PhD Thesis

Journey Planning in Delay Prone Public Transit Networks

PhD Program in Computer Science: XXXI cycle

Author:

Imran Khan
imran.khan@gssi.it

Advisor:

Asst. Prof. Mattia D'Emidio
mattia.demidio@univaq.it

April 27, 2020

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

GSSI Gran Sasso Science Institute
Viale Francesco Crispi, 7 - 67100 L'Aquila - Italy

Declaration of Authorship

I, Imran KHAN, declare that this thesis titled, 'Journey Planning in Delay Prone Public Transit Networks' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this Institute.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

Imran Khan

Date: 27-Apr-20

*Dedicated to my **deceased father** Mr. Abdul Salam (may Allah be pleased with him), who effectively shared his vision of enlightenment-through-education with us. He sacrificed himself while removing all the barriers in-between us and the light of education.*

---

*Dedicated to my **Mother**, who, despite being a lonely mother responsible for raising her children, never thought about closing the door of education for us, even though it was hard to keep going.*

---

*Dedicated to my **Maternal-Grandparents**, who, for years, shared their bread with me along with my siblings, and encouraged us all the time in all possible ways.*

Acknowledgements

I am thankful to the GSSI PhD selection committee, back in 2015, who allowed me to begin my PhD in Computer Science. In particular, I am thankful to Prof. Rocco De Nicola, Prof. Michele Falmmini, Prof. Luca Aceto and Antonia Bertolino. I am also thankful to all of the faculty members who introduced me and our colleagues to known researchers (thanks to all of them) in the field by arranging seminars and guest lectures.

I am thankful to my PhD Supervisor Prof. Mattia D'Emidio, who accepted my request to supervise my PhD thesis at the time when others left me, and literally pushed me to say *good-bye* to GSSI. His guidance, kind-support, and advice made my thesis work possible, and I am deeply indebted with him. Not only these, he also encouraged me throughout my stay at GSSI, and increased my motivation despite the rejections we received for most of our articles.

I would like to thank the administration of GSSI, especially the people responsible for providing IT support, for their kindness and willingness to help me with everything I needed. Moreover, I would like to thank the other co-authors of some works which this thesis is based on: Prof. Daniele Frigioni, University of L'Aquila, Italy.

I would like to thank all of my colleagues, who helped me in every possible situation, and made themselves available all the time.

Last but not least, I am grateful to my family, especially to my wife Ms. *Nazia Iqbal*, who increased my motivation towards getting the PhD title, and to my elder sister Ms. *Rohi Ijaz*, who, despite of being in a man dominant society, stood besides me like a father.

Imran Khan

Abstract

In this thesis, we study 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, that is it asks to answer to natural queries such as, e.g., "find a journey that starts from a given source stop and arrives at a given target stop as early as possible".

The solution to this problem exhibiting the smallest query times is the algorithmic framework named *Public Transit Labeling* (PTL), proposed for the first time in [Delling et al., SEA 2015]. Said method consists of 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 preprocessing); 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. Contrary to this, there are solutions to handle dynamic timetables, however, their query times are very high as compared to that of PTL.

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 handle the journey planning problem in delay-prone transit networks.

List of Publications

In the following, the list of the author's publications which some chapters of this doctoral dissertation are based on:

1. Mattia D'Emidio, Imran Khan, "Dynamic Public Transit Labeling." In International Conference on Computational Science and Its Applications, pp. 103-117. Springer, Cham, 2019, https://link.springer.com/chapter/10.1007/978-3-030-24289-3_9

2. Mattia D'Emidio, Imran Khan, Daniele Frigione, "Journey Planning Algorithms for Massive Delay-prone Transit Networks", Algorithms, 13(12), 2020.

Contents

List of Figures

List of Tables

1

Introduction

Computing "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 [9]. In particular, the *journey planning problem* takes as input a *timetable* (or *schedule*), i.e. the description, in terms of departure and arrival times, of transits of vehicles between stops within the system, and asks 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 such problem constitutes a fundamental primitive in the world of information technologies and intelligent transport systems since, nowadays, millions of people use computer-based journey planning information systems to obtain public transit directions. Most of these systems use modern algorithms developed by applied algorithms researchers in the last couple of decades.

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 [9,29,44], 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 [9]. Concerning the model/data structure, the schedule normally is stored normally in the form of arrays [52] or via a graph based representation [28].

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*, i.e. 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 [29]. In this work, we focus on the realistic scenario, which is much more meaningful from an application viewpoint.

1.1 Motivation

The fastest solutions to the journey planning problem with respect to query time are those in [41,99]. Of them, the one in [41] relies on an algorithmic framework referred to, throughout this document, as *Public Transit Labeling* (PTL, for short). Such 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 [47], CSA [52] 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 [41].

In more details, PTL consists of three main ingredients: i) the well-known *time-expanded graph* model to store transit networks (see e.g. [83]); 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. Furthermore, specific to earliest arrival and profile queries, PTL has a compact form of labeling tailored for reducing the query times and memory requirements.

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 [41]. 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 labeling-based representations of transitive closures have been investigated in the past, in other application domains, due to the effectiveness of such structures to retrieve graph properties [2, 27, 35–37, 86]. 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.

1.2 Our Contribution

In this work, 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. Note that, although decreases in departure times are typically not allowed in transit networks [9], hence updating the information in such case would not be necessary, our solution can be easily extended to manage such scenario. The algorithm is based on suited combinations of graph update routines inspired to those in [9] and labeling dynamic algorithms, extensions of those in [2, 37, 105]. 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.

Specific to the multi-criteria queries, we present an algorithm for computing a compact form of the labeling employed by the multi-criteria PTL. We also present an approach for answering multi-criteria queries using our compact labeling, and discuss its correctness.

Finally, we present an algorithm for dynamically updating the compact labeling we compute for answering multi-criteria queries.

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. Furthermore, our results show that answering multi-criteria queries using our compact labeling results in reducing the query times by orders of magnitude as compared to the query times obtained using the original multi-criteria PTL.

1.3 Dissertation Organization

This document is organized as follows. Chapter 2 gives the basic notation and the definitions related to graph. It also describes techniques, based on preprocessing graphs, for efficiently answering queries about the structure of a graph. Chapter 3 formally describes the journey planning problem, and its variatns. It also describes models/data structures for storing a schedule in memory, and provides an overview of the state of art approaches for solving the journey planning problem. Chapter 4 describes the dynamic journey planning problem, and discuss relevant approaches that can be used for solving the dynamic journey planning problem. Chapter 5 present our new dynamic algorithms, in the basic and multi-criteria versions, respectively, and discusses the correctness and complexity of the new methods. It also describes the working of the new dynamic algorithms applied to example scenarios. Chapter 6 provides a description of the experiments we conducted for showing the effectiveness of our new dynamic algorithms. Finally, Chapter 7 provides conclusive remarks and outlines possible future research directions.

2

Background

In this Chapter, we provide definitions, notations, and terminologies related to graph, that are important for understanding the problem we investigated in this thesis, more precisely to the domain of *algorithmic frameworks for public-transit networks*. In particular, Section 2.1 introduces graphs and related notions, while Section 2.2 introduces preprocessing techniques for computing a preprocessed structure on top of a graph.

2.1 Graphs

A graph $G = (V, A)$ consists of a pair of a set V of vertices and a set A of arcs: vertices in V are *connected* by arcs in A, that is arcs are pairs of vertices expressing a relation of connection between them.

A graph $G = (V, A)$ is *weighted* if there exists a weight function $w : A \to \mathbb{R}^+$ associating a real–valued *weight* to each arc. Such a graph is typically denoted by $G = (V, A, w)$. We denote by (u, v) a generic arc between two vertices $u, v \in V$, and by $w(u, v)$ the weight of an arc (u, v) of a weighted graph. A graph $G = (V, A)$ is *directed* if all arcs are *directed* from one vertex to another, that is connection relation does not satisfy the commutative property, namely $(u, v) \neq (v, u)$. From now onwards, we will deal with directed graphs

only, therefore, in what follows, we use the term graph (weighted graph, respectively) whenever we refer to a directed graph (directed weighted graph, respectively).

