC-EA(s_i, s_j, τ) is as follows. First, a vertex $c \in$ DV$[i]$ satisfying the following conditions is computed:

1. $time(c) \geq \tau$;
2. there is no arc (c', c) in G such that $c' \in$ DV$[i]$ and $time(c') \geq \tau$, meaning that vertex c is associated with the smallest departure time larger than τ.

Then, the algorithm computes $d(c, v)$, by querying the shortest path labeling L, for all $v \in$ AV$[j]$ such that $time(c) < time(v)$, and selects the arrival vertex v having minimum $d(c, v)$. Finally, the time associated to such vertex is the earliest arrival time while $d(c, v)$ is the associated number of transfers. Hence, the corresponding journey is the one having the smallest number of transfers among those exhibiting the earliest arrival time: the structure of the journey again can be retrieved by applying a recursive query procedure [13,21]. Note that, to achieve the fastest possible query times, PTL employs some pruning mechanisms [13]. Notice also that, differently from basic PTL, no compact version of shortest path labelings is known, that is there is no analog of stop labelings (see Section 3.3) for multi-criteria queries.

5. Dynamic Public Transit Labeling

In this section, we introduce Dynamic Public Transit Labeling (D-PTL, for short), a new technique that is able to maintain the PTL data structure under delays occurring in the given transit network. In particular, we first show a dynamic algorithm (referred to as basic D-PTL) to update the basic PTL framework, that is how to maintain both a RED-TE graph $G = (V, A)$, the corresponding 2HC-R labeling L and stop labeling SL under delays affecting connections, and then discuss on how to extend this procedure to the multi-criteria setting.

Formally, a *delay* is an increase in the departure time of an elementary connection of a finite quantity $\delta > 0$. Hence, it is easy to see how a delay can induce an arbitrary number of changes to both the graph and labelings [2,13], depending on the structure of the trip the connection belongs to, thus in turn inducing arbitrarily wrong answers to queries.

A general strategy to achieve the purpose of updating both G, the 2HC-R labeling L and the stop labeling SL, after a delay, while preserving the correctness of the queries, is to first update the graph representing the timetable (via, e.g., the solutions in References [2,9,10]) and then reflect all these changes on both L and SL by: (i) detecting and removing obsolete label entries; and (ii) adding new updated label entries induced by the new graph, as done in other works on the subject [21,24]. However, this results in a quite high computational effort, as shown by preliminary experimentation we conducted.

In order to minimize the number of changes to both L and SL, we hence exploit the specific structure of the RED-TE graph and design a dynamic algorithm that alternates phases of update of the graph with phases of update of the labeling L through the procedures given in Reference [24]. At the end of such phases, changes to L are reflected onto its compact representation SL through a dedicated routine. In particular, our algorithm is based on the following observation: a delay affecting a connection of a trip might be propagated to all subsequent connections in the same trip, if any. Hence, the impact of a given delay on both the graph and the labelings strongly depends on δ, on the structure of the trip and, in particular, on the departure times of subsequent connections. Therefore, D-PTL processes connections of a trip incrementally, and in order with respect to departure time. In details, D-PTL comprises two sub-routines, called, respectively, removal phase (Algorithm REM-D-PTL, see Algorithm 1) and insertion phase (Algorithm INS-D-PTL, see Algorithm 2) that update L along with the graph. Such phases are then followed by a bundle update of SL by a suitable procedure (Algorithm UPDATESTOPLAB, see Algorithm 3).

Algorithm 1: Algorithm REM-D-PTL.

Input: RED-TE graph G, a delay $\delta > 0$ affecting a connection c_m, the trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$ including the connection

Output: RED-TE graph G not including vertices of connections violating RED-TE constraints and the 2HC-R labeling L of G

for $j = m, m+1, \ldots, k-1, k$ **do**
  Let s_s and s_t be departure and arrival, respectively, stops of c_j;
  PRED $\leftarrow \infty$;
  SUCC $\leftarrow \infty$;
  $time(v_d^{c_j}) \leftarrow time(v_d^{c_j}) + \delta$;
  $time(v_a^{c_j}) \leftarrow time(v_a^{c_j}) + \delta$;
  foreach $v \in N_{out}(v_d^{c_j})$ **do** // Outgoing arcs (if any)
    if $v \in \text{DV}[s]$ **then** SUCC $\leftarrow v$; // Waiting arc in the graph
  if $time(v_d^{c_j}) > time(\text{SUCC})$ **then**
    foreach $v \in N_{in}(v_d^{c_j})$ **do** // Incoming arcs (if any)
      if $v \in \text{DV}[s]$ **then** PRED $\leftarrow v$; // Waiting arc in the graph
      if $v \in \text{AV}[s]$ **then** $\overline{A} \leftarrow \overline{A} \cup \{v\}$; // Transfer arc in stop s_s
    $V \leftarrow V \setminus \{v_d^{c_j}\}$;
    L $\leftarrow$ DEC-BU$(G, \text{L}, v_d^{c_j})$;
    if PRED $\neq \infty \wedge$ SUCC $\neq \infty$ **then** $A \leftarrow A \cup \{(\text{PRED}, \text{SUCC})\}$; // Add waiting arc;
    foreach $w \in \overline{A}$ **do** $A \leftarrow A \cup \{(w, \text{SUCC})\}$; // Add transfer arcs;
    L $\leftarrow$ INC-BU$(G, \text{L}, \text{SUCC})$;
  foreach $v \in N_{out}(v_a^{c_j})$ **do** // Outgoing arcs (if any)
    if $v \in \text{DV}[t] \wedge time(v) < time(v_a^{c_j}) + \text{MTT}_t$ **then**
      $V \leftarrow V \setminus \{v_a^{c_j}\}$;
      L $\leftarrow$ DEC-BU$(G, \text{L}, v_a^{c_j})$;
  if *G has not changed* **then break**;

Algorithm 2: Algorithm INS-D-PTL.

Input: RED-TE graph G not including vertices of connections violating RED-TE constraints, the 2HC-R labeling L of G, delay $\delta > 0$, delayed connection c_m, trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$

Output: RED-TE graph G including vertices of connections affected by the delay, the 2HC-R labeling L of G, the delay $\delta > 0$ affecting the connection c_m and the trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$ including the connection

for $j = m, m+1, \ldots, k-1, k$ **do**
  Let s_s and s_t be departure and arrival stops of c_j, respectively.;
  PRED $\leftarrow \infty$;
  SUCC $\leftarrow \infty$;
  foreach $v \in N_{out}(v_d^{c_j})$ **do** // Outgoing arcs (if any)
    if $v \in \text{DV}[s]$ **then** SUCC $\leftarrow v$; // Waiting arc in the graph
  foreach $v \in N_{in}(v_d^{c_j})$ **do** // Incoming arcs (if any)
    if $v \in \text{DV}[s]$ **then** PRED $\leftarrow v$; // Waiting arc in the graph
  if $v_d^{c_j} \in V \wedge v_a^{c_j} \in V$ **then** // (Case I) - Both not removed
    Execute REWIRETRANSFERDEP$(G, v_d^{c_j}, \text{SUCC}, s_s)$; // i.e., Algorithm 4
    if *G has changed* **then**
      L $\leftarrow$ INC-BU$(G, \text{L}, v_d^{c_j})$;
  else if $v_d^{c_j} \notin V \wedge v_a^{c_j} \notin V$ **then** // (Case II) - Both Removed
    $V \leftarrow V \cup \{v_d^{c_j}\}$; // Add $v_d^{c_j}$ to G
    Execute REWIREWAITINGDEP$(G, v_d^{c_j}, s_s)$; // i.e., Algorithm 4
    Execute REWIRETRANSFERDEP$(G, v_d^{c_j}, \text{SUCC}, s_s)$; // i.e., Algorithm 5
    L $\leftarrow$ INC-BU$(G, \text{L}, v_d^{c_j})$;
    $V \leftarrow V \cup \{v_a^{c_j}\}$; $A \leftarrow A \cup \{(v_d^{c_j}, v_a^{c_j})\}$; // Add $v_a^{c_j}$ and connection arc to G
    Execute REWIREARR$(G, v_a^{c_j}, \text{TRIP}_i, s_t)$; // i.e., Algorithm 6
    L $\leftarrow$ INC-BU$(G, \text{L}, v_a^{c_j})$;
  else if $v_d^{c_j} \notin V \wedge v_a^{c_j} \in V$ **then** // (Case III) - Only $v_d^{c_j}$ removed
    $V \leftarrow V \cup \{v_d^{c_j}\}$;
    $A \leftarrow A \cup \{(v_d^{c_j}, v_a^{c_j})\}$; // Add $v_d^{c_j}$ and connection arc to G
    Execute REWIREWAITINGDEP$(G, v_d^{c_j}, s_s)$; // i.e., Algorithm 4
    Execute REWIRETRANSFERDEP$(G, v_d^{c_j}, \text{SUCC}, s_s)$; // i.e., Algorithm 5
    L $\leftarrow$ INC-BU$(G, \text{L}, v_d^{c_j})$;
  else // (Case IV) - Only $v_a^{c_j}$ removed
    $V \leftarrow V \cup \{v_a^{c_j}\}$; $A \leftarrow A \cup \{(v_d^{c_j}, v_a^{c_j})\}$; // Add $v_a^{c_j}$ and connection arc to G
    Execute REWIREARR$(G, v_a^{c_j}, \text{TRIP}_i, s_t)$; // i.e., Algorithm 6
    L $\leftarrow$ INC-BU$(G, \text{L}, v_a^{c_j})$;

Algorithm 3: Algorithm UPDATESTOPLAB.

Input: Outdated stop labeling SL, 2HC-R labeling L of G, sets US_{out}, US_{in}
Output: Updated stop labeling SL of L

foreach $s_i \in \text{US}_{out}$ **do**
  $Q \leftarrow \emptyset$;
  $\text{SL}_{out}(i) \leftarrow \emptyset$;
  while $\{\text{DV}[s] \setminus Q\} \neq \emptyset$ **do**
    $m \leftarrow argmax_{v \in \text{DV}[s]}\, time(v)$;
    foreach $u \in \text{L}_{out}(m)$ **do**
      if $u \notin \text{SL}_{out}(i)$ **then**
        $\text{SL}_{out}(i) \leftarrow \text{SL}_{out}(i) \cup \{(u, time(m))\}$;
    $Q \leftarrow Q \cup \{m\}$;
Sort $\text{SL}_{out}(i)$ with respect to vertices ids;
foreach $s_i \in \text{US}_{in}$ **do**
  $Q \leftarrow \emptyset$;
  $\text{SL}_{in}(i) \leftarrow \emptyset$;
  while $\{\text{AV}[s] \setminus Q\} \neq \emptyset$ **do**
    $m \leftarrow argmin_{v \in \text{DV}[s]}\, time(v)$;
    foreach $u \in \text{L}_{in}(m)$ **do**
      if $u \notin \text{SL}_{in}(i)$ **then**
        $\text{SL}_{in}(i) \leftarrow \text{SL}_{in}(i) \cup \{(u, time(m))\}$;
    $Q \leftarrow Q \cup \{m\}$;
Sort $\text{SL}_{in}(i)$ with respect to vertices ids;

Algorithm 4: Algorithm REWIREWAITINGDEP.

Input: Graph $G = (V, A)$, departure vertex $v_d^{c_j}$, stop s_s
if $\text{DV}[s] \setminus \{v_d^{c_j}\} \neq \emptyset$ **then**
  $m \leftarrow argmax_{v \in \text{DV}[s]}\, time(v)$;
  if $time(v_d^{c_j}) \geq time(m)$ **then** `// Add waiting arc`
    $A \leftarrow A \cup \{(m, v_d^{c_j})\}$;
else
  Let $m_1, m_2 \in \text{DV}[s]$ be such that $time(m_1) \leq time(v_d^{c_j}) \leq time(m_2)$;
  $A \leftarrow A \setminus \{(m_1, m_2)\}$; `// Remove outdated waiting arc`
  $A \leftarrow A \cup \{(m_1, v_d^{c_j}), (v_d^{c_j}, m_2)\}$; `// Add new waiting arcs`

Algorithm 5: Algorithm REWIRETRANSFERDEP.

Input: Graph $G = (V, A)$, departure vertex $v_d^{c_j}$, successor vertex SUCC, stop s_s

if SUCC $= \infty$ **then** // (Sub-case I.a)
  CANDIDATES $\leftarrow \emptyset$;
  foreach $v \in$ AV$[s]$ **do**
    TO_ADD $\leftarrow$ *true*;
    foreach $u \in N_{out}(v)$ **do** // Outgoing transfer arcs (if any)
      if $u \in$ DV$[s]$ **then** // Has transfer arc
        TO_ADD $\leftarrow$ *false*;
        break;
    if TO_ADD $\wedge$ $time(v_d^{c_j}) \geq time(v) +$ MTT$_s$ **then**
      CANDIDATES $\leftarrow$ CANDIDATES $\cup \{v\}$
  foreach $v \in$ CANDIDATES **do**
    $A \leftarrow A \cup \{(v, v_d^{c_j})\}$
else // (Sub-case I.b)
  $T \leftarrow \emptyset$;
  foreach $v \in$ AV$[s]$ **do**
    foreach $u \in N_{out}(v)$ **do** // Outgoing transfer arcs (if any)
      if $u =$ SUCC $\wedge$ $time(v_d^{c_j}) \geq time(v) +$ MTT$_s$ **then**
        $T \leftarrow T \cup \{(v, u)\}$;
  foreach $(v, u) \in T$ **do**
    $A \leftarrow A \setminus \{(v, u)\}$;
    $A \leftarrow A \cup \{(v, v_d^{c_j})\}$;

Algorithm 6: Algorithm REWIREARR.

Input: Graph $G = (V, A)$, arrival vertex $v_a^{c_j}$, trip TRIP$_i$, stop s_t

MIN_NODE $\leftarrow$ $\underset{v \in \text{DV}[t],\ time(v) \geq time(v_a^{c_j}) + \text{MTT}_t}{argmin} \{time(v)\}$;
$A \leftarrow A \cup \{(v_a^{c_j}, \text{MIN\_NODE})\}$; // Add proper transfer arc
if $j > 0$ **then** // Not first connection of the trip
  $A \leftarrow A \cup \{(v_a^{c_{j-1}}, v_a^{c_j})\}$; // Add bypass arc
if $j < k$ **then** // Not last connection of the trip
  $A \leftarrow A \cup \{(v_a^{c_j}, v_a^{c_{j+1}})\}$; // Add bypass arc

In the removal phase, we first remove from G vertices and arcs that are associated with the delayed connection that violate the RED-TE constraints. We say a vertex (arc, respectively) violates the RED-TE constraints whenever the associated time (the difference of the times of the endpoints, respectively) does not satisfy at least one of the inequalities imposed by the RED-TE model discussed in Section 2. Note that, vertices and arcs of the above kind can be: (i) departure and arrival vertices of the delayed connection; (ii) departure and arrival vertices following the delayed connection in the same trip; (iii) arcs adjacent to vertices in (i) and (ii).

Once the above is done, we might have that G is no longer a RED-TE graph, since the removal of the above vertices and arcs can, in turn, induce some other vertex/arc to violate RED-TE constraints. Hence, we first reflect such removals onto L by running the decremental algorithm DEC-BU of Reference [24] and then check if we need to insert into G some new arcs to let it be again a RED-TE graph. Accordingly, if this is the case, we add label entries induced by these insertions by using the incremental algorithm INC-BU of Reference [24]. At this point, the graph G is a RED-TE graph of a timetable that does not include the delayed connection. Then, if some changes has been applied to

G (and L) in the above step, we proceed by analyzing the connections following the delayed one in the same trip, one by one, and by removing vertices and arcs that violate the RED-TE graph. At the end of these iterations, we have that G is a RED-TE graph of a timetable that does not include neither the delayed connection nor those following it in the same trip that have violated the RED-TE constraints because of δ.

After completing the above, we perform the insertion phase, where we check whether we need to insert back into G some vertices and arcs, with updated associated times, to let the graph be a RED-TE graph of the updated timetable. This might require to execute algorithm INC-BU to add label entries induced by such insertions. Once both G and L have been updated, we reflect the changes onto the stop labeling via a suited routine (see Algorithm 3). In the next sections we describe in detail the above sub-routines.

5.1. Removal Phase

In the negative case, we do not remove $v_d^{c_j}$ since, after updating $time(v_d^{c_j})$, all vertices of $\text{DV}[s]$ do not violate the time inequalities imposed by waiting arcs. In the affirmative case (see Line 9), instead, $v_d^{c_j}$ must be removed and the arcs adjacent to vertices in $\text{DV}[s]$ and $\text{AV}[s]$ must be rewired. In particular, we proceed as follows: if there exists some waiting arc $(v_d^{c_j}, v)$ in A, that is, there is some other $v \in \text{DV}[s]$ whose time was larger than or equal to that of $v_d^{c_j}$ before the delay), and $time(v_d^{c_j}) > time(v)$ (thus the ordering imposed by waiting arcs is violated), then we compute a set $\overline{A}$ of vertices that will be wired at v, given by $\overline{A} = \{w \ : \ (w, v_d^{c_j}) \in A \ : \ w \in \text{AV}[s]\}$. Note that the time of said vertex v is necessarily larger than the time of vertices $w \in \text{AV}[s]$ such that $(w, v_d^{c_j}) \in A$ plus MTT_s, thus satisfy the RED-TE inequality for transfer arcs.

Moreover, we search for two vertices, named PRED and SUCC respectively, defined as follows:

- PRED is the unique vertex (if any) such that $\text{PRED} \in \text{DV}[s]$ and $(\text{PRED}, v_d^{c_j}) \in A$;
- SUCC is the unique vertex (if any) such that $\text{SUCC} \in \text{DV}[s]$ and $(v_d^{c_j}, \text{SUCC}) \in A$.

These are the vertices adjacent to the waiting arcs having $v_d^{c_j}$ as one endpoint, that we will need to rewire to preserve the RED-TE properties. Then, we remove $v_d^{c_j}$ from V, and run DEC-BU to obtain an updated version of the 2HC-R labeling (see Line 14). Note that the removal of a vertex $v_d^{c_j}$ also removes all arcs $(v, v_d^{c_j})$ and $(v_d^{c_j}, v)$ (if any) from A. Finally, we add: a waiting arc $(\text{PRED}, \text{SUCC})$ to A, if both PRED and SUCC are vertices in the graph, and a transfer arc for each entry in $\overline{A}$. In particular, for each vertex $w \in \overline{A}$, we add a new transfer arc (w, SUCC). To reflect such changes on L, we run INC-BU (see Line 16).

Regarding vertex $v_a^{c_j}$, graph G remains unchanged either if there is no transfer arc in A having $v_a^{c_j}$ as endpoint, or if there is a transfer arc $(v_a^{c_j}, v)$ but such arc is not affected by the delay, that is, when $time(v) \geq time(v_a^{c_j}) + \text{MTT}_t$. In all other cases, we proceed by removing $v_a^{c_j}$ from G and by updating L via DEC-BU (see Line 21). An example of execution of the removal phase is shown in Figure 2.

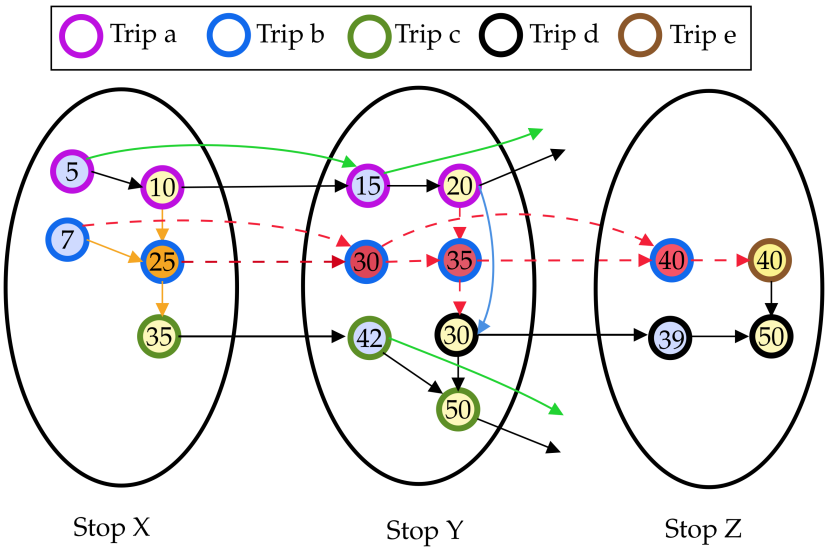

Figure 2. The RED-TE graph obtained after performing Algorithm REM-D-PTL (Algorithm 1) on the graph of Figure 1, as a consequence of a delay δ of 10 min occurring on the first connection of Trip b. The time associated to the departure vertex (filled in orange) of said connection is updated, but the vertex and its corresponding transfer arc (drawn in orange) are not removed, since the ordering and the RED-TE properties are not broken. Arrival and departure vertices of the connections following the one affected by the delay in the same trip are filled in red. Since they break the RED-TE properties (e.g., 35 becomes larger than 30 in Stop Y) they are removed from the graph, along with the corresponding adjacent arcs, shown via dashed red arrows. A waiting arc, in blue, is added during the removal phase to connect departure vertices that remain in $\text{DV}[Y]$ to restore the RED-TE properties.

As a final remark on this part, notice that (see Figure 2) the removal phase is stopped at a given connection c_i of trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_i, \ldots, c_k)$, with $m \leq i \leq k$ whenever the delay does not induce a change neither in the time associated to $v_a^{c_i}$ and $v_d^{c_i}$ nor in their adjacent arcs, as this trivially implies that no change will be performed on all vertices $v_a^{c_j}$ and $v_d^{c_j}$ (and their adjacent arcs) for all j, with $i < j \leq k$. This can be detected by comparing the *status* of vertices (namely time and set of adjacent arcs) before and after performing the procedure for a given connection. In the remainder of the paper, for the sake of brevity, we denote this test by writing either "the graph has changed" or not.

5.2. Insertion Phase

In this section, we discuss in details Algorithm INS-D-PTL whose aim is adding to G vertices and arcs according to the delayed connection in such a way G is a RED-TE graph properly representing the updated timetable, and then to update accordingly L (see Algorithm 2). In particular, once Algorithm 1 has been executed, the following four cases can occur, for each connection c_j , $j = m$ to k in trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$ that has been affected by the delay, depending on whether the vertices associated have been removed or not from the graph:

(a) $v_d^{c_j} \in V$ and $v_a^{c_j} \in V$;
(b) $v_d^{c_j} \notin V$ and $v_a^{c_j} \notin V$;
(c) $v_d^{c_j} \notin V$ and $v_a^{c_j} \in V$;
(d) $v_d^{c_j} \in V$ and $v_a^{c_j} \notin V$.

In what follows we describe in detail how Algorithm INS-D-PTL manage each of these cases.

5.2.1. Discussion on Case I

In this case, when both vertices have remained in G (see Line 9 of Algorithm 2), we only check whether some transfer arcs have to be updated. This process is summarized in Algorithm 5 which is called as sub-routine by Algorithm 2. In particular, if $v_d^{c_j}$ is the last vertex in $\text{DV}[s]$ (see Line 2 of Algorithm 5—Sub-case I.a), i.e., there is no waiting arc outgoing $v_d^{c_j}$ then we compute the subset CANDIDATES of vertices in $\text{AV}[s]$ that do not have any adjacent transfer arc and would not violate the RED-TE constraints, i.e., we add a vertex $v \in \text{AV}[s]$ to CANDIDATES if and only if $time(v_d^{c_j}) \geq time(v) + \text{MTT}_s$ and v does not have any adjacent transfer arc.

Then, for each vertex $v \in$ CANDIDATES we add a new arc $(v, v_d^{c_j})$ to A.

If, instead, $v_d^{c_j}$ is not the last vertex in $\text{DV}[s]$ (see Line 14 of Algorithm 5—Sub-case I.b), i.e., there exists some waiting arc connecting $v_d^{c_j}$ to a vertex SUCC $\in \text{DV}[s]$, then some of the transfer arcs having SUCC as endpoint in G may need to be updated and connected to $v_d^{c_j}$ (i.e., *rewired* to $v_d^{c_j}$). To this purpose, we first determine the subset TA of transfer arcs in A having w as endpoint and then, for each arc (v, w) in TA, if $time(v_d^{c_j}) \geq time(v) + \text{MTT}_s$ we replace arc (v, w) by a new arc $(v, v_d^{c_j})$. Notice that, for replaced transfer arcs we do not need to update L, since any two vertices that were reachable before such update remain reachable afterward. Moreover, also vertices in $\text{DV}[s]$ remain in ordered form, therefore we do not need to add/replace any waiting arc of A. On the contrary, if some modification has been applied to the topology of G or to the ordering of the vertices, then we run INC-BU to obtain an updated version of the 2HC-R labeling (see Line 11).

5.2.2. Discussion on Case II

In this case, occurring when both vertices have been removed from V (see Line 13), we know that the affected connection has no counterpart in G in terms of departure and arrival vertices. Thus, to make G reflect the updated network as a correct RED-TE model, we proceed as follows.