Given a graph $G = (V, A)$, we denote by $N^G_{out}(v) = \{u \in V \ : \ (v, u) \in A\}$ ($N^G_{in}(v) = \{u \in V \ : \ (u, v) \in A\}$, respectively) the set of *outgoing* (*incoming*, respectively) neighbors of a vertex $v \in V$. In addition to these, we denote by $\text{DEG}(v)$ the degree of a vertex $v \in V$ that is equal to the sum of the cardinality of $N^G_{out}(v)$ and $N^G_{in}(v)$. An arc (u, v) is *adjacent* to a vertex v in a graph if $v \in N^G_{out}(u)$. An arc (u, v) is *incident* to a vertex v in a directed graph if $v \in N^G_{in}(u)$. Two arcs are *consecutive* if they share a common endpoint, e.g. (u, v) and (v, w) are adjacent on vertex v.

A path P_{uv} between two vertices u and v of V is a sequence of k consecutive arcs that is: i) the first arc is adjacent to u; ii) the k–th arc is incident to v; iii) for each i such that $0 \leq i \leq k - 1$ the i–th arc and the $(i + 1)$–th arc are consecutive. k is called the length of such a path. A graph is *acyclic* if it does not contain a *cycle*, that is a path P_{uu} whose first and last vertex coincide.

We say a vertex u *is reachable from* (*reaches*, respectively) another vertex v if and only if there exists a path P_{vu} from v to u (P_{uv} from u to v, respectively) in G. The weight $d_G(P_{vu})$ of a path P_{vu} in a graph G is given by the sum of the weights of the arcs in the path. If the graph is unweighted, the weight of any path coincides with its length, by definition.

A path P_{vu} is said to be the *shortest path* between vertices v and u in G, and it is denoted by $P^*_{(v,u)}$, if $d_G(P^*_{(v,u)})$ is minimum among all paths between v and u in G. The weight of a shortest path is often referred to as the *distance* from u to v in G. Given a graph G and two vertices v and u in V, a shortest path $P^*_{(v,u)}$ in G can be computed using e.g. Dijkstra's algorithm, originally presented in [53]. The worst-case running time of the Dijkstra algorithm, if implemented using Fibonacci heaps [59], is $O(|A| + |V| \log(|V|))$. For achieving an improved running time, one can use a *bidirectional search* [39,60] when scanning vertices using the Dijkstra algorithm. A bidirectional search, for computing the shortest path $P^*_{(v,u)}$ in G, simultaneously scan vertices starting from u and v using a forward and a backward search, respectively. The search is stopped once the intersection of the set of vertices visited using the forward search and that using the backward search contains a common vertex lying on the shortest path $P^*_{(u,v)}$.

For determining whether a vertex u reaches another vertex v of a graph G, it suffices to run a breadth-first search algorithm (shortly, BFS) rooted at u [33]. The algorithm explores the graph until either v is encountered or all vertices are visited (in the latter case, u does not reach v). Note that the worst-case time of the BFS algorithm for a graph $G = (V, A)$ is $O(|A| + |V|)$ [33]. In practice, however, similar to Dijkstra algorithm, one

can use the bidirectional search when scanning vertices during the BFS. In particular, for computing a simple path P_{uv} in G, one can use a forward BFS starting from u and backward BFS starting from v, and stop the search as soon as a common vertex is reached using both the forward search and the backward search started at u and v, respectively [102].

Unfortunately, answering path queries, either of type shortest path or reachability, between any pair of vertices in a given graph G using the above-mentioned approaches is not feasible in practice in scenarios where thousands of queries need to be answered each hour [9], and that G is a large-sized graph containing millions of vertices and arcs. One may think of taking an advantage of the parallel computation (see, e.g., parallel versions of the Dijsktra algorithm [34,67,94] and references therein), however, such solutions are not scalable. To cope with this issue, a wide range of preprocessing techniques have been presented in the literature, explained in the section below.

2.2 Preprocessing Techniques

Preprocessing techniques aim at computing a preprocessed structure for a given input graph before answering queries of type either shortest path or reachability. This is usually carried out via a computationally intensive dedicated phase, after which, the preprocessed structure is used for answering queries. Such approaches include, but not limited to, those based on the notion of: i) *Goal-directed techniques* [65,80]; ii) *Computing all-pair shortest paths* [92]; iii) *Geometric containers* [21,90,95]; iv): *Arc-Flags* [46,66,69]; v) *Separator-based techniques* [43,93]; vi) *2-hop-cover labeling* [24,31,101], etc. In what follows, we discuss each of these in detail.

2.2.1 Goal-Directed Techniques

Goal-directed techniques as the name indicate compute shortest paths in a given graph $G = (V, A)$ by prioritizing vertices to be scanned during a search. In detail, by adding some intelligence to the Dijkstra algorithm, that is by deploying for example A^* algorithm [65, 80], which reduces the number of vertices scanned during a search using a heuristic estimate, e.g., Manhattan [78] if the graph represents geographical locations, etc. In more details, for any two vertices s and t in a generic graph G, A^* computes $P^*_{(s,t)}$ by associating heuristic estimate $\pi(u)$ with each vertex u in V, where $\pi(u)$ is a lower bound on $d_G(P_{ut})$ in G. It then runs a modified Dijkstra algorithm starting from s, and for each vertex u it scans, it adds $d_G(P_{su}) + \pi(u)$ in the queue. By doing so, the number of scanned vertices can be reduced as compared to that using the classical

Dijkstra algorithm. In particular, if $\pi(u)$ is exactly equal to $d_G(P^*_{(u,t)})$, then A^* scans only those vertices lying on the shortest path $P^*_{(s,t)}$ [9].

2.2.2 Computing All-Pair Shortest Paths

A simple yet computationally expensive approach is to compute all pair shortest paths [92] using e.g., iterative Dijkstra Algorithm, Bellman-Ford algorithm [58], etc. Alternatively, one can use the Floyd-Warshall algorithm [33], which runs in $O(|V|^3)$ time, and is reasonably faster than running the Dijkstra algorithm ($|V|$) times. The results of such execution are stored in a suitable structure, e.g., using compressed path databases [16, 18], to be retrieved later when answering queries. Note that, compressed path databases stores all pair shortest paths explicitly thus making the retrieval of shortest paths much faster (see, e.g., [17, 19]).

2.2.3 Geometric Containers

Given a graph $G = (V, A)$, a generic approach based on geometric containers associate a label set $L((u, v))$ with each arc $(u, v) \in A$ such that $L((u, v))$ contains all those vertices to which shortest path from u contains the arc (u, v) [21, 90, 95]. The cardinality of the set $L((u, v))$ is then minimized by making use of geometric coordinates (see, e.g., [90] and references therein).

It can be easily seen that, once the labels based on geometric-containers are computed for all of the arcs in A, the shortest path in G between any pair of vertices in V can be computed using a pruned-Dijkstra algorithm. In particular, for computing the shortest path $P^*_{(s,t)}$ in G, where s and t can be any two vertices in V, the pruned-Dijkstra algorithm scan only those arcs which are contained in $P^*_{(s,t)}$. Such arcs can be identified using the label sets computed in the preprocessing phase, which unfortunately requires the computation of all pair shortest paths in G.

2.2.4 Arc-Flags

The notion of Arc-Flags [46, 66, 69] is somewhat similar to that of geometric containers, and works as follows. Given a graph $G = (V, A)$, an Arc-Flags based algorithm proceeds by dividing G into k segments such that all segments roughly contain the uniform number of vertices. It then computes the shortest paths between all pairs of vertices in each segment, and associate k bits label with each arc (u, v) in A. In particular, the i^{th} bit of

the label associated with arc (u, v) is set to true only if (u, v) is contained in the shortest path ending at least one of the vertices belonging to i^{th} segment. Once such labels are computed, queries can be then answered using a pruned Dijkstra algorithm. Unfortunately, arc-flags based approaches, similar to those based on geometric containers, require a heavy preprocessing step (lasting even for several hours for moderately sized networks [9]). It is noteworthy to mention that this notion of graph partitioning can be also used for optimizing goal-directed techniques for computing the shortest paths in a given graph (see, e.g., [71] and reference therein).