First, we add a vertex $v_d^{c_j}$ to V and to $\text{DV}[s]$ and set its associated time to be equal to the new departure time of the (delayed) connection. After that, we add arcs adjacent to $v_d^{c_j}$, depending on the presence of other vertices in $\text{DV}[s]$ and $\text{AV}[s]$ and on their times. In particular, if $time(v_d^{c_j}) \geq time(v)\ \forall\ v \in \text{DV}[s]$ and $\text{DV}[s] \setminus \{v_d^{c_j}\} \neq \emptyset$, i.e., there is no waiting arc outgoing vertex $m = argmax_{v \in \text{DV}[s]}\, time(v)$ and there exists another departure vertex besides $v_d^{c_j}$ in $\text{DV}[s]$, we need to add a waiting arc incoming into $v_d^{c_j}$, in particular we insert arc $(m, v_d^{c_j})$ into A.

On the other hand, if there exist some vertices $m_1, m_2 \in \text{DV}[s]$ such that $time(m_1) \leq time(v_d^{c_j}) \leq time(m_2)$, then we remove waiting arc (m_1, m_2) and add two new waiting arcs $(m_1, v_d^{c_j})$ and $(v_d^{c_j}, m_2)$ to A. It is worth to remark here that $v_d^{c_j}$ cannot be such that $time(v_d^{c_j}) < time(v)\ \forall\ v \in \text{DV}[s]$ since otherwise the original vertex $v_d^{c_j}$ would have not been removed by Algorithm 1. The pseudo-code of this part of the insertion phase is shown in Algorithm 4 which is again executed as sub-routine of Algorithm 2. Regarding transfer arcs, after $v_d^{c_j}$ is inserted we execute Algorithm 5, as already discussed for case I. Finally, we run INC-BU to update the 2HC-R labeling L (see Line 17).

Once vertex $v_d^{c_j}$ has been handled, we focus on the arrival stop s_t and insert a vertex $v_a^{c_j}$ into V and $\text{AV}[t]$, and a connection arc $(v_d^{c_j}, v_a^{c_j})$ to A. Then, to properly set transfer arcs induced by such connection arc, we search for the vertex v in $\text{DV}[t]$ such that: (i) $time(v) \geq time(v_a^{c_j}) + \delta$ and (ii) $time(v)$ is minimum among vertices satisfying (i). If such a vertex v exists, then we add arc $(v_a^{c_j}, v)$ to A. Moreover, to properly set bypass arcs, if $j \geq 1$ we add an arc $(v_a^{c_{j-1}}, v_a^{c_j})$, where we remark that $v_a^{c_{j-1}}$ is the arrival vertex of connection c_{j-1} of TRIP_i. Similarly, $j \leq k-1$ we add an arc $(v_a^{c_j}, c_{j+1})$ where $v_a^{c_{j+1}}$ is the arrival vertex of connection c_{j+1} of TRIP_i (see Algorithm 6 for the pseudo-code of this phase). Again, we run INC-BU to update L (see Line 20).

5.2.3. Discussion on Case III

In this case, when $v_d^{c_j}$ has been removed while $v_a^{c_j}$ is in V (see Line 21 of Algorithm 2), we first add a vertex $v_d^{c_j}$ to V and to $\text{DV}[s]$ and a connection arc $(v_d^{c_j}, v_a^{c_j})$ to A. This is followed by the wiring of suited transfer and waiting arcs to $v_d^{c_j}$, in order to preserve the RED-TE properties. As in the previous cases, this is achieved by Algorithms 4 and 5, discussed above. Algorithm INC-BU is also run to reflect changes on the 2HC-R labeling (see Line 26).

5.2.4. Discussion on Case IV

In this case, occurring when $v_d^{c_j}$ is part of V while $v_a^{c_j}$ has been removed by the removal phase (see Line 27), we insert a vertex $v_a^{c_j}$ into V and $\text{AV}[t]$, and the corresponding connection arc $(v_d^{c_j}, v_a^{c_j})$ into A. This is followed by the addition of bypass and transfer arcs adjacent to $v_a^{c_j}$, achieved again by Algorithm 6. Furthermore, we obtain the final version L of the 2HC-R labeling (see Line 30).

An example of execution of the insertion phase is shown in Figure 3. In addition, for the sake of simplicity in understanding, we show an example of execution of the procedures for: (i) rewiring transfer arcs (Algorithm 5) and waiting arcs (Algorithm 4) to a departure vertex in Figure 4; and (ii) rewiring arcs to an arrival vertex (Algorithm 6) in Figure 5.

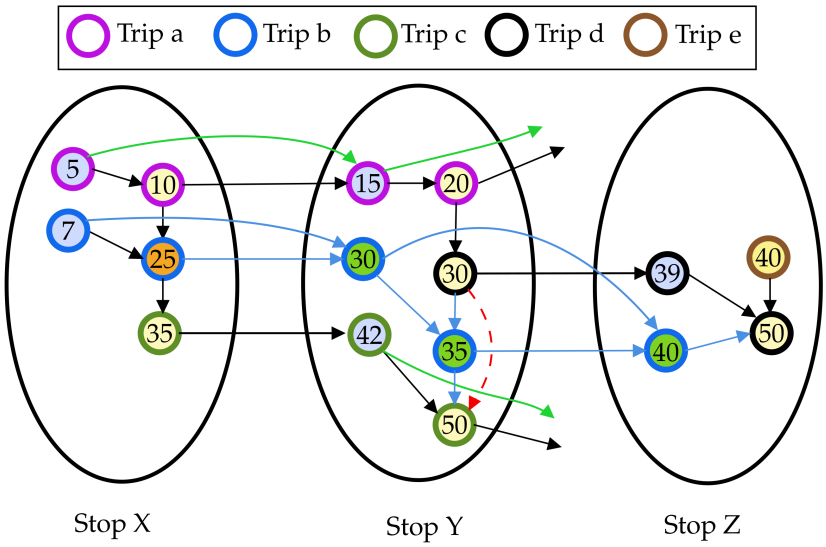

Figure 3. The RED-TE graph obtained after performing Algorithm INS-D-PTL (Algorithm 2) on the graph of Figure 2. Newly added vertices (arcs, respectively) are drawn in green (blue, respectively). The dashed arc drawn in red is the waiting arc of Figure 2 that is removed in the insertion phase.

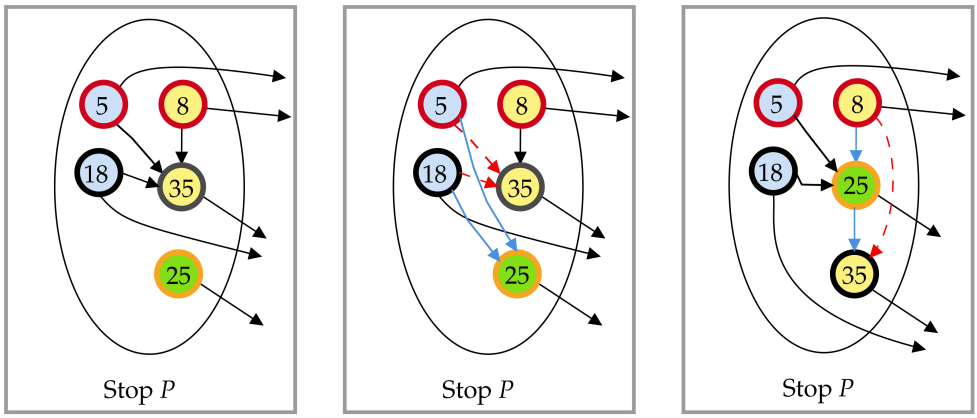

Figure 4. An example of execution of the procedure for rewiring transfer and waiting arcs given in Algorithms 4 and 5, respectively. On the left we show part of a sample graph, relative to a stop P with the assumption that $\text{MTT}_P = 5$ min, that is violating the RED-TE properties. In particular, no waiting and transfer arcs are associated with the vertex colored in green. To restore the RED-TE properties both transfer and waiting arcs must be added. Concerning the former, Algorithm 5 is executed and the resulting graph is shown in the middle, where dashed arcs in red (arcs in blue, respectively) are the removed arcs (newly inserted arcs, respectively). Regarding the latter, instead, Algorithm 4 is executed. The resulting RED-TE graph (on the right side) is the final outcome. Dashed arcs in red are the removed arcs, while arcs in blue are the newly added ones.

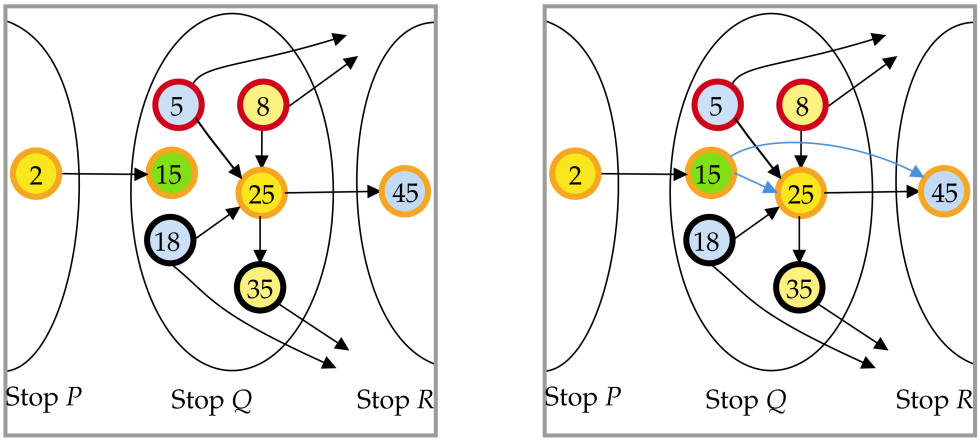

Figure 5. Part of a sample graph, relative to three stops, namely P, Q and R, is shown on the left side, where the minimum transfer time is assumed to be, for the sake of simplicity, $\text{MTT}_P = \text{MTT}_Q = \text{MTT}_R = 5$ min for all stops. Both transfer and bypass arcs for the vertex of $\text{AV}[Q]$ highlighted in green must be rewired, in order to restore RED-TE properties. To this end, Algorithm 6 is executed, and the result is shown on the right, with newly added arcs are highlighted in blue.

5.3. Updating the Stop Labeling

Once both the graph and the 2HC-R labeling have been updated, if a corresponding compressed stop labeling SL is available and one wants to reflect the mentioned updates on said compressed structure, a straightforward way would be that of recomputing the stop labeling from scratch, via for example, the routine in Reference [13]. This computational effort is not large as that required for recomputing the 2HC-R labeling. However, we propose a alternative routine that is incorporated in D-PTL and avoids (and it is faster than) the recomputation from scratch of the stop labeling. Our routine requires, during the execution of Algorithms 1 and 2, to compute two sets of so–called *updated stops*, denoted, respectively, by US_{out} and US_{in}. These are defined as the stops $s_i \in \text{S}$ such that vertices in $\text{DV}[i]$ ($\text{AV}[i]$, respectively) had their time value or forward label (backward label, respectively)

changed during Algorithm REM-D-PTL or during Algorithm INS-D-PTL. Sets US_{out} and US_{in} can be easily determined by inserting stops satisfying the property in said sets during the execution of Algorithms 1 and 2, after each update to times or labels.

Once this is done we update the stop labeling SL by recomputing only the entries of $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) for each $s_i \in \text{US}_{out}$ (for each $s_i \in \text{US}_{in}$, respectively). To this aim, for each stop $s_i \in \text{US}_{out}$ ($s_i \in \text{US}_{in}$, respectively) we first reset $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) to the emptyset. Then, we scan departure (arrival, respectively) vertices in decreasing (increasing, respectively) order with respect to time and add entries to $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) accordingly. In particular, for all departure (arrival, respectively) vertices v of s_i in the above mentioned order, we add a pair $(u, stoptime_i(v))$ for each u in $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) only if there is no pair $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) having u as hub vertex. This guarantees that each pair contains latest departure (earliest arrival, respectively) times. After updating the stop labels, we sort both $\text{SL}_{out}(i)$ and $\text{SL}_{in}(i)$ to restore the ordering according to the hub vertices [13]. Details on how to update the stop labeling by executing the procedure are given in Algorithm 3.

We are now ready to give the following results.

Theorem 1 (Correctness of Basic D-PTL)**.** *Given an input timetable and a corresponding* RED-TE *graph G, let* L *be a* 2HC-R *labeling of G and let* SL *be a stop labeling associated to* L*. Assume $\delta > 0$ is a delay occurring on a connection, that is, an increase of δ on its departure time. Let G',* L$'$*, and* SL$'$ *be the output of* D-PTL *when applied to G,* L *and* SL*, respectively, by considering the delay. Then: (i) G' is a* RED-TE *graph for the updated timetable; (ii)* L$'$ *is a* 2HC-R *labeling for G'; (iii)* SL$'$ *is a stop labeling for* L$'$*.*

Notice that, the above theorem is based on the correctness of the approaches in References [2,13,24]. In particular, it is easy to see that whenever we update the graph, we do it by preserving the constraints imposed by the RED-TE model on both vertices, by suitably modifying connection arcs and associated waiting, bypass, and transfer arcs. In more details, it is easy to prove, by contradiction, that after the execution of Algorithms 1 and 2, G is a RED-TE graph. Concerning the labeling data structures, observe that after each change to G we use either DEC-BU or INC-BU, depending on the type of performed modification. These algorithms have been shown to compute a labeling that is a 2HC-R labeling for the modified graph [24]. Hence, at the end of Algorithm 2, L is a 2HC-R labeling for G. Finally, Algorithm 3 applies the definition of stop labeling, by updating the entry of a stop with the proper hub vertices and times values. Hence, after the execution of Algorithm 3, SL is a stop labeling of L and the theorem follows.

Theorem 2 (Complexity of Basic D-PTL)**.** *Algorithm* D-PTL *takes $\mathcal{O}(|\mathcal{C}|^3 \log |\mathcal{C}|)$ computational time in the worst case.*

Proof. The complexity of Algorithm D-PTL is given by the sum of the complexities of Algorithms 1, 2 and 3. In what follows, we analyze separately the three algorithms.

Concerning Algorithm 1, we first bound the cost of executing Lines 1–21, that is, the amount of computational time per connection. Lines 1–8 require a time that is linear in the number of neighbors (incoming and outgoing) of $v_d^{c_j}$, which is a constant in RED-TE graphs, while lines 9–21 spend a time that grows as said number of neighbors times the time required for performing the dynamic algorithms DEC-BU and INC-BU. Each execution of these algorithms takes $\mathcal{O}(|V|^2 \log |V|)$ in the worst case [24]. Thus, lines 1–21 require $\mathcal{O}(|V|^2 \log |V|)$ time in the worst case. These lines are repeated for all stops traversed by the vehicle of the trip from connection c_m to c_k, therefore in the worst case for all stops of the transit network, which are $|\text{S}| \leq |\mathcal{C}|$. Since $|V| \in \mathcal{O}(|\mathcal{C}|)$, we have that Algorithm REM-D-PTL runs in $\mathcal{O}(|\mathcal{C}|^3 \log |\mathcal{C}|)$ worst case time.

Concerning Algorithm 2, notice that all sub-routines require a time that is linear in the size of the processed stop (i.e., in the number of associated arcs). Hence, by summing up the contribution for all considered stops (those traversed by the trip from connection c_m to c_k), we obtain that updating

the graph via Algorithm INS-D-PTL takes $\mathcal{O}(|\mathcal{C}|)$, as $|\mathcal{C}| \geq \max\{|S|, |\mathcal{Z}|\}$ and, in the worst case, the affected trip can traverse all stops of the network. On top of that, we need again to consider the time for executing DEC-BU and INC-BU, which are performed again $|S| \leq |\mathcal{C}|$ times in the worst case. Since $|V| \in \mathcal{O}(|\mathcal{C}|)$, we have that Algorithm INS-D-PTL runs in $\mathcal{O}(|\mathcal{C}|^3 \log |\mathcal{C}|)$ worst case time.

Concerning Algorithm 3, it scans label entries of vertices in both US_{in} and US_{out} in non–increasing and non–decreasing order, respectively (thus requiring either to sort them or to use a priority queue). In both cases, we have an additional logarithmic factor in terms of computational time per vertex. Since all vertices for all stops can be $\mathcal{O}(|\mathcal{C}|)$, and since sorting stop labels with respect to hub vertices at the end of the procedure requires $\mathcal{O}(|\mathcal{C}| \log |\mathcal{C}|)$ worst-case time, it follows that the worst case time of Algorithm 3 is $\mathcal{O}(|\mathcal{C}| \log |\mathcal{C}|)$. If we sum up the complexities of Algorithms 1, 2 and 3, the claim follows. □

Notice that, Theorem 2 implies that D-PTL, in the worst case, is slower than the reprocessing from scratch via PTL, whose worst case running time is cubic in the size of the graph due to the recomputation of the labeling [20].

However, our experimental study, which is described in Section 7, clearly shows that D-PTL always outperforms PTL in practice.

6. Dynamic Multi-Criteria Public Transit Labeling

In this section, we extend D-PTL to handle the multi-criteria setting. We refer to the extended version as multi-criteria D-PTL.

We remark that, to update the data structures employed by the basic PTL framework, D-PTL exploits the structure of the RED-TE graph and alternates phases of modifications of the graph itself with corresponding updates of the reachability labeling via the procedures given in [24]. These phases are bundled in two blocks, namely the *removal phase* (Algorithm REM-D-PTL, see Algorithm 1) and *insertion phase* (Algorithm INS-D-PTL, see Algorithm 2) that update the labeling along with the graph.

The above two routines, however, cannot be directly employed within the multi-criteria PTL approach, that relies on a shortest path labeling rather than on a reachability one. In particular, while the modifications to the graph applied by the two routines are almost same for WRED-TE graphs (the only exception is that whenever we add a transfer arc we need also to add a suited intermediate vertex for modeling a transfer, whenever the two vertices are associated to connections of different trips.), we cannot use algorithm BUTTERFLY, which is designed for reachability labelings, to update the shortest path labeling at hand. Hence, we need to replace DEC-BU (in lines 14 and 21 of Algorithm 1) and INC-BU (in lines 11, 17, 20, 26, and 30 of Algorithm 2) with decremental and incremental algorithms that are suited to update the 2HC-SP labeling. To this regard, we can employ the decremental algorithm DECPLL of Reference [21] and the incremental algorithm INCPLL of Reference [15], respectively, that are designed to update 2HC-SP in general graphs.

Unfortunately, by preliminary experiments we conducted on some relevant instances of the problem (we recall the reader that graphs treated in this paper are specifically DAGs), we observed that, while INCPLL is quite fast and updates the labeling within few seconds even in very large graphs, DECPLL is painfully slow, and sometimes its computational time is comparable with that required for recomputing the labeling from scratch. This is most likely due to the sparse nature of the RED-TE graph and to how DECPLL updates 2HC-SP labelings. In more details, DECPLL works in three phases whose running time depends proportionally on the cardinality of the set of vertices that contain at least a label entry that is incorrect. It is easy to see that this cardinality tends to the number of vertices of the graph in DAGs in most of the cases (see Reference [21] for more details on this part of the computation).

For such reasons, in what follows we propose an extension of algorithm DEC-BU, named DAG-DECPLL, that is explicitly designed to update shortest path labelings in DAGs, instead of reachability labelings, as a consequence of decremental updates to the graph. The main intuition behind DAG-DECPLL is to exploit the specific relationships between shortest paths in DAGs, which are instead neglected by DECPLL, which is designed for general graphs.

Given a graph $G = (V, A)$, we discuss the new approach by focusing on how to handle the removal of a vertex, say $x \in V$, which is the decremental operation of interest in our scenario. Note that, the routine can be easily extended to handle arc removals or arc weight increases, as discussed at the end of this section. In what follows, we call $G' = (V', A')$ the graph obtained by removing vertex x from V. Furthermore, we denote by $d_S(u, v)$ the distance (i.e., the weight of a shortest path) between two vertices u and v of a graph, say S. and define two subsets of vertices of V, namely RIGHT_x and LEFT_x, as follows:

- RIGHT_x: the set of vertices of V that are reachable from x in G, i.e., $u \in \text{RIGHT}_x$ if and only if there exists a path from x to u in G;
- LEFT_x: the set of vertices of V that can reach x in G, i.e., $u \in \text{LEFT}_x$ if and only if there exists a path from u to x in G.

Since G is a DAG, it is easy to see that RIGHT_x and LEFT_x are inherently disjoint, that is $\text{RIGHT}_x \cap \text{LEFT}_x = \emptyset$. Additionally, given the above definitions, we say a label entry $(h, \delta_{vh}) \in \text{L}_{out}(v)$ of some vertex $v \in V$ is *affected* by the removal of a vertex $x \in V$ only if x lies on a shortest path between v and h induced by L. Similarly, a label entry $(h, \delta_{hv}) \in \text{L}_{in}(v)$ is *affected* by the removal of a vertex $x \in V$ only if x lies on a shortest path between h and v induced by L.

In what follows, given a vertex $x \in V$, we highlight some simple yet important properties of the two sets RIGHT_x and LEFT_x that are easily derived by the structure of DAGs.

Property 1. *For any vertex $v \in V$ such that $v \notin \text{RIGHT}_x \cup \{x\}$ no label entry in $\text{L}_{in}(v)$ is affected by the removal of x from G.*

Corollary 1. *For any vertex $v \in \text{RIGHT}_x$, a label entry (h, δ_{hv}) in $\text{L}_{in}(v)$ may be affected only if $h \in \text{LEFT}_x$ or if $v = x$.*

Property 2. *For any vertex $v \in V$ such that $v \notin \text{LEFT}_x \cup \{x\}$ no label entry in $\text{L}_{out}(v)$ is affected by the removal of x from G.*

Corollary 2. *For any vertex $v \in \text{LEFT}_x$, a label entry (h, δ_{vh}) in $\text{L}_{out}(v)$ may be affected only if $h \in \text{RIGHT}_x$ or if $v = x$.*

Lemma 3. *For any pair of vertices u, v in V, if $u \in \text{LEFT}_x$ and $v \notin \text{RIGHT}_x$, then $\text{QUERY}(u, v, \text{L}) = d_{G'}(u, v)$. Symmetrically, if $v \in \text{RIGHT}_x$ and $u \notin \text{LEFT}_x$ then $\text{QUERY}(v, u, \text{L}) = d_{G'}(v, u)$.*

Proof. The above easily follows by Properties 1 and 2. Notice that, when $h \notin \text{LEFT}_x \cup \{x\}$ ($h \notin \text{RIGHT}_x \cup \{x\}$, respectively), the shortest path from h to v (from v to h, respectively) cannot pass through x by the definition of LEFT_x (RIGHT_x, respectively). □

According to the previous observations, we now provide a strategy to carefully identify the label entries that are affected by the removal of a vertex x from G. In particular, for each vertex $v \in \text{RIGHT}_x$ ($v \in \text{LEFT}_x$, respectively) we know that $\text{L}_{in}(v)$ ($\text{L}_{out}(v)$, respectively) can contain affected label entries, which must be either removed or updated in order to preserve the correctness of the query algorithm. The routine to achieve the update is based on the notion of *marking* label entries, that is we assume to store an additional boolean field, attached to each label entry, encoding the information "the label entry is marked or not". We assume initially all these bits are set to false.

Given the additional boolean field, we define a so–called *marked query* between two vertices u and v, denoted as $\text{MQUERY}(u, v, \text{L})$, that behaves as a regular query on the labeling with the difference that it considers only those label entries that are either marked or such that their associated vertices do not belong to either LEFT_x or RIGHT_x. This is done with the purpose of distinguishing label entries that have already been updated with the correct distance or such that the attached distance is not changed

by the removal of x. We will show later in the section how this modified query is used to retrieve correct distances during the update.

Algorithm DAG-DECPLL, whose pseudocode is given in Algorithm 7, exploits the above properties and definitions and works as follows. Given the vertex x, the algorithm first computes a topological order $T$of the graph in linear time. Then, sets RIGHT_x and LEFT_x are determined, again in linear time via a forward and backward, respectively, execution of the well known breadth-first search (BFS, for short) algorithm, starting from x. This is followed by the removal of x from G. Now, if either RIGHT_x or LEFT_x are empty, the algorithm simply removes all entries that have x as first field in the labeling L, by linearly scanning it, and terminates. Note that, it is very unlikely for RIGHT_x or LEFT_x to be empty, therefore the removal of x from L is done in the trivial way, rather than employing explicitly some data structure storing an inverted index for each label entry in L. Otherwise, the algorithm proceeds in two phases, called *forward update* and *backward update*, that scan vertices that can contain obsolete label entries (namely vertices in LEFT_x and RIGHT_x, respectively) with the purpose of either removing them or updating the associated distances. The two phases are described in details separately in the following sections. At the end of the two, DAG-DECPLL removes $\text{L}_{out}(x)$ and $\text{L}_{in}(x)$ from L and returns the updated label set. In the pseudocode, we denote by $N^G_{out}(v)$ ($N^G_{in}(v)$, respectively) the out-neighbors (in-neighbors, respectively) of the generic vertex v of graph G.

Algorithm 7: Algorithm DAG-DECPLL.