2.2.5 Separator-based Techniques

Separator-based techniques mainly aim at computing components of a given input graph $G = (V, A)$ using either *vertex-separators* or *arc-separators*. In particular, vertex separators [30, 57] aim at computing a subset of vertices whose removal from a graph G decomposes G into various components (preferably with the uniform number of vertices across each such component). In the same line of approaches, those based on Arc separators also divide the vertices in G into $k \leq |V|$ partitions, once again preferably with the same number of vertices in each of the components [3], however, using the notion of cut arcs [33], which are arcs between boundary vertices of different components. Such partitions are then used for computing overlay graphs [23,66]. An overlay graph $G' = (V', A')$ for a given input graph $G = (V, A)$ is a graph with $V' \subseteq V$ and A' corresponding to shortest paths in G, thus forming a comparatively sparse graph. When computing the shortest paths, the overlay graph is used instead of the original graph. However, this once again involves scanning vertices of a graph, which may not be scalable.

2.2.6 2-Hop-Cover Labeling

2-Hop-Cover labeling, to this day, is the best available structure for answering queries (either shortest paths or reachability) since it offers comparatively low preprocessing time combined with low query time [2, 37]. Broadly, 2-Hop-Cover labeling comes in two flavors, namely *reachability labeling*, and *shortest path labeling*, where the former is used for answering reachability queries in a graph, while the later is used for answering shortest path queries in a graph.

In what follows, we formally describe both reachability and shortest path labeling.

2.2.6.1 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 [31]. 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 [31]. 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 [31]. However, numerous approaches have been presented to heuristically improve both the time to compute the labeling and its size [24, 101, 105]. Among them, the one in [105], called BUTTERFLY, has been shown to exhibit superior performance for DAGs, and is summarized as follows.

BUTTERFLY Algorithm Given a DAG $G = (V, A)$, the BUTTERFLY algorithm computes a 2HC-R labeling L of G is as follows. As a first step, the algorithm computes a *topological order* T of G. In details, T on 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 w.r.t. size of the graph (see, e.g. [33, 105] and references therein).

Once T is computed, the algorithm then transforms T into a so-called level ordering l useful in minimizing the labeling size. In detail, for each vertex $v \in V$, the algorithm assigns $l(v)$ a value in a range $[1, |V|]$ as the rank of v in l, also referred to as the level of v. The rank for each vertex v is defined in such a way that, no simple path from u to v in G contains a vertex w with $l(w) < l(u)$, i.e. by definition of T.

After the computation of l, the algorithm considers all vertices $v \in V$ in the ascending order of $l(v)$, and for each such vertex v it considers, the algorithm runs a forward BFS rooted at v. During the BFS, the algorithm performs a check to verify if $\text{L}_{out}(v) \cap \text{L}_{in}(u) = \emptyset$. In the affirmative cases, the algorithm adds v in $\text{L}_{in}(u)$ if $l(v) < l(u)$, otherwise u is added in $\text{L}_{out}(v)$. In the negative cases, instead, the BFS rooted at v is pruned at u due to the cover property of labeling (for details, see [101]).

For the sake of completion, the above steps are repeated for all of the vertices in V. Concerning the ordering criteria for sorting vertices in V, it is noteworthy to mention that, one may use, e.g. ordering with respect to degree, hierarchical ordering [1], approximate betweenness centrality [14,87], etc., in which cases, the BUTTERFLY procedure has been shown to be equally valid [105].

2.2.6.2 Shortest Path Labeling

Given a directed graph $G = (V, A)$, 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:

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

It can be shown that, for any 2HC-SP labeling, $\textsc{query}(u, v, \mathrm{L})$ always equals $d(u, v)$ [31], 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 the 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 $\textsc{L}_{out}(v)$ ($\textsc{L}_{in}(v)$, respectively) whenever $(h, \delta_{vh}) \in \textsc{L}_{out}(v)$ ($(h, \delta_{hv}) \in \textsc{L}_{in}(v)$, 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 [32] for more details.

Definition 2.1 (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 $\textsc{Path}(s, t, L)$.

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 [31]. However, numerous approaches have been presented to heuristically improve both the time to compute the labeling and its size [24, 101, 105].

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 $\textsc{L}$ 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 [32] 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 $\textsc{L}$.

3

The Journey Planning Problem

In this Chapter, we discuss the journey planning problem and its variants. We also discuss relevant approaches for solving the journey planning problem. In particular, Section 3.1 discusses the journey planning problem and its variants, while Section 3.2 discusses the approaches that can be used for solving the journey planning problem.

3.1 Journey Planning Problem

A journey planning problem takes as an 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 $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$, where $\mathcal{Z}$ is a set of *vehicles*, $\mathcal{S}$ 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 $c = (Z, \mathrm{s}_i, \mathrm{s}_j, t_d, t_a)$. Such a tuple is interpreted as vehicle $Z \in \mathcal{Z}$ leaves *departure stop* $\mathrm{s}_i \in \mathcal{S}$ at *departure time* t_d, and the immediately next stop of vehicle Z is stop $\mathrm{s}_j \in \mathcal{S}$ at time t_a (i.e. t_a is the *arrival time* of Z at *arrival stop* $\mathrm{s}_j \in \mathcal{S}$). 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 $|\mathcal{C}| \geq \max\{|\mathcal{S}|, |\mathcal{Z}|\}$, that is we do not consider vehicles and stops that do not take part in any connection. In the realistic scenario, each stop $\text{S}_i \in \mathcal{S}$ 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 3.1 (Trip)**.** A *trip* $\text{TRIP}_Z = (c_1, c_2, \ldots, c_k)$ is a sequence of k connections that: i) are operated by a vehicle $Z \in \mathcal{Z}$; ii) share pairwisely departure and arrival stop, i.e., formally, we have $c_{i'-1} = (Z, \text{S}_i, \text{S}_j, t_d, t_a)$ and $c_{i'} = (Z, \text{S}_j, \text{S}_k, t'_d, t'_a)$ 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, say TRIP_Z, 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_Z.

Definition 3.2 (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) allow to reach a given *target stop* starting from a distinguished *source stop* at a given departure time $\tau \geq 0$, i.e. 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 w.r.t. 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* $\text{EA}(\text{S}_i, \text{S}_j, \tau)$ asks, given a triple $\text{S}_i, \text{S}_j, \tau$ consisting of a *source stop* S_i, a *target stop* S_j, and a departure time $\tau \geq 0$, to compute a *quickest journey*, i.e. 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 3.3 (Time-Dominated Journey)**.** Let J' and J'' be two journeys, both connecting two same stops S_i and S_j. Then journey J'' is *time-dominated* by another 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''.

Definition 3.4 (Non-Time-Dominated Journey)**.** Let $\mathcal{J}$ be the set of all journeys connecting two stops S_i and S_j in a transit network. Then, a journey $J \in \mathcal{J}$ is *non-time-dominated* if and only if either of the two following conditions hold:

- the departure time of the first connection of J is larger than the departure time of the first connection of all other journeys in $\mathcal{J}$;
- the arrival time of the last connection of J is smaller than the arrival time of the last connection of all other journeys in $\mathcal{J}$.

Given the two above definitions, a *profile query* $\mathrm{PQ}(\mathrm{S}_i, \mathrm{S}_j, \tau, \tau')$ 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 $\tau < \tau'$, i.e. 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, a *multi-criteria query* $\text{MC-EA}(\mathrm{S}_i, \mathrm{S}_j, \tau)$ asks 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, e.g. monetary cost (see, e.g., [6, 103, 104] and references therein).