Input: Directed Acyclic Graph G, 2HC-SP labeling L of G, vertex x to be removed from G
Output: Directed Acyclic Graph $G' = G \setminus \{x\}$, 2HC-SP labeling L of $G \setminus \{x\}$

```
1  Compute a topological ordering T of G;
2  Let t(u) be the position of vertex u ∈ V according to T;
3  Compute LEFT_x and RIGHT_x via BFSs;
4  G' ← G \ {x};
5  if LEFT_x = ∅ or RIGHT_x = ∅ then
6  |  Remove all label entries containing x from L;
7  else
8  |  FORWARD(G', L, x, RIGHT_x, LEFT_x);
9  |  BACKWARD(G', L, x, RIGHT_x, LEFT_x);
10 L ← L \ {L_in(x), L_out(x)};
```

6.1. Forward Update

The procedure processes vertices in LEFT_x in decreasing order with respect to a topological ordering $T$of G. Assume we are processing a given vertex, say v. If v has a maximum value in Tas compared to that for the rest of vertices in LEFT_x, then we know by the definition of Tthat no vertex in $N^{G'}_{out}(v)$ belongs to LEFT_x. We also know that a label entry $(h, \delta_{vh}) \in \text{L}_{out}(v)$ may be affected if $h = x$ and $h \in \text{RIGHT}_x$ (see Corollary 1). Moreover, it can be easily seen that, for any vertex $u \in N^{G'}_{out}(v)$ with $u \notin \text{LEFT}_x$, no label entry in $\text{L}_{out}(u)$ is affected by the removal of x from G (see Corollary 2). Additionally, for the rest of cases where $u \in N^{G'}_{out}(v)$ and $u \in \text{LEFT}_x$, by definition of T, u must have been processed before v.

The routine hence proceeds by removing all affected label entries from $\text{L}_{out}(v)$. Notice that, after removing such label entries, we can retrieve the correct distance in the new graph $d_{G'}(v, w)$ for any vertex $w \in V'$ such that $w \notin \text{RIGHT}_x$, by performing a query $\text{QUERY}(v, w, \text{L})$, since the path induced by the labeling does not contain x in these cases. However, to guarantee that the cover property of L is satisfied with respect to all pairs of vertices of the new graph G', we may need to add new label entries to $\text{L}_{out}(v)$ and possibly to backward label sets of vertices in RIGHT_x. To this aim, we exploit the notion of *superset of hubs*, originally presented in Reference [24], and incorporate it in the DAG-DECPLL update procedure after suitably adapting it in order to make it compatible with 2HC-SP labeling.

In more details, the superset of hubs for a forward label $\text{L}_{out}(v)$, denoted by $\text{C}_{out}(v)$, is defined as the union of the hub vertices, belonging to RIGHT_x, in all forward label sets of all vertices in $N^{G'}_{out}(v)$. More formally:

$$\text{C}_{out}(v) = \bigcup_{\forall u \in N^{G'}_{out}(v)} \{k \,|\, (k, \delta_{uk}) \in \text{L}_{out}(u) \wedge h \in \text{RIGHT}_x\}.$$

In the case of reachability labeling one can exploit the notion of superset of hubs to update the reachability properties of a given vertex v: if a neighbor of v is reachable from a given vertex, so is v. Here, instead, we exploit it to simplify the update of the distances stored in the label entries. In details, since we are updating a 2HC-SP labeling, to achieve the update of the label of a given vertex v, we need to compute $d_{G'}(v, h)$ for all $h \in \text{C}_{out}(v)$ and use it to update entries $\delta_{vh} \in \text{L}_{out}(v)$ so that they correspond to distances in the new graph.

One way to do this is to execute a baseline algorithm for computing shortest paths in DAGs. However, even if it is well known that this costs linear time with respect to the graph size, this can easily become a computational bottleneck when dealing with medium to large scale graphs, since we need to compute many distances during an update.

To overcome this limit, we propose a hybrid approach that exploits G' and L to compute distances faster. In more details, it is easy to observe that for any $h \in \text{C}_{out}(v)$ and for any $w \in V'$ such that $w \notin \text{LEFT}_x$, the correct distance $d_{G'}(w, h)$ can be computed via a query on the labeling $\text{QUERY}(w, h, \text{L})$, since the path induced by the labeling from w to h cannot include x (see Lemma 3). Moreover, for any $h \in \text{C}_{out}(v)$ the path between v and h must pass through at least a vertex in RIGHT_x. This implies that, if we have the set of vertices $S = \{v \in V \,|\, v \notin \text{LEFT}_x \wedge \exists u \in N^{G'}_{in}(u) \,:\, u \in \text{LEFT}_x\}$, that are reachable from v in G', then $d_{G'}(v, h)$ is given by the minimum value between $\delta_{vu} + \text{QUERY}(u, h, \text{L})$ among all vertices $u \in S$ (note that δ_{vu} can be retrieved from L). Therefore, to compute $d_{G'}(v, h)$ for all $h \in \text{C}_{out}(v)$, we run a pruned BFS starting from v (see sub-routine shown in Algorithm 8).

Algorithm 8: Algorithm CUSTOMBFS.

Input: Directed acyclic graph G, a vertex s of G, sets RIGHT_x and LEFT_x
Output: Set of pairs of vertices and relative distances from s

```
1  Q ← ∅ ;
2  A_s ← ∅;
3  foreach u ∈ V do
4  |  VISIT[u] ← 0;
5  Q.insert(s, 0);
6  VISIT[s] ← 1;
7  while Q ≠ ∅ do
8  |  (u, δ_vu) ← Q.extractMin();
9  |  VISIT[u] = 2;
10 |  if u ∉ LEFT_x then
11 |  |  A_s ← A_s ∪ {(u, δ_vu)} ;
12 |  else
13 |  |  foreach v ∈ N^G_out(u) do
14 |  |  |  if VISIT[v] = 0 then
15 |  |  |  |  Q.insert(v, δ_vu + w(u, k));
16 |  |  |  else if VISIT[v] = 1 and Q.key(v) > δ_vu + w(u, k) then
17 |  |  |  |  Q.decreaseKey(v, δ_vu + w(u, k));
18 return A_s // List of pairs of vertices in RIGHT_x along with distances from
       s
```

Once all distances are available, we process the vertices in $\text{C}_{out}(v)$ in increasing order with respect to topological sorting. In particular, for each $w \in \text{C}_{out}(v)$ in increasing order of $t(w)$, we update the label entries by using the computed distances, the notion of superset, and the labeling L (see Lines 11–21 of

Algorithm 9). Whenever we add a new label entry or update an existing one, we *mark* the entry so that we keep trace of distances that have already been checked. On top of that, after the first iteration, we exploit the marked query every time we need to check whether a discovered distance d, passing through a vertex, is already encoded in the labeling or not. Finally, notice that, whenever we add a new label entry to the 2HC-SP labeling, we insert it in order to preserve the well-ordered property [26]. This property guarantees that the labeling is minimal in size (i.e., if a single entry is removed, the cover property is broken). To achieve it, vertices are sorted according to any reasonable criterion before the initial preprocessing takes place and, whenever a label entry associated with an hub h has to be added to the label set of a vertex v, this is done if and only if h preceedes v in the established order (we refer the reader to Reference [21,26] for more details). We denote by $l(v)$ the position of a vertex $v \in V$ according to the established order.

Algorithm 9: Procedure FORWARD.

Input: Directed Acyclic Graph G, 2HC-SP labeling L of G, vertex x to be removed from G, sets RIGHT_x and LEFT_x

foreach $v \in \text{LEFT}_x$ *in decreasing order of* $t(v)$ **do**
  $\text{C}_{out}(v) \leftarrow \emptyset$;
  foreach $(h, \delta_{vh}) \in \text{L}_{out}(v)$ **do**
    if $h \in \text{RIGHT}_x$ *or* $h = x$ **then**
      $\text{L}_{out}(v) \leftarrow \text{L}_{out}(v) \setminus \{(h, \delta_{vh})\}$;
  foreach $u \in N^G_{out}(v)$ **do**
    foreach $(h, \delta_{uh}) \in \text{L}_{out}(u)$ **do**
      if $h \in \text{RIGHT}_x$ ***and*** $h \notin \text{C}_{out}(v)$ **then**
        $\text{C}_{out}(v) \leftarrow \text{C}_{out}(v) \cup \{h\}$;
  $A_v \leftarrow \text{CUSTOMBFS}(G, v, \text{RIGHT}_x, \text{LEFT}_x)$;
  foreach $w \in \text{C}_{out}(v)$ *in increasing order of* $t(w)$ **do**
    $d \leftarrow \min_{(u,\delta_{vu}) \in A_v} \{\delta_{vu} + \text{QUERY}(u, w, \text{L})\}$;
    if $\text{MQUERY}(v, w, \text{L}) > d$ **then**
      if $l(v) < l(w)$ **then**
        $\text{L}_{out}(v) \leftarrow \text{L}_{out}(v) \cup \{(w, d)\}$;
      else
        if $(v, \delta_{vw}) \in \text{L}_{in}(w)$ **then**
          $\delta_{vw} \leftarrow d$;
        else
          $\text{L}_{in}(w) \leftarrow \text{L}_{in}(w) \cup \{(v, d)\}$;
          Mark (v, δ_{vw}) in $\text{L}_{in}(h)$;

6.2. Backward Update

The procedure processes vertices in RIGHT_x in increasing order with respect to the same topological ordering $T$of G. Assume we are processing a given vertex, say v. We know that a label entry $(h, \delta_{vh}) \in \text{L}_{in}(v)$ is affected if $h = x$ and $h \in \text{LEFT}_x$ (see Lemma 3). However, in this case, there may be marked label entries, for example, $(h, \delta_{vh}) \in \text{L}_{in}(v)$ such that $h \in \text{LEFT}_x$, that have been added in the forward update phase and that are therefore not considered as affected (they have already been updated). Moreover, it can be easily seen that, for any vertex $u \in N^{G'}_{in}(v)$ with $u \in \text{RIGHT}_x$, no label entry in $\text{L}_{in}(u)$ is affected by the removal of x from G (see again Lemma 3). Additionally, for the rest of cases where $u \in N^{G'}_{in}(v)$ and $u \in \text{RIGHT}_x$, by definition of T, u must have been processed before v. Hence, we proceed by removing all affected entries from $\text{L}_{in}(v)$, where a label entry $(h, \delta_{hv}) \in \text{L}_{in}(v)$ now is affected only if $h = x$ or $h \in \text{LEFT}_x$ and (h, δ_{hv}) is *not marked*.

Notice that, after removing affected label entries from $\mathrm{L}_{in}(v)$, we can compute correct values of $d_{G'}(w, v)$ for any $w \in V'$ such that $w \notin \textsc{left}_x$, via a query $\textsc{query}(w, v, \mathrm{L})$. However, to restore the cover property for other vertices, we may need to add new label entries to $\mathrm{L}_{in}(v)$ and possibly to forward label sets of some of the vertices in $\textsc{left}_x$. To this end, symmetrically to the forward update case, we compute the superset of hubs, this time for the backward label $\mathrm{L}_{in}(v)$, denoted as $\mathrm{C}_{in}(v)$, as the union of the hub vertices, belonging to $\textsc{left}_x$, for all backward label sets of all vertices in $N_{in}^{G'}(v)$, that is:

$$\mathrm{C}_{in}(v) = \bigcup_{\forall u \in N_{in}^{G'}(v)} \{k \mid (k, \delta_{ku}) \in \mathrm{L}_{in}(u) \wedge h \in \textsc{left}_x\}.$$

Observe that, for any $w \in \mathrm{C}_{in}(v)$, the path between w and v in G' must pass through one of the vertex in $N_{in}^{G'}(v)$, by the structure of the DAG G. Moreover, we also know that for any $w \in V'$ and for any $u \in N_{in}^{G'}(v)$, the distance $d_{G'}(w, u)$ can be correctly computed via a query $\textsc{query}(w, v, \mathrm{L})$. In particular, it is given by the minimum value we obtain for $\textsc{query}(w, u, \mathrm{L}) + \textsc{query}(u, v, \mathrm{L})$ among all vertices $u \in N_{in}^{G'}(v)$. If this value is not encoded in the labeling, we add (u, δ_{uv}) to $\mathrm{L}_{in}(v)$ if $l(v) > l(u)$, otherwise we add (v, δ_{uv}) to $\mathrm{L}_{out}(u)$.

We are now ready to discuss on the correcntess of the newly proposed approach.