It is long known that the problem to determine all Pareto-optimal journeys is (weakly) NP-hard in general [100] since they can be exponential in number, while it is polynomially solvable, by a simple multi-criteria modification of the Dijkstra's algorithm, based on lexicographical optimality [100], whenever an ordering on the optimization criteria is imposed, e.g. one wants to compute the quickest journey between S_i and S_j, starting at a time greater than or equal to τ and (also) having the minimum number of trips (i.e. transfers). In this thesis, we focus on this latter realistic variant. Notice that, profile queries are a special case of multi-criteria ones using arrival and departure times as criteria. For the sake of completeness, we remark that there exist few scenarios where, despite the worst-case exponential number of Pareto optima, practical heuristics can be designed, see, e.g., [77].

3.2 Existing Approaches

A wide range of approaches for solving the journey planning problem has been presented in the last decade. Such approaches can be broadly classified into two categories: those representing the timetable as an *array* [47, 52], and those representing it as a *graph* [9]. Moreover, for optimizing query times, some of these approaches rely on a preprocessed structure computed via a computationally intensive preprocessing step [8, 49].

In what follows, we discuss the above-mentioned approaches in detail.

3.2.1 Array-based

Two of the most successful (and effective) examples of array based approaches are the Connection Scan Algorithm (CSA) [52] and the Round-bAsed Public Transit Optimized Router (RAPTOR) [47].

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 single array, which is scanned only once per query. CSA has been extended to answer profile queries [51] using a single pass over the connections, therefore eliminating the need for redundant scans. In addition to this, other several variants of CSA have been presented recently(see, e.g., [82, 93], etc.). In particular, the approach in [93], named *Accelerated Connection Scan Algorithm* (ACSA), is based on the idea of partitioning applied to stops. In detail, ACSA works by computing a partition of stops such that the number of connections with endpoints in different partitions is minimized. The partial journeys are computed inside each partition and are then used for answering the earliest arrival and profile queries. On the other hand, the one in [82] extends CSA for handling the unrestricted walking scenario [82] by integrating 2-Hop-Cover labeling of the footpath graph in the classical CSA. Unrestricted walking based public transit routing allows transfers from one stop to another stop using footpaths. In [97], it is experimentally shown that allowing unrestricted walking can significantly reduce the traveling time, even by hours for networks of Germany and Switzerland.

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 [47]. 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 heuristic improvements or considering more refined modeling strategies, have been presented and experimentally analyzed in the last couple of years (see,

e.g., [40, 98]). Furthermore, [42] adapts the idea of partitioning in RAPTOR, however, the improvements as compared to the original version of RAPTOR are not significant. A recently presented extension of RAPTOR, named *Ultra Transfers*, in [13] is based on the computation shortcut arcs [63, 89] between pairs of stops such that all queries can be answered correctly. In the same article, the idea of Ultra-Transfers is applied to the classical CSA, and is shown to be orders of magnitude faster as compared to that in [97]. Unfortunately, in the case of multi-criteria queries, RAPTOR with Ultra-Transfers yields less improvement as compared to the classical version of RAPTOR.

3.2.2 Graph-based

In this section, we discuss graph-based approaches for solving the journey planning problem. For the graph-based representation of a given timetable, a wide range of models has been presented in the last decade (see, e.g., [9, 28, 29, 45, 73, 74, 85] and references therein). Among them, the so-called reduced-time-expanded model in [29], which in-turn is based on the classical reduced-time-expanded model [73], is considered as a standard in the scientific community. In detail, in the context of public transit networks, a reduced-time expanded model can be used if one is interested in answering earliest arrival and profile queries. For answering multi-criteria queries, however, a variant of the reduced-time expanded model, namely, *weighted-reduced-time expanded model* [28], can be used. Below, we describe both of these models in detail, and discuss the relevant approaches for answering earliest arrival, profile and multi-criteria queries.

3.2.2.1 Reduced-Time Expanded Model

The input timetable $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$ associated to the transit network is modelled via a *reduced time-expanded graph* (RED-TE, for short) [29]. In the case of an aperiodic timetable, the RED-TE graphs is a directed acyclic graph (DAG) $G = (V, A)$ [41]. Starting from initially empty sets of vertices V and arcs A, the DAG G associated to the aperiodic timetable $\mathcal{T}$ is built as follows.

For each elementary connection $c = (Z, \mathrm{S}_i, \mathrm{S}_j, t_d, t_a)$:

- 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* DV$[i]$ (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 TRIP $= (c_0, c_1, \ldots, c_k)$, and for each connection $c_i \in$ 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 $u, v \in$ DV$[i]$, a *waiting arc* (u, v) is added to A if $time(v) \geq time(u)$, and there is no w in DV$[i]$ such that $time(v) \geq time(w) \geq time(u)$.
- for each $u \in$ AV$[i]$ and for each $v \in$ DV$[i]$, a *transfer arc* (u, v) is added to A if $time(v) \geq time(u) +$ MTT$_i$, and there is no $w \in$ DV$[i]$ such that $time(w) < time(v)$ and $time(w) \geq time(u) +$ MTT$_i$.

An example of the construction of RED-TE graph is given in Fig. 3.1, built using the timetable of Table 3.1 as input.

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	–

TABLE 3.1: An example of timetable with three stops X, Y, Z and five vehicles $\alpha, \beta, \gamma, \phi, \theta$.

Answering Earliest Arrival Queries Given a RED-TE graph $G = (V, A)$ for an input timetable $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$, an earliest arrival query EA(S_i, S_j, τ) can be answered using Dijkstra algorithm as follows.

First, a vertex $a \in$ DV$[i]$ satisfying the following conditions is computed:

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

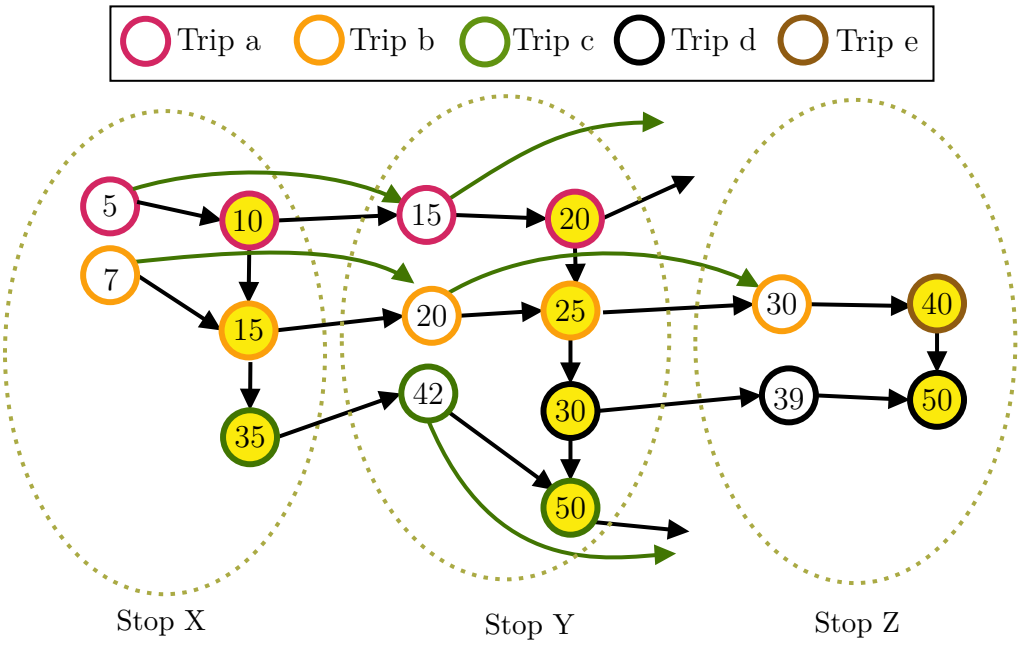

FIGURE 3.1: An example of RED-TE graph for a timetable with three stops X, Y and Z, five trips $\{a, b, c, d, e\}$. Details of the timetable are reported in Table 3.1. Each ellipse groups together vertices in $\text{DV}[i] \cup \text{AV}[i]$ for each stop $i \in \{X, Y, Z\}$, where vertices on the left side of each group, filled in white, 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, $\text{MTT}_X = \text{MTT}_Y = \text{MTT}_Z = 5$ minutes for all stops. Departure and arrival vertices of connections of the same trip are highlighted via the same border color. Bypass arcs are drawn in green while consecutive departure vertices of the same set $\text{DV}[i]$ are connected via waiting arcs.

Then, vertices in $\text{AV}[j]$ are scanned to search for vertices $v \in \text{AV}[j]$ such that a reaches v in G. 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 $v' \in \text{AV}[j]$ such that a reaches v' in G, and $time(v') < time(v)$ (this is easy to check as vertices are typically sorted by time).

For achieving comparatively improved query times, known adaptations of the Dijkstra algorithm [25, 28, 29, 33, 39, 60, 61, 83] can be also applied. Furthermore, as in the case of aperiodic timetables, the RED-TE model essentially yields a DAG [49], therefore the acyclic nature of the RED-TE graph can be used for computing a topological ordering of the graph (see, e.g., [26, 33] and references therein). By doing so, expensive prioritized queue operations can be eliminated and therefore result in improved query times [72]. In addition to this, one can also benefit from parallel computation, i.e. by doing tasks in parallel using multiple cores (see, e.g., [56, 70, 81] and references therein).

Despite adding the above-mentioned improvements, answering earliest arrival queries

via scanning vertices in the basic graph-based representation of a given timetable is far slower than that using CSA [44, 52]. One of the reasons is the increased number of vertices when representing a timetable using the graph. In particular, in the case of RED-TE graph-based representation of a given timetable, the number of vertices is equal to twice the number of connections.

Answering Profile Queries For answering a profile query $\text{PQ}(\text{S}_i, \text{S}_j, \tau, \tau')$, the easiest way is to iteratively run the Dijkstra algorithm for each vertex $v \in \text{DV}[i]$ such that $\tau \leq time(v) \leq \tau'$ (see, e.g., [79], etc.). In detail, the algorithm needs to compute a set of non-time-dominated journeys, let us call it PROFILE, by proceeding as follows.

First, the routine computes a vertex $a \in \text{DV}[i]$ such that:

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

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

1. v is reachable from a;
2. there is no other vertex $v' \in \text{AV}[j]$ having $time(v') < time(v)$ and is reachable from a in G.

Note that $time(v)$ is the earliest time to arrive at stop S_j given the departure time τ from S_i. Now, since set PROFILE must contain all non-time-dominated journeys, the latest departure time that allows reaching S_j is also necessary to complete the computation of the query. To this aim, the algorithm computes another vertex $a'' \in \text{DV}[i]$ such that there is no arc (a'', a''') having $time(a'') \leq time(a''')$ and that a''' reaches v in G. The computed time $time(a'')$ is the latest departure time that allows to reach S_j. Hence, the algorithm adds pair $(time(a''), time(v))$ to PROFILE (or alternately the corresponding journey, both versions of the set PROFILE are equivalent [41]), and repeats the process above by setting the value of τ to $time(a'') + 1$.

It can be easily seen that the above approach is costly due to multiple scans of some of the vertices, which however depends on the difference between τ' and τ. The *Self Pruning Connection Setting* (SPCS) algorithm in [45] eliminates multiple visits to the same vertices by using a *shared priority queue*. A shared priority queue Q has entries of pairs of the form $(h, time(v))$, where h is the vertex being scanned using a search started at some vertex v, and $time(v)$ is the time value associated with v. For the sake of performance, entries in Q are ordered w.r.t the time values.

The SPCS procedure runs a pruned Dijkstra algorithm simultaneously for all $v \in \text{DV}[i]$ with $\tau \geq time(v) \leq \tau'$ while maintaining the shared priority queue. In particular, whenever a vertex u is visited by a search started at some vertex $v' \in \text{DV}[i]$, a check is made to verify if u is already visited by another search started at some other vertex $v'' \in \text{DV}[i]$ with $time(v'') \geq time(v')$. In the affirmative cases, the search started at v' is pruned at u as an answer to a profile query contains non-dominated journeys only.

Analogous to the case with earliest arrival queries, CSA outperform SPCS in terms of query time as, for answering a profile query, CSA scans connections only once [44, 52].

3.2.2.2 Weighted Reduced Time-expanded Model

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 the minimum number of transfers between vehicles.

For the sake of understanding, we show in Fig. 3.2 an example weighted RED-TE graph for the timetable reported in Table 3.1.

Answering Multi-criteria Queries For answering to a multi-criteria query MC-EA$(\text{S}_i, \text{S}_j, \tau)$ using a graph-based representation of a schedule based public transportation network, one can run Dijkstra algorithm for each vertex $v \in \text{DV}[i]$ with $\tau \geq time(v) \leq \tau'$, and build Pareto-optimal set accordingly (see, e.g. [54] and references therein). In particular, given a WRED-TE graph $G = (V, A, w)$ of an input timetable $\mathcal{T}$, the routine for answering a multi-criteria query MC-EA$(\text{S}_i, \text{S}_j, \tau)$ is as follows:

1. first, a vertex $a \in \text{DV}[i]$ satisfying the following conditions is computed:
   (a) $time(a) \geq \tau$;
   (b) there is no arc (a', a) in G such that $a' \in \text{DV}[i]$ and $time(a') \geq \tau$, meaning that vertex a is associated with the smallest departure time larger than τ.

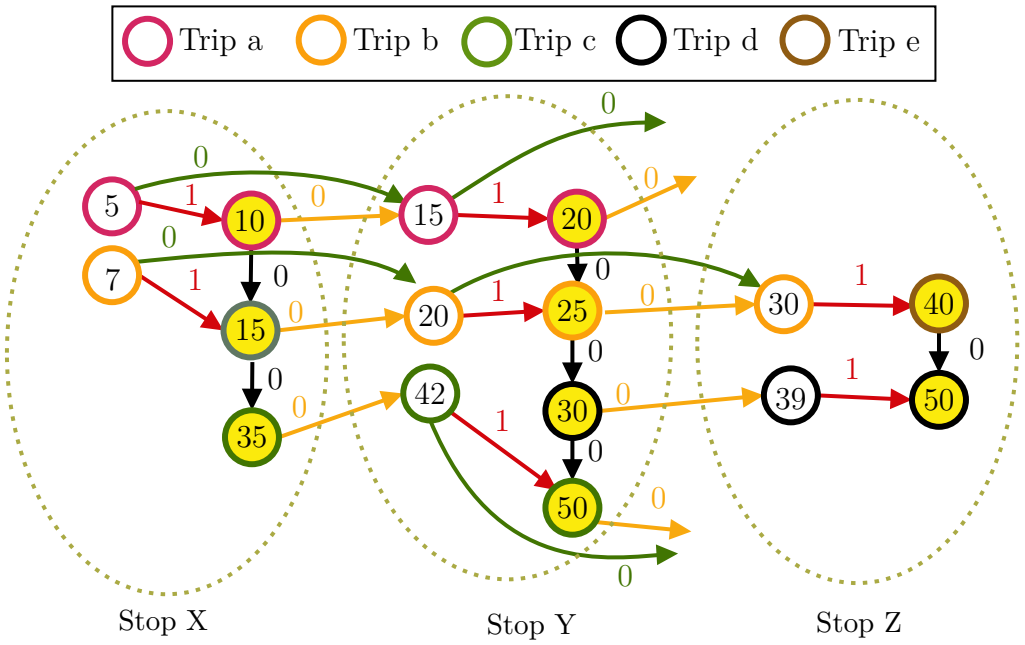

FIGURE 3.2: An example of weighted RED-TE graph for a timetable with three stops X, Y and Z, five trips $\{a, b, c, d, e\}$ reported in Table 3.1. Each ellipse groups together vertices in $\text{DV}[i] \cup \text{AV}[i]$ for each stop $i \in \{X, Y, Z\}$, where vertices on the left side of each group, filled in white, are arrival vertices, while vertices on the right side, filled in light yellow, are departure vertices. Transfer arcs are colored in red, with a weight value of 1 assigned to each of them. Waiting arcs (connection arcs, respectively) are colored in black (orange, respectively) with a weight value of 0 associated with each such arc. Lastly, bypass arcs are colored in green and are assigned a weight of value 0 to each of them.