Theorem 4 (Correctness of DAG-DECPLL)**.** *Let G be a DAG, let L be a* 2HC-SP *labeling of G, and let x be a vertex of G. Let $G' = G \setminus \{x\}$ and L' be the output of algorithm* DAG-DECPLL *when applied to G, L and x. Then: a) $G' = G \setminus \{x\}$ is a DAG and b) L' is a* 2HC-SP *labeling of G'.*

Proof. Concerning (a), the proof is trivial. In fact, if the topological ordering property is true on the arcs of G, then it will hold on G', as we only remove a vertex and its adjacent edges. Regarding (b), we need to show that the cover property holds for all pairs of vertices of the new graph. To this end, first observe that we remove all label entries that induce paths that include the removed vertex x. Then, notice that, in both forward and backward procedures, we test the property for all and only the vertices that are affected by the removal of x (sets $\textsc{right}_x$ and $\textsc{left}_x$) and that the algorithm adds new label entries to vertices by considering them in the order imposed by the topological sorting. The addition of new label entries is done incrementally, by either relying on distances that are: (i) either computed in the new graph via the CUSTOMBFS; or (ii) obtained by combining distances encoded in the labeling that have surely not changed because of the removal of x; or (iii) marked, and hence already updated by previous iterations of the two procedures. □

Theorem 5 (Complexity of DAG-DECPLL)**.** *Algorithm* DAG-DECPLL *takes $O(|V|^3)$ in the worst case.*

Proof. Concerning (a), the proof is trivial. In fact, if the topological ordering property is true on the arcs of G, then it will hold on G', as we only remove a vertex and its adjacent edges. Regarding (b), we need to show that the cover property holds for all pairs of vertices of the new graph. To this end, first observe that we remove all label entries that induce paths that include the removed vertex x. Then, notice that, in both forward and backward procedures, we test the property for all and only the vertices that are affected by the removal of x (sets $\textsc{right}_x$ and $\textsc{left}_x$) and that the algorithm adds new label entries to vertices by considering them in the order imposed by the topological sorting. The addition of new label entries is done incrementally, by either relying on distances that are: (i) either computed in the new graph via the CUSTOMBFS; or (ii) obtained by combining distances encoded in the labeling that have surely not changed because of the removal of x; or (iii) marked, and hence already updated by previous iterations of the two procedures. □

Theorem 6 (Complexity of DAG-DECPLL)**.** *Algorithm* DAG-DECPLL *takes $O(|V|^3)$ in the worst case.*

Proof. Note that, for each vertex in LEFT_x, the algorithm: (i) scans the neighbors and analyzes the label sets of such neighbors (possibly removing some entries); (ii) executes procedure CUSTOMBFS; (iii) processes vertices in $\text{C}_{out}(v)$ and for each one of them possibly performs a marked query. Concerning (i), asymptotically this costs overall quadratic time in the size of G, since the graph is acyclic and the worst case label size is $|V|$. Concerning (ii), again we have an asymptotical time complexity that is quadratic with respect to. $|V|$, since CUSTOMBFS must explore the whole graph in the worst case. Finally, the asymptotical time complexity for executing (iii) can be bounded by observing that vertices in $\text{C}_{out}(v)$ can be at most $|V|$ and that for each of them we may execute a constant number of queries, which take $O(|V|)$ each. Similar considerations can be done to bound the time spent by the algorithm for each vertex in RIGHT_x, with the exception of procedure CUSTOMBFS which is not executed, as vertices in LEFT_x have already been processed. Therefore the claim follows. □

6.3. On Handling Arc Removals or Arc Weight Increases by DAG-DECPLL

In this section, we provide an overview on how to extend DAG-DECPLL to handle arc removals and arc weight increases. In details, given an arc (u, v) to be removed from G, then to update a 2HC-SP labeling L via DAG-DECPLL, we model the removal as the removal of a virtual vertex, say x', having arcs (u, x') and (x', v) in G. In particular, we remove (u, v) from G and run the DAG-DECPLL procedure by considering x' as the vertex to be removed. It is easy to observe that this has the same effect of updating L after the removal of (u, v) from G. It is important to mention that as x' is a virtual vertex in G', therefore no label set exists that is associated to x'. To handle an arc weight increase for a generic arc (u, v), as a first step, we remove (u, v) and then update L using the approach mentioned above. We then insert a new arc (u, v) with the updated arc weight value, and run INCPLL which update L by possibly adding new label entries to L.

6.4. Compacting a Multi-Criteria Public Transit Labeling

In this section, we propose an extension of the notion of stop labeling SL, named *extended stop labeling* (shortly, E-SL), suited for answering to multi-criteria queries. In particular, given a 2HC-SP labeling L of a WRED-TE graph G, we associate to each stop $s_i \in \text{S}$ two sets, namely a *forward stop label* $\text{E-SL}_{out}(i)$ and a *backward stop label* $\text{E-SL}_{in}(i)$ where, in this case, the forward (backward, respectively) stop label is a list of *triples* of the form $(v, stoptime_i(v), transfers_i(v))$ where

- v is a *hub vertex* reachable from (that reaches, respectively) at least one vertex in $\text{DV}[i]$ ($\text{AV}[i]$, respectively);
- $stoptime_i(v)$ encodes the latest departure (earliest arrival, respectively) time to reach hub vertex v from one vertex in $\text{DV}[i]$ (to reach a vertex in $\text{AV}[i]$ from vertex v, respectively);
- $transfers_i(v)$ encodes the minimum number of transfers to reach hub vertex v from one vertex in $\text{DV}[i]$ (to reach a vertex in $\text{AV}[i]$ from vertex v, respectively).

Our approach to compact a labeling for multi-criteria queries is as follows. To compute $\text{E-SL}_{out}(i)$, we process vertices in $\text{DV}[i]$ in decreasing order with respect to departure time. In particular, let v be the vertex under consideration. Then, for each $(h, \delta_{vh}) \in \text{L}_{out}(v)$, we add $(h, stoptime_i(h) = time(v), transfers_i(h) = \delta_{vh})$ to $\text{E-SL}_{out}(i)$ only if one of the following conditions hold:

1. there is no entry $(h, stoptime_i(h), transfers_i(h))$ in $\text{E-SL}_{out}(i)$;
2. there exists an entry $(h, stoptime_i(h), transfers_i(h))$ in $\text{E-SL}_{out}(i)$ but $\delta_{vh} < \text{MT}$ where

$$\text{MT} = \min_{(h, stoptime_i(h), transfers_i(h)) \in \text{E-SL}_{out}(i)} transfers_i(h).$$

To compute $\text{E-SL}_{in}(i)$, symmetrically, we process vertices in $\text{AV}[i]$ in increasing order with respect to arrival times. If v is the vertex under consideration then, for each $(h, \delta_{hv}) \in \text{L}_{in}(v)$, we add $(h, stoptime_i(h) = time(v), transfers_i(h) = \delta_{hv})$ to $\text{E-SL}_{in}(i)$ only if one of the following conditions hold:

1. there is no entry $(h, stoptime_i(h), transfers_i(h))$ in E-SL$_{in}(i)$;
2. there exists an entry $(h, stoptime_i(h), transfers_i(h))$ in E-SL$_{in}(i)$ but $\delta_{hv} <$ MT where

$$\text{MT} = \min_{(h, stoptime_i(h), transfers_i(h)) \in \text{E-SL}_{in}(i)} transfers_i(h).$$

Note that the above second conditions are necessary since, differently from the original stop labeling, here a generic hub h can be added more than once to E-SL$_{in}(i)$ or E-SL$_{in}(i)$, since we might have more paths toward h (or from h) having different number of transfers.

Notice that, for the sake of efficiency, we sort entries in E-SL$_{out}(i)$ and E-SL$_{in}(i)$ with respect to the first, second and third fields, in this order, similarly to what is done in Reference [13] for the stop labeling. The detailed procedure for computing the extended stop labeling is shown in Algorithm 10.

Algorithm 10: Algorithm E-SL Computation.

Input: WRED-TE graph G, 2HC-SP labeling L of G
Output: Extended stop labeling E-SL

foreach $s_i \in$ S **do**
  E-SL$_{in}(i) \leftarrow \emptyset$;
  E-SL$_{out}(i) \leftarrow \emptyset$;
  foreach $v \in V$ **do**
    $Visit[v] = 0$;
    $MT[v] = \infty$;
  foreach $v \in$ AV$[i]$ *in increasing order of arrival time* **do**
    foreach $(h, \delta_{hv}) \in$ L$_{in}(v)$ **do**
      if $Visit[v] = 0$ **then**
        E-SL$_{in}(i) \leftarrow$ E-SL$_{in}(i) \cup \{(h, time(v), \delta_{hv})\}$;
        $MT[v] \leftarrow \delta_{hv}$;
        $Visit[v] \leftarrow 1$;
      else if $MT[v] > \delta_{hv}$ **then**
        E-SL$_{in}(i) \leftarrow$ E-SL$_{in}(i) \cup \{(h, time(v), \delta_{hv})\}$;
        $MT[v] \leftarrow \delta_{hv}$;
  Sort E-SL$_{in}(i)$ with respect to. first, second and third field in each label entry;
  foreach $v \in V$ **do**
    $Visit[v] = 0$;
    $MT[v] = \infty$;
  foreach $v \in$ DV$[i]$ *in decreasing order of departure time* **do**
    foreach $(h, \delta_{vh}) \in$ L$_{out}(v)$ **do**
      if $Visit[v] = 0$ **then**
        E-SL$_{out}(i) \leftarrow$ E-SL$_{out}(i) \cup \{(h, time(v), \delta_{hv})\}$;
        $MT[v] \leftarrow \delta_{vh}$;
        $Visit[v] \leftarrow 1$;
      else if $MT[v] > \delta_{vh}$ **then**
        E-SL$_{out}(i) \leftarrow$ E-SL$_{out}(i) \cup \{(h, time(v), \delta_{vh})\}$;
        $MT[v] \leftarrow \delta_{hv}$;
  Sort E-SL$_{in}(i)$ with respect to. first, second and third field in each label entry;
return E-SL;

6.5. Answering to Multi-criteria Queries via Extended Stop Labeling

For answering a multi-criteria query MC-EA(s_i, s_j, τ) via extended stop labeling, we proceed as follows. Note that E-SL$_{out}(i)$ and E-SL$_{in}(j)$ are arrays sorted with respect to ids, the algorithm as a first step finds the vertex v in E-SL$_{out}(i)$ (E-SL$_{in}(j)$, respectively) whose time is greater than or equal to τ. Assume that said vertex is in position p (q, respectively) in such arrays.

Then, a linear sweep, starting from location p, is performed on SL$_{out}(i)$ to find the first entry $(v, stoptime_i(v), transfers_i(v))$ satisfying the condition that $stoptime_i(v) \geq \tau$. Let us assume this entry is stored in location $p' \geq p$. This part of the computation is known as the process of computing *relevant hubs* and it is followed by the computation of all hubs that are both E-SL$_{out}(i)$ and E-SL$_{in}(j)$, stored at locations greater than p' and q in E-SL$_{out}(i)$ and E-SL$_{in}(j)$, respectively. We then perform a linear sweep on both E-SL$_{out}(i)$ and E-SL$_{in}(j)$ starting from p' and q, respectively. While performing the linear sweep, we add the corresponding journey for each matched hub we found to a temporary set, say M. Once M is computed, we order the journeys in M with respect to their arrival times. Finally, we process journeys in M sequentially, and add a journey J to PROFILE only if the accumulated number of transfers in J is less than the number of transfers for journeys added so far to PROFILE. Finally, we return PROFILE as the answer to the profile query MC-EA(s_i, s_j, τ).

6.6. Updating the Extended Stop Labeling

If an extended stop labeling E-SL is available and the network undergoes a delay then, after updating both the WRED-TE graph and the 2HC-SP labeling, the trivial way to update E-SL is to recompute it from scratch. However, to reduce the required computational effort, in what follows we propose a procedure that is able to exploit the information about the changed part of the graph and the 2HC-SP labeling to update the corresponding extended stop labeling in very short time.

In details, the procedure for dynamically updating the extended stop labeling requires, during the execution of Algorithms 7–11, to compute two sets of so–called *updated stops*, denoted, respectively, by US$_{out}$ and US$_{in}$. In the multi-criteria case these two sets are defined as the stops $s_i \in$ S such that vertices in DV$[i]$ (AV$[i]$, respectively) had their time value or forward label (backward label, respectively) changed during Algorithm DAG-DECPLL. Once this is done we update the extended stop labeling E-SL by recomputing only those entries of E-SL$_{out}(i)$ (E-SL$_{in}(i)$, respectively) for each $s_i \in$ US$_{out}$ (for each $s_i \in$ US$_{in}$, respectively).

We are now ready to provide the following results.

Theorem 7 (Correctness of Multi-criteria D-PTL)**.** *Given an input timetable and a corresponding* WRED-TE *graph G. Let* L *be a* 2HC-SP *labeling of G and let* E-SL *be an extended stop labeling associated to* L *Assume* $\delta > 0$ *is a delay occurring on a connection, i.e. an increase of* δ *on its departure time. Let* G', L$'$, *and* E-SL$'$ *be the output of* D-PTL *when applied to G,* L *and* E-SL*, respectively, by considering the delay, in the multi-criteria setting. Then: (i)* G' *is a* WRED-TE *graph for the updated timetable; (ii)* L$'$ *is a* 2HC-SP *labeling for* G'*; (iii)* E-SL$'$ *is an extended stop labeling for* L$'$.