2. then, the algorithm computes $d_G(P^*_{(a,v)})$ (using e.g., Dijkstra algorithm) for all $v \in \text{AV}[j]$ such that $time(a) < time(v)$, and selects the arrival vertex v having minimum $d_G(P^*_{(a,v)})$.

Finally, the time associated with such vertex is the earliest arrival time while $d_G(P^*_{(a,v)})$ is the associated number of transfers. Note that, optimizations such as those based on shared priority queues can be added to the Dijkstra algorithm by adapting it suitably according to the optimization criteria being considered.

Multi-criteria Label-Setting (MLS) algorithm [75] exploits the label setting the property of the Dijkstra algorithm to compute multi-criteria shortest paths while avoiding iterative execution of the Dijkstra algorithm. On the contrary to this, a rather naturalization of the Dijkstra algorithm, named *Layered Dijkstra* (LD) [22, 84], is shown to be more efficient than the iterative counterpart [44]. The LD algorithm proceeds by copying a time-table graph into k layers, where k is an upper bound on the number of transfers, and running a single criterion Dijkstra algorithm on each layer, and combine results in incremental fashion w.r.t the number of transfers. For more details about LD algorithm,

we refer the interested reader to [22, 84] and references therein. It is worthy to mention that, RAPTOR is experimentally shown to outperform both LD and MLS algorithms in terms of query time [9].

3.2.2.3 Speedup Techniques

A wide range of generic speedup techniques, that is those based on a preprocessing phase before actually answering queries, presented originally in other contexts have been adapted to optimize query processing in the domain of public transit networks. In addition to this, speedup approaches developed specifically in the context of public transit networks are also presented recently.

In what follows, we draw a detailed comparative overview of the speedup techniques, both the adapted ones and those developed originally in the context of public transit networks.

We start by discussing at first the speedup techniques applied in conjunction with the Dijkstra algorithm. In particular, one can use a goal-directed technique such as A^*, etc. Alternatively, one can compute all pair shortest paths and store the results in a suitable structure to be retrieved later when answering queries. However, this may not be feasible for large-sized public transit networks.

For solving the space problem when storing all pair shortest paths, one can use techniques, such as those based on the notion of geometric containers [21, 90, 95], arc flags [46,66,69], and separators (arc/vertex-based) [43,93], etc. In the same line of graph partitioning based approaches, it is worthy to mention that, the notion of the overlay graph is widely investigated in the context of road networks (see, e.g., [64, 68, 95], and references therein). Specific to the public transit network, [90] considered a given set of important stops for computing an overlay graph such that the distances in the original graph were preserved. Such an overlay graph, as compared to the original graph, has been shown to yield improved query times [91]. Notice that, in the context of geometric containers in scenarios where geometric coordinates are not available, overlay graphs are also used for approximating label sets associated with each arc [21].

Furthermore, among those approaches originally presented in the context of public transit networks, a so-called *transit–node–routing* (TNR) [5, 7, 11, 12, 88] paradigm is also based on the idea of partitioning. In detail, given a graph $G = (V, A)$, a TNR based approach proceed by selecting a subset $V' \subseteq V$ of vertices known as *transit vertices*. Transit vertices are selected based on some criteria, e.g. using maximum degree, centrality measures (see, e.g. [20] and references therein), etc. Given V', the algorithm

computes all pair shortest paths for all of the vertices in V'. After this, the algorithm computes access vertices $AV \subseteq V'$ for each vertex $v \in V \setminus V'$. In more detail, a vertex $v' \in V'$ is added in AV, associated with v, only if v' is the first transit vertex lying on the shortest path between v and any other vertex u in G. Once access vertices are computed, they are then used for computing the shortest paths in G between any two vertices s and t in V. For more details about such approaches, we refer the reader to [11, 88] and references therein. In [9], it has been shown that approaches based on TNR do not outperform Transfer Patter (TP) [8], which is specifically developed for public transit networks.

In details, TP is based on the observation that the number of transfers in optimal journeys are rare [8], thus if the sequence of transfers is precomputed, then this information can be used to answer multi-criteria queries without running Dijkstra algorithm on the original graph-based representation of the public transit network. Clearly, for public transit networks of large size, computing all transfer patterns may not be practical. To cope with this issue, the authors in [8] select a subset of important stops (given as input), namely hub stops, and compute transfer patterns from hub stops to all stops and all non-hub stops to hub stops. For reducing further the preprocessing effort, one can impose the constraint on the maximum number of transfers, e.g. 2 or 3, etc., when computing transfer patterns, however leading to a heuristics-based solution and not the optimal one. Nonetheless, TP (thanks to preprocessing) outperform ACSA and RAPTOR, however, it has been shown that the preprocessing phase of TP takes even days for moderately sized networks(see, e.g., [9] and references therein).

Concerning the query time, the state-of-the-art method to solve the journey planning problem is commonly referred to as *Public Transit Labeling* (PTL) [41]. In detail, PTL 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, e.g. to seek for earliest arrival journeys that also minimize the number of transfers. Below, we describe each of the two versions in detail.

Basic Public Transit Labeling The main idea underlying the basic 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 graph G, computes a 2HC-R labeling L of G, and compresses it into a set of *stop labels* SL of L [41]. Note that, there is a one-to-one correspondence between RED-TE graphs and classic time–expanded graphs [29]. Concerning the stop labels, we have a *forward stop label* $\text{SL}_{out}(i)$ and a *backward stop label* $\text{SL}_{in}(i)$ for each stop $\text{S}_i \in \mathcal{S}$. 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 [41].

In particular, the routine for answering to an earliest arrival query $\text{EA}(\text{S}_i, \text{S}_j, \tau)$ 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 $\text{EA}(\text{S}_i, \text{S}_j, \tau)$.

The query algorithm for answering a profile query $\text{PQ}(\text{S}_i, \text{S}_j, \tau, \tau')$ 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-time-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 $\text{PQ}(\text{S}_i, \text{S}_j, \tau, \tau')$.

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 [41] 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, a *shortest path labeling* replaces the reachability labeling, and the query algorithm is modified accordingly. Again note that there is a one-to-one correspondence between RED-TE graphs and classic time–expanded graphs [29]. 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) [41].

It is shown that using a WRED-TE graph $G = (V, A, w)$ and a 2HC-SP labeling of G, one can answer efficiently to both earliest arrival and profile queries on a timetable (see [41] for more details). Moreover, multi-criteria queries can answered using the approach discussed above in Section 3.2.2.2, however, rather than using the Dijkstra algorithm for computing shortest paths in G, 2HC-SP labeling L of G is used to serve the purpose. It is worthy to mention that, the structure of the journey again can be retrieved by applying a recursive query procedure [37,41]. Note also that, to achieve the fastest possible query times, PTL employs some pruning mechanisms [41]. 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 for multi-criteria queries.

4

Dynamic Journey Planning Problem

In this Chapter, we describe the dynamic journey planning problem, and discuss relevant approaches for solving the said problem. In particular, Section 4.1 describes the dynamic journey planning problem, while Section 4.2 discusses relevant approaches that can be used for solving the it.

4.1 Dynamic Journey Planning Problem

A dynamic journey planning problem is the problem of computing journeys in a timetable $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$ that is subjected to unpredictable changes. Changes in $\mathcal{T}$ can be for example due to the addition of a new elementary connection in $\mathcal{C}$ etc. The most frequently occurring changes in $\mathcal{T}$, however, is due to delays in the departure times of one or more elementary connections in $\mathcal{C}$ [9, 76].

Formally, a *delay* is an increase in the departure time of an elementary connection of a finite quantity $\delta > 0$. In detail, given an elementary connection $c_i = (Z, \mathrm{S}_i, \mathrm{S}_j, t_d, t_a)$ served in a trip $\textsc{trip}_Z$, a delay δ in c_i is the number of minutes for which the departure of the vehicle Z from the stop S_i is delayed. Note that, if c_i is followed by other connections in the same trip $\textsc{trip}_Z$, then the delay δ in c_i may also trigger delays in the connections following c_i in $\textsc{trip}_Z$.

There are several mechanisms defined for propagating delays in a public transit network (see, e.g., [4,55], etc.). Among them, the most common approach (which we also considered in our work) is to propagate the delay only if the sum of the delayed arrival time of a generic vehicle Z at some stop S_j and MTT_j is not exceeding the departure time of the following connection in TRIP_Z. More precisely, given two elementary connections $c_i = (Z, \text{S}_i, \text{S}_j, t_d, t_a)$ and $c_{i+1} = (Z, \text{S}_j, \text{S}_{j'}, t_{d'}, t_{a'})$, if a delay of δ time-units is introduced in c_i, then the delay affects c_{i+1} only if $(t_a + \delta) + \text{MTT}_j < t_{d'}$. By generlizing this procedure, given an input delay δ introduced in a generic connection c_i served in a trip TRIP_Z, one can easily compute the list of all elementary connections in TRIP_Z, affected by the delay δ in c_i.

4.2 Existing Approaches

For handling delays in a given timetable $\mathcal{T}$ modeled using a graph-based representation such as RED-TE graph etc., the simplest way is to accordingly update the graph, after each update in $\mathcal{T}$, using the procedures of [28]. Concerning the updates to G, they can be *incremental* (*decremental*, respectively), if they are additions (removals, respectively) of a vertex/arc. Note that the removal of a vertex v from G also results in the removal of all arcs (u, v) and (v, u), for any $u \in V$, from A.

In addition to updating RED-TE graph, one can also use the dynamic graph-based model of [28], which is tailored for handling delays, thus making handling delays much simpler task. However, approaches of such kind rely on the execution of the Dijkstra algorithm (or a variant of the Dijkstra algorithm) for answering queries.

Concerning the array-based models, [51] extends CSA to operate under randomly occurring delays and is based on the idea that, minimizing the expected arrival time helps to minimize the number of transfers. This is true because each "unnecessary" transfer introduces additional uncertainty. The expected arrival time in [51] is computed with respect to a simple model for the probability that a transfer breaks. The approach, as an output, in addition to the journey, also provides instructions to the user at each point of their journey, including cases in which a connecting trip is missed. On the contrary to this, one can update already computed journeys using the notion of robust recoverable journeys [62]. Robust recoverable journeys essentially allows for updating the journeys on the fly if a delay is detected at any point in the journey.

In addition to the above mentioned approaches, a realistic stochastic model predicting the propagation of delays is presented in [15]. It is worthy to mention that this model has been evaluated on the real data for Deutsche Bahn (for details, see [15]). Moreover,

TP has been studied in the context of delays, and it has been shown that using TP in the presence of delays, one can correctly answer queries for 99% of the cases (see, [10]).

In the presence of a preprocessed structure computed on top of the graph-based representation, one can adapt the query answering mechanisms, originally presented in the context of road-networks (see, e.g., [48]), to deal with factors such as traffic congestion, etc. However, road-networks are not time-dependent, and in our case, the dependency on the time makes it inevitably hard to adapt the query answering mechanism used in the case of road-networks.

For scenarios where the preprocessed structure for a given timetable $\mathcal{T}$ is computed using geometric containers, updates in $\mathcal{T}$ can be reflected onto the preprocessed structure using the procedures in [96]. The same can be done for arc-flags based preprocessed structure however using the update procedures of [38]. In scenarios where 2-Hop-Cover labeling is used as the preprocessed structure, one can update the underlying labeling (either 2HC-R or 2HC-SP) using the approaches in [2, 37, 105].

In what follows, we describe approaches for updating 2-Hop-Cover labeling of a graph in detail.

4.2.1 Updating 2-Hop-Cover Labeling

Given a 2-Hop-Cover labeling L (either 2HC-SP or 2HC-R) of a graph $G = (V, A)$, let G' be the graph obtained by applying an incremental or decremental update to G, then one can compute L$'$ of G' by updating L [2, 37, 105]. In particular, after an incremental update, say the addition of an arc (u, v) in G and let the resultant graph be G', then using the incremental algorithm INCPLL of [2], one can dynamically compute L$'$ of G'. In detail, INCPLL compute L$'$ by running pruned BFS starting from each hub vertex in the label sets associated with u and v, and add accordingly new label entries to L. Notice that, after an incremental update, it is not possible to have an increase in the length of a shortest path for any pair of vertices in G, therefore no need to remove existing label entries from L. On the positive side, INCPLL has been shown to update L in a fraction of the total time required for computing L from the scratch. In addition to this, INCPLL is a generic algorithm and works for all kinds of graphs. On the negative side, updating L using INCPLL, in the long run, may results in an increased labeling size.

In contrast to INCPLL, dynamically updating L after decremental updates in G is shown to be comparatively complex in terms of running time [37] as any such procedure, before adding new label entries in L, must remove at first invalid hubs. A generic hub h in

$(h, \delta_{uh}) \in \text{L}_{out}(u)$ ($(h, \delta_{hv}) \in \text{L}_{in}(v)$, respectively), after the removal of an arc (s, t) (vertex s, respectively) from G, is invalid if the induced path $\text{PATH}(u, h, L)$ ($\text{PATH}(h, v, L)$, respectively) passes through the removed arc (vertex, respectively). Moreover, the label entry (h, δ_{uh}) ((h, δ_{hv}), respectively) is said to be an invalid label entry.

The decremental algorithm DECPLL of [37], after a decremental update (say caused by the removal of an arc) in G, compute L' by identifying and removing all invalid hub entries, followed by the addition of new label entries in L. More formally, given a 2-Hop-Cover labeling L of an input graph $G = (V, A)$, and an arc (u, v) removed from G, the DECPLL algorithm computes two sets of vertices AFF_u and AFF_v, that is the set of vertices whose associated label sets contain invalid label entries. In detail, for computing AFF_u and AFF_v, the algorithm, as a first step, adds u and v in AFF_u and AFF_v, respectively. It then run a backward BFS rooted at u, and on scanning a vertex $u' \in V$, if the the path $\text{PATH}(u', v, L)$ passes through any of the vertices in AFF_u, then the algorithm adds u' in AFF_u. It then runs a forward BFS rooted at v, and for each vertex $v' \in V$ scanned during the BFS, if the path $\text{PATH}(u, v', L)$ passes through any of the vertices in AFF_v, then the algorithm adds v' in AFF_v. Once AFF_v and AFF_u are computed, then DECPLL remove all invalid label entries from L. Finally, it runs a forward BFS (backward BFS, respectively) for each vertex $h \in AFF_v$ ($h \in AFF_u$, respectively), and add new label entries in L. For more details on the working of the DECPLL procedure, we refer the interested reader to [37].

Both algorithms INCPLL and DECPLL works for any kind of graphs, however, the authors in [105] provide dedicated algorithms for dynamically updating 2HC-R labeling for DAGs. In what follows, we describe the working of dynamic algorithms of [105], that is BUTTERFLY decremental and incremental algorithms.

BUTTERFLY Decremental Algorithm Given a DAG graph $G = (V, A)$ and a 2HC-R L of G, if a generic vertex $x \in V$ is removed from G, then let G' be the new graph and let l' be the updated level ordering of G', the decremental algorithm of [105] for dynamically updating L is as follows.