Proof. The correctness of D-PTL in the multi-criteria case is based on that of the approach in [15] and on Theorem 4. In particular observe that, whenever we update the graph, we do it by preserving the constraints imposed by the WRED-TE model on both vertices, by suitably modifying connection arcs and associated waiting, bypass, and transfer arcs. In more details, it is easy to prove, by contradiction, that after the execution of the D-PTL algorithm for the multi-criteria case, G is a WRED-TE graph. Concerning the labeling data structures, observe that after each change to G we use either DAG-DECPLL or INCPLL, depending on the type of performed modification. These algorithms have been shown to compute a labeling that is a 2HC-SP labeling for the modified graph (see Theorem 4 or the proof in [24]). Hence, at the end of Algorithm 2, L is a 2HC-SP labeling for G. Finally, note that the algorithm described in Section 6.6 applies the definition of extended stop labeling, by updating the entry of a stop

with the proper hub vertices, times and distances values. Hence, after the execution of algorithm described in Section 6.6, E-SL is still an extended stop labeling of L. □

Algorithm 11: Procedure BACKWARD.

Input: Directed Acyclic Graph G, 2HC-SP labeling L of G, vertex x to be removed from G, sets LEFT_x and RIGHT_x.

```
1  foreach v ∈ RIGHT_x in increasing order of t(v) do
2      C_out(v) ← ∅ ;
3      foreach (u, δ_uv) ∈ L_in(v) do
4          if u ≠ v then
5              if (u ∈ LEFT_x and (u, δ_uv) is not marked) or u = x then
6                  L_in(v) ← L_in(v) \ {(u, δ_uv)};
7      foreach u ∈ N^G_in(v) do
8          foreach (h, δ_hv) ∈ L_in(u) do
9              if h ∈ LEFT_x and h ∉ C_in(v) then
10                 C_in(v) ← C_in(v) ∪ {h} ;
11     foreach w ∈ C_in(v) in increasing order of t(w) do
12         d ← min_{u∈N^G_in(v)} {QUERY(w, u, L) + QUERY(u, v, L)} ;
13         if QUERY(w, v, L) > d then
14             if l(v) < l(w) then
15                 L_in(v) ← L_in(v) ∪ {(w, d)};
16             else
17                 L_out(w) ← L_out(w) ∪ {(v, d)};
```

Theorem 8 (Complexity of Multi-criteria D-PTL)**.** *Algorithm* D-PTL *in the multi-criteria setting takes* $\mathcal{O}(|\mathcal{C}|^4)$ *computational time in the worst case.*

Proof. The proof can be derived by the argument given in the proof of Theorem 2. In particular, we know that the worst case time complexity of both INCPLL and DAG-DECPLL is $\mathcal{O}(|V|^3)$ for a graph with $|V|$ vertices and that these routines, in Algorithm D-PTL, can be executed, in the worst case, for all stops, which are $|\mathrm{S}| \leq |\mathcal{C}|$. Since $|V| \in \mathcal{O}(|\mathcal{C}|)$ for any WRED-TE graph, the claim follows. □

7. Experimental Study

In this section, we present our experimental study to assess the performance of D-PTL. In particular, we implemented, in C++, both PTL and D-PTL, and developed a simulation environment to evaluate the two algorithms on given input transit networks. Our entire framework is based on NetworKit [30], a widely adopted open-source toolkit for graph algorithms and interactive large-scale network analysis. Our code has been compiled with GNU g++ v.4.8.5 (O3 opt. level) under Linux (Kernel 4.4.0-148) and all tests have been executed on a workstation equipped with an Intel Xeon© CPU and 128 GB of main memory.

7.1. Experimental Setup

Our experimental evaluation is divided in two parts. The first part deals with the basic version of PTL and single criterion queries and thus aims at evaluating the performance of D-PTL in its basic version, that updates the RED-TE graph, the 2HC-R labeling and the stop labeling. The second part, on the other hand, focuses on the multi-criteria version of PTL and hence on the performance of D-PTL in this latter case, that has to update the WRED-TE graph, the 2HC-SP labeling and the extended stop labeling. We remark that in this paper we provide a compacted version of the data structure used by multi-criteria PTL, namely the *extended stop labeling*, and a new dynamic (decremental) algorithm for

updating 2HC-SP labelings in DAGS. Both are experimentally evaluated to assess their performance in terms of query time and update time, respectively.

Our experimental study is structured as follows: depending on the considered setting (either basic or multi-criteria) for each input, we build either the RED-TE graph G or the WRED-TE one, and execute PTL to compute corresponding labelings, namely:

- in the basic case: a 2HC-R labeling L, and a stop labeling SL of G;
- in the multi-criteria case: a 2HC-SP labeling L, and an extended stop labeling E-SL of G.

Then, we select a connection c_j of the timetable uniformly at random and delay it by δ minutes, where δ is randomly chosen within $[5, time(m) - time(v_d^{c_j}) + 10]$ and $m = argmax_{v \in \mathrm{DV}[s]} time(v)$. Choosing δ in such a way ensures the occurrence of all meaningful cases, that is the corresponding departure and arrival vertices can be shifted through the whole set of departure and arrival vertices of the corresponding stop.

Finally, we run D-PTL to update both the graph and the labelings. In particular, the specific version to handle either the basic or the multi-criteria setting are executed. In parallel, we run PTL to recompute graph and labelings from scratch (again, we recompute the specific basic or multi-criteria version). After each execution, we measure both the *update time* of D-PTL and the computational time taken by PTL for the recomputation from scratch.

Moreover, we also measure the *average size of the labelings* and the *average query time*. The former is the average space occupancy, in megabytes, of the different labelings employed by the approaches. The latter, instead, is obtained by computing the average time to answer to 100,000 queries, of both earliest arrival, profile and multi-criteria type via the corresponding query algorithms described in Sections 5 and 6. This is done to evaluate the quality of the data structures when updated via D-PTL against that of the data structures recomputed from scratch and to show that using D-PTL to update the graph and the labelings does not affect the performance of the framework. The two most important quality metrics in this context are space occupancy and query time (which are also somehow related). Note that, for the above queries, stops and departure times (ranges for profile queries, respectively) are chosen uniformly at random. For each query, for the sake of validity, we compare the result by comparing the two outputs with the result of an exhaustive Dijkstra's-like visit on the graph [9]. We repeat the above process for 50 connections, in order to compute average values and collect statistically significant results.

As inputs to our experiments, we considered, as other studies of this kind [2,4,10,13], real-world transit networks whose data is publicly available (Public Transit Feeds Archive—https://transitfeeds.com/.) and is formatted according to the *General Transit Feeds Specification* (shortly, GTFS). In particular, GTFS is a data specification standard for transit datasets, which enforces uniformity in the structure of data coming from different sources in order to be consumed by a wide variety of software applications, such as journey planners. For more details about GTFS, see https://gtfs.org/ and https://developers.google.com/transit/gtfs/. Details on the used inputs for basic PTL are given in Table 2 while those of the inputs considered for the multi-criteria setting are given in Table 3.

We remark here that we were forced to use smaller inputs in the latter case with respect to the basic one, since: (i) the preprocessing phase is much more time consuming with respect to basic case; and (ii) the labelings require several GB of main memory to be stored. Hence, we were unable to test on the same large instances considered for the basic setting due to the limitations of our hardware.

In each table we report, for each network, the number of stops, the size of the corresponding RED-TE graph (WRED-TE graph, respectively) in terms of vertices and arcs, the time for preprocessing the network to compute the labelings L and SL (either 2HC-R and stop labeling or 2HC-SP and extended stop labeling, respectively). Finally, we report of both L and SL, in megabytes.

Table 2. Details of input datasets for the basic setting: preprocessing time is expressed in seconds, labeling size in megabytes.

Network	# Stops	Graph		Preprocessing Time		Labeling Size	
		$\|V\|$	$\|A\|$	L	SL	L	SL
London	5221	3,066,852	5,957,246	4494.00	5.19	5856	529
Madrid	4698	3,971,870	7,859,375	10,559.10	13.66	12,295	2653
Rome	9273	5,502,796	10,893,752	17,081.05	30.18	18,531	5262
Melbourne	27,237	9,757,352	18,389,454	3774.00	12.79	8293	1136

Table 3. Details of input datasets for the multi-criteria setting: preprocessing time is expressed in seconds, labeling size in megabytes.

Network	# Stops	Graph		Preprocessing Time		Labeling Size	
		$\|V\|$	$\|A\|$	L	E-SL	L	E-SL
Palermo	1714	563,064	1,112,110	7828.00	3.43	4687	372
Barcelona	3232	1,201,256	2,075,005	14,207.00	7.99	4219	359
Luxembourg	2802	1,239,870	2,438,413	42,701.70	11.75	30,491	1129
Prague	4940	1,755,078	2,475,801	29,288.60	18.57	4243	694
Venice	2173	1,373,674	2,526,500	19,114.03	5.87	7426	189

7.2. Analysis

The main results of our experiments are summarized in Tables 4 and 5, where we report the average time taken by D-PTL to update L and SL, respectively (cf 2nd and 3rd columns), the average time taken by PTL for recomputing from scratch L and SL, respectively, (cf 4th and 5th columns) and the average speed-up obtained by using D-PTL instead of PTL (cf 6th column). This is given by the ratio of the average total time taken by PTL to the average total update time of D-PTL.

Table 4. Comparison between D-PTL and PTL in the basic setting, in terms of computational time. The first column shows the considered network while the 2nd and the 3rd columns show the average time taken by D-PTL to update the labeling and the stop labeling, respectively, after a delay occurs in the network. The 4th and the 5th columns show the average time taken by PTL to recompute from scratch the labeling and the stop labeling, respectively, after a delay occurs in the network. Finally, the 6th column shows the speed-up, that is the ratio of the sum of the values in the 2nd and the 3rd columns to the sum of the values in the 4th and the 5th columns.

Network	(Basic) D-PTL Avg. Update Time (seconds)		(Basic) PTL Avg. Reprocessing Time (seconds)		Speed-up
	L	SL	L	SL	
London	8.64	2.48	4417.65	5.50	397.77
Madrid	17.47	7.76	10,495.40	14.20	416.55
Rome	12.36	14.49	16,847.00	29.50	628.55
Melbourne	4.08	7.25	3807.00	11.50	337.03

Table 5. Comparison between D-PTL and PTL in the multi-criteria setting, in terms of computational time. The first column shows the considered network while the 2nd, the 3rd, and the 4th columns show the average time taken by D-PTL to update the labeling and the extended stop labeling, respectively, after a delay occurs in the network. The time taken to update the labeling, in this case, is divided in two fields to highlight which of the two components of D-PTL is more time consuming. The 5th and the 6th columns show the average time taken by PTL to update the WRED-TE graph and recompute from scratch the labeling and the extended stop labeling, respectively, after a delay occurs in the network. Finally, the 7th column shows the speed-up, that is the ratio of the sum of the values in the 2nd, 3rd and 4th columns to the sum of the values in the 5th and the 6th columns.

Network	(Multi-Criteria) D-PTL Avg. Update Time (seconds)			(Multi-Criteria) PTL Avg. Reprocessing Time (seconds)		Speed-up
	L		E-SL	L	E-SL	
	INCPLL	DAG-DECPLL				
Palermo	210.37	135.86	3.01	7807.14	3.39	22.36
Barcelona	25.19	2.50	5.65	14,156.90	7.62	424.85
Luxembourg	617.44	348.92	6.57	43,275.70	11.50	44.49
Venice	167.67	3.10	3.50	19,146.60	5.69	109.90
Prague	597.93	22.92	10.16	29,435.90	20.84	40.86

In both cases we observe that D-PTL is able to update the labeling (either 2HC-R or 2HC-SP) and the (extended) stop labeling in a time that is always more than an order of magnitude smaller than that taken by the recomputation from scratch via PTL (up to more than 600 times smaller). This is true in both the basic and the multi-criteria cases. In this latter case, we notice that the newly proposed decremental algorithm DAG-DECPLL is very effective, since it is tailored for DAGs, and is always faster than the general incremental algorithm INCPLL, which is designed to work for any graph. This is somehow a novel result with respect to Reference [21], where DECPLL is always by far slower than INCPLL, that might drive further investigation on updating 2HC-SP labeling in general graphs. Furthermore, the experiments show that graphs and labelings updated via D-PTL and those recomputed from scratch via PTL are equivalent in terms of both query time and space overhead (cf Tables 6–9). In particular, both sizes and query times are very similar, as expected, thus suggesting that the use of D-PTL does not induce any degradation in the performance of the data structures. This is most likely due to the fact that D-PTL preserves by design the minimality of the labeling, an important property that has been shown to be tied to performance in labelings [20,21,26]. On top of that, a further consideration that can be done by analyzing the data in Table 8 is that the newly proposed extended stop labeling is at least as effective as the original stop labeling in accelerating the query algorithm and reducing the corresponding query time in the multi-criteria case (see Reference [13] for a detailed comparison with the reduction provided by the original stop labeling).

Table 6. Comparison between D-PTL and PTL in the basic setting, in terms of query time. The first column shows the considered network. The 2nd and 4th columns (3rd and 5th, respectively) show the average computational time for performing an earliest arrival (profile, respectively) query. In particular, columns 2nd and 3rd refer to average query times obtained from the labelings updated via D-PTL, while columns 4th and 5th refer to those obtained from the labelings recomputed from scratch via PTL.

Network	(Basic) D-PTL Avg. Query Time (milli-seconds)		(Basic) PTL Avg. Query Time (milli-seconds)	
	EAQ	PQ	EAQ	PQ
London	0.01	0.10	0.01	0.14
Madrid	0.03	0.35	0.03	0.34
Rome	0.04	0.18	0.04	0.19
Melbourne	0.05	0.26	0.06	0.27

Table 7. Comparison between D-PTL and PTL in the basic setting, in terms of space overhead. The first column shows the considered network. The 2nd and the 3rd columns show the average size of the 2HC-R labeling and the stop labeling, respectively, updated via D-PTL. The 4th and the 5th columns, instead, show the average size of the 2HC-R labeling and the stop labeling, respectively, when recomputed from scratch via PTL.

Network	(Basic) D-PTL Avg. Space (MB)		(Basic) PTL Avg. Space (MB)	
	L	SL	L	SL
London	5894	533	5902	532
Madrid	12,391	2669	12,395	2663
Rome	18,531	5260	18,512	5252
Melbourne	8298	1140	8300	1138

To summarize, all the above observations and data give a strong evidence of the following facts:

- D-PTL is a very effective and practical option for journey planning when dynamic, delay-prone networks have to be handled, especially when they are of very large size and yet require fast query answering;
- DAG-DECPLL is a prominent solution to update 2HC-SP labelings in DAGs, faster than algorithm DECPLL, which is designed for general graphs;
- the newly proposed extended stop labeling is an effective compact version of the data structures used by PTL in the multi-criteria case that allows a significant reduction in the average query time.

Table 8. Comparison between D-PTL and PTL in the multi-criteria setting, in terms of query time. The first column shows the considered network. The 2nd and the 4th (3rd and 5th, respectively) columns show the average computational time for performing a multi-criteria query by the two approaches without (with, respectively) extended stop labelings. Columns 2nd and 3rd refer to average query times obtained from the labelings updated via D-PTL, while columns 4th and 5th refer to those obtained from the labeling recomputed from scratch, respectively.

Network	(Multi-criteria) D-PTL Avg. Query Time (milli-seconds)		(Multi-criteria) PTL Avg. Query Time (milli-seconds)	
	MC-EA	MC-EA with E-SL	MC-EA	MC-EA with E-SL
Palermo	1.25	0.52	1.25	0.53
Barcelona	0.09	0.01	0.08	0.01
Luxembourg	2.64	1.10	2.66	1.10
Venice	0.81	0.06	0.76	0.06
Prague	2.93	0.03	3.54	0.35

Table 9. Comparison between D-PTL and PTL in the multi-criteria setting, in terms of space overhead. The first column shows the considered network. The 2nd and the 3rd columns show the average size of the 2HC-SP labeling and the extended stop labeling, respectively, when updated via D-PTL, while the 4th and the 5th columns show the average size of the 2HC-SP labeling and the extended stop labeling, respectively, when recomputed from scratch. Note that, regarding the extended stop labeling (cf 3rd and 5th column), the update is done by the procedure given in this paper (cf Section 6.6) while the recomputation from scratch is done by Algorithm 10, that is also not originally included in the PTL framework.

Network	(Multi-Criteria) D-PTL Avg. Space (MB)		(Multi-Criteria) PTL Avg. Space (MB)	
	L	E-SL	L	E-SL
Palermo	4764	372	4737	373
Barcelona	4253	360	4219	360
Luxembourg	30,557	1129	30,519	1126
Venice	7443	190	7413	190
Prague	4250	694	4251	695

8. Conclusions

In this paper we have studied the journey planning problem in the context transit networks, with a specific focus on tolerance to disruptions and scalability. The problem asks to answer to various types of queries, seeking journeys exhibiting optimality with respect to. different metrics, on suitable data structures representing *timetables* of *schedule-based transportation system* (consisting of buses, trains, and trams, for example).

We have analyzed the state-of-the-art solution, in terms of query time, for this problem, that is *Public Transit Labeling* (PTL). We have attacked what can be considered the main limitation of this preprocessing-based approach, that is not being natively designed to tolerate updates in the schedule, which are instead very frequent in real-world applications. We have hence introduced a new framework, called D-PTL, that extends PTL to function under delays. In particular, we have provided a new algorithm able to update the employed data structures efficiently whenever a delay affects the network, without performing any recomputation from scratch. We have demonstrated the effectiveness of our new solution through an extensive experimental evaluation conducted on real-world networks. Our experiments show that the time required by the new algorithm is, on average, always at least an order of magnitude smaller than that required by the recomputation from scratch, in

both flavours of PTL, that is basic and multi-criteria. As byproducts of our investigation, to handle the multi-criteria case we have presented: (i) a new algorithm for updating 2HC-SP labelings in directed acyclic graphs as a consequence of decremental updates and (ii) a new compact version of the data structure employed by PTL in the multi-criteria setting. Concerning (i), the new method has been shown to be, empirically, much faster than the only known solution DECPLL [21]. For the sake of fairness, we recall that the latter works also for general graphs. Regarding (ii), we have provided strong experimental evidences of the effectiveness of the compact representation in reducing the required query times.

Several research directions deserve further investigation. Perhaps the most relevant one is to extend the experimentation to larger and more diverse inputs, to strengthen the obtained conclusions. Another line of research that might be pursued could be that of designing an improved version of the proposed solution able to provide higher speedups, especially for those networks where D-PTL exhibits a speedup in the order of few tens. This could require a more refined analysis of D-PTL performance and of its relationship with the structure of the pathological inputs.

Author Contributions: All authors have equally contributed to this manuscript.

Funding: This work has been partially supported by the Italian National Group for Scientific Computation GNCS-INdAM—Program "Finanziamento GNCS Giovani Ricercatori 2018/2019"—Project "Efficient Mining of Distances in Fully Dynamic Massive Graphs".

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bast, H.; Delling, D.; Goldberg, A.V.; Müller-Hannemann, M.; Pajor, T.; Sanders, P.; Wagner, D.; Werneck, R.F. Route Planning in Transportation Networks. In *Algorithm Engineering—Selected Results and Surveys*; Lecture Notes in Computer Science; Kliemann, L.; Sanders, P., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9220, pp. 19–80.
2. Cionini, A.; D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Giannakopoulou, K.; Paraskevopoulos, A.; Zaroliagis, C.D. Engineering graph-based models for dynamic timetable information systems. *J. Discret. Algorithms* **2017**, *46-47*, 40–58.
3. Delling, D.; Goldberg, A.V.; Pajor, T.; Werneck, R.F. Customizable Route Planning in Road Networks. *Transp. Sci.* **2017**, *51*, 566–591.
4. Dibbelt, J.; Pajor, T.; Strasser, B.; Wagner, D. Connection Scan Algorithm. *J. Exp. Algorithmics* **2018**, *23*, 1–7.
5. Delling, D.; Pajor, T.; Werneck, R.F. Round-Based Public Transit Routing. *Transp. Sci.* **2015**, *49*, 591–604.
6. Wagner, D.; Zündorf, T. Public transit routing with unrestricted walking. In Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017), Vienna, Austria, 7–8 September 2017; Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Saarbrücken/Wadern, Germany, 2017.
7. Delling, D.; Dibbelt, J.; Pajor, T.; Zündorf, T. Faster transit routing by hyper partitioning. In Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017), Vienna, Austria, 7–8 September 2017; Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Saarbrücken/Wadern, Germany, 2017.
8. Delling, D.; Dibbelt, J.; Pajor, T. Fast and Exact Public Transit Routing with Restricted Pareto Sets. In Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX), SIAM, San Diego, CA, USA, 7–8 January 2019; pp. 54–65.
9. Pyrga, E.; Schulz, F.; Wagner, D.; Zaroliagis, C. Efficient models for timetable information in public transportation systems. *ACM J. Exp. Algorithmics* **2008**, *12*, 1–39.
10. Cionini, A.; D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Giannakopoulou, K.; Paraskevopoulos, A.; Zaroliagis, C.D. Engineering Graph-Based Models for Dynamic Timetable Information Systems. In Proceedings of the Proceedings of 14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS14), Wroclaw, Poland, 11 September 2014; Schloss Dagstuhl: Saarbrücken/Wadern, Germany, 2014; Volume 42, pp. 46–61.

11. Giannakopoulou, K.; Paraskevopoulos, A.; Zaroliagis, C. Multimodal Dynamic Journey-Planning. *Algorithms* **2019**, *12*, 213.
12. Witt, S. Trip-Based Public Transit Routing. In *Algorithms–ESA 2015*; Bansal, N., Finocchi, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2015; pp. 1025–1036.
13. Delling, D.; Dibbelt, J.; Pajor, T.; Werneck, R.F. Public Transit Labeling. In *International Symposium on Experimental Algorithms (SEA15)*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9125, pp. 273–285.
14. Wang, S.; Lin, W.; Yang, Y.; Xiao, X.; Zhou, S. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD15), ACM, Melbourne, Australia, 31 May–4 June 2015; pp. 967–982.
15. Akiba, T.; Iwata, Y.; Yoshida, Y. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In Proceedings of the 23rd International World Wide Web Conference (WWW14), ACM, Seoul, Korea, 7–11 April 2014; pp. 237–248.
16. Cicerone, S.; D'Emidio, M.; Frigioni, D. On Mining Distances in Large-Scale Dynamic Graphs. In Proceedings of the 19th Italian Conference on Theoretical Computer Science (ICTCS18), Urbino, Italy, 18–20 September 2018; Volume 2243, pp. 77–81.
17. D'Angelo, G.; D'Emidio, M.; Frigioni, D. Fully dynamic update of arc-flags. *Networks* **2014**, *63*, 243–259.
18. D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Vitale, C. Fully Dynamic Maintenance of Arc-flags in Road Networks. In *International Symposium on Experimental Algorithms*; Lecture Notes in Computer Science; Springer, Berlin/Heidelberg, Germany, 2012; Volume 7276, pp. 135–147.
19. D'Andrea, A.; D'Emidio, M.; Frigioni, D.; Leucci, S.; Proietti, G. Experimental Evaluation of Dynamic Shortest Path Tree Algorithms on Homogeneous Batches. In *International Symposium on Experimental Algorithms*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2014; Volume 8504, pp. 283–294.
20. D'Angelo, G.; D'Emidio, M.; Frigioni, D. Distance Queries in Large-Scale Fully Dynamic Complex Networks. In *International Workshop on Combinatorial Algorithms*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2016; Volume 9843, pp. 109–121.
21. D'Angelo, G.; D'Emidio, M.; Frigioni, D. Fully Dynamic 2-Hop Cover Labeling. *J. Exp. Algorithmics* **2019**, *24*, 1–6.
22. Qin, Y.; Sheng, Q.Z.; Falkner, N.J.G.; Yao, L.; Parkinson, S. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *World Wide Web* **2017**, *20*, 915–937.
23. D'Emidio, M.; Khan, I. Dynamic Public Transit Labeling. In Proceedings of the International Conference on computational Science And Its Applications, Saint Petersburg, Russia, 1–4 July 2019; Volume 11619, pp. 103–117.
24. Zhu, A.D.; Lin, W.; Wang, S.; Xiao, X. Reachability queries on large dynamic graphs: a total order approach. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD14), ACM, Snowbird, UT, USA, 22–27 June 2014; pp. 1323–1334.
25. Warburton, A. Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems. *Oper. Res.* **1987**, *35*, 70–79, doi:10.1287/opre.35.1.70.
26. Cohen, E.; Halperin, E.; Kaplan, H.; Zwick, U. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* **2003**, *32*, 1338–1355.
27. Cheng, J.; Huang, S.; Wu, H.; Fu, A.W.C. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD13), New York, NY, USA, 22–27 June 2013; pp. 193–204.
28. Yano, Y.; Akiba, T.; Iwata, Y.; Yoshida, Y. Fast and Scalable Reachability Queries on Graphs by Pruned Labeling with Landmarks and Paths. In Proceedings of the 22nd ACM International Conference on Information & Knowledge Management (CIKM13), ACM, San Francisco, CA, USA, 27 October–1 November 2013; pp. 1601–1606.
29. Colella, F.; D'Emidio, M.; Proietti, G. Simple and Practically Efficient Fault-tolerant 2-hop Cover Labelings. In Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic, Naples, Italy, 26–28 September 2017; Volume 1949, pp. 51–62.

30. Staudt, C.L.; Sazonovs, A.; Meyerhenke, H. NetworKit: A tool suite for large-scale complex network analysis. *Netw. Sci.* **2016**, *4*, 508–530, doi:10.1017/nws.2016.20.