The algorithm begins by removing all the references to x in L. It then computes two sets of vertices, namely $B^+(x)$ and $B^-(x)$. In particular, $B^+(x)$ ($B^-(x)$, respectively) contains all those vertices that can be reached using a forward BFS (using a backward BFS, respectively) rooted at x in G. Once $B^+(x)$ and $B^-(x)$ are computed, then the algorithm considers each vertex $v \in B^+(x)$ in the ascending order of $t(v)$, and compute a set of vertices, namely $C_{in}(v)$. In particular, $C_{in}(v)$ is computed as the union of $\text{L}_{in}(u)$ for all $u \in N_{in}(v)$ in G'. Once $C_{in}(v)$ is computed, the algorithm then checks if

$\text{L}_{out}(h) \cap \text{L}_{in}(v) = \emptyset$ for all $h \in C_{in}(v)$. In the affirmative cases, it adds h to $\text{L}_{in}(v)$ if $l'(v) > l'(h)$. If instead $l'(v) < l'(h)$, then v is added to $\text{L}_{out}(h)$.

Once all of the vertices in $B^+(x)$ are processed, the algorithm then considers each vertex $v \in B^-(x)$ in the descending order of $t(v)$, and compute a set of vertices, namely $C_{out}(v)$, as the union of $\text{L}_{out}(u)$ for all such $u \in N_{out}(v)$ in G'. After which, for each vertex $h \in C_{out}(v)$, it checks if $\text{L}_{in}(h) \cap \text{L}_{out}(v) = \emptyset$. In the affirmative cases, the algorithm adds h to $\text{L}_{out}(v)$ if $l'(v) > l'(h)$, otherwise v is added to $\text{L}_{in}(h)$. It is noteworthy to mention that the algorithm also handle decremental updates caused by the removal of arcs etc.

The asymptotic worst-case runtime complexity for the BUTTERFLY decremental algorithm is shown to be $O(V^2)$ [105]. However, when applied in practice, the decremental algorithm is shown to update L within microseconds for very large-sized graphs. For more details about the BUTTERFLY decremental algorithm, we refer the reader to [105].

For the sake of simplicity, throughout this document, we denote by DEC-BU(G, L, v) the result of the application of the dynamic algorithm to the labeling L of a graph G to handle decremental operation occurring on vertex v.

BUTTERFLY Incremental Algorithm The incremental algorithm of [105], given a DAG graph $G = (V, A)$ and a 2HC-R labels L of G, dynamically update L after an incremental update is introduced in G. In particular, after an incremental update, say caused by the insertion of a new vertex x along with the arcs adjacent (incident, respectively) to (on, respectively) x in G and let the new graph be G', the incremental algorithm dynamically update L in a two-phased approach, described as follows.

In the first phase, the incremental algorithm computes $\text{L}_{in}(x)$ ($\text{L}_{out}(x)$, respectively) using the in-label sets (out-label sets, respectively) of vertices in $N^{G'}_{in}(x)$ ($N^{G'}_{out}(x)$, respectively). In details, for computing $\text{L}_{in}(x)$, the algorithm iterate through all the label entries h in $\text{L}_{in}(u)$ for all $u \in N^{G'}_{in}(x)$, and for each such h it scans, it checks if $\text{L}_{out}(h) \cap \text{L}_{in}(x) = \emptyset$. In the affirmative cases, it adds h to $\text{L}_{in}(x)$ if $l'(h) < l'(x)$, otherwise it adds x to $\text{L}_{out}(h)$. Note that, l' is essentially the level ordering of G'.

For computing $\text{L}_{out}(x)$, instead, the algorithm iterate through all the label entries h in $\text{L}_{out}(u)$ for all $u \in N^{G'}_{out}(x)$, and for each such h it scans, it checks if $\text{L}_{out}(x) \cap \text{L}_{in}(h) = \emptyset$. In the affirmative cases, it adds h to $\text{L}_{out}(x)$ if $l'(h) < l'(x)$, otherwise it adds x to $\text{L}_{in}(h)$.

In the second phase instead, the incremental algorithm update L by considering the labels in $\text{L}_{in}(x)$ and $\text{L}_{out}(x)$ computed in the first phase. In details, the algorithm considers each $u \in \text{L}_{in}(x)$ in ascending order of $l'(u)$, and for each label $w \in \text{L}_{out}(x)$ with $l'(w) > l'(x)$,

it performs two operations: i) It adds u to $\text{L}_{in}(w)$ if $\text{L}_{out}(u) \cap \text{L}_{in}(w) = \emptyset$; and ii) for each vertex $v \in V$ with $w \in \text{L}_{in}(v)$, it adds u in $\text{L}_{in}(v)$ if $\text{L}_{out}(u) \cap \text{L}_{in}(v) = \emptyset$. Once the operations i) and ii) are performed for all of the vertices in $\text{L}_{in}(x)$, the algorithm then consider the vertices in $\text{L}_{out}(x)$. In details, the algorithm considers each vertex $u \in \text{L}_{out}(x)$ in the descending order of $l'(u)$, and for each vertex $w \in \text{L}_{in}(x)$ with $l'(w) > l'(u)$, and performs two operations: i) It adds u to $\text{L}_{out}(w)$ if $\text{L}_{in}(u) \cap \text{L}_{out}(w) = \emptyset$; and ii) for each vertex $v \in V$ with $w \in \text{L}_{out}(v)$, it adds u in $\text{L}_{out}(v)$ if $\text{L}_{out}(v) \cap \text{L}_{in}(u) = \emptyset$.

Analogous to the decremental algorithm of [105] (discussed above), the incremental algorithm has a worst-case run time complexity of $O(V^2)$. However, the incremental update procedure is experimentally shown to update L within seconds for graphs contains millions of vertices [105].

For the sake of simplicity, throughout this document, we denote by $\text{INC-BU}(G, \text{L}, v)$ the result of the application of the dynamic algorithm of [105] to the labeling L of a graph G to handle an incremental operation occurring on vertex v (we refer the reader to [105] for more details on the above dynamic algorithm).

For the sake of ease in reading, in Table 4.1, we provide a list of notations we introduced so far, that are important for understanding the rest of the document.

Remarks

As mentioned in the earlier Chapter, PTL is the state of the art approach, in terms of query time, for solving the journey planning problem. However, the inherently dynamic nature of public transit networks essentially makes the use of PTL nearly impossible due to the preprocessing overhead. One may utilize the dynamic algorithms for updating the 2-Hop-Cover labeling, however, we observed in our earlier experiments that, applying labeling update approaches, for updating the preprocessed structure (excluding the stop labels) employed by the PTL, incurs very high update times, which sometimes exceeds the time required for computing a 2-Hop-Cover labeling from scratch. Following our discussion, it is easy to observe that, there exist no efficient algorithm (in terms of running time) tailored for dynamically updating the preprocessed structure employed by PTL, both the basic and the multi-criteria versions. Moreover, to the best of our knowledge, we are not aware of any approach for dynamically updating the stop labels.

Notations	**Definitions**
$G = (V, A)$	A directed graph with a set V of vertices and a set A of arcs
$G = (V, A, w)$	A weighted-directed graph with a set V of vertices, a set A of arcs, and a weight function w
(u, v)	An arc between u and v
$w(u, v)$	The weight of an arc (u, v)
$N^G_{out}(v)$	A set of out-neighbors of a vertex v in a directed graph
$N^G_{in}(v)$	A set of in-neighbors of a vertex v in a directed graph
$\text{DEG}(v)$	The degree of a vertex v
P_{uv}	A path between two vertices u and v in G
$d_G(P_{uv})$	The length of a path P_{uv} in G
$P^*_{(u,v)}$	The shortest path between two vertices u and v in G
$\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$	A timetable of a public transportation network
$\mathcal{Z}$	A set of vehicles in $\mathcal{T}$
$\mathcal{S}$	A set of stops in $\mathcal{T}$
$\mathcal{C}$	A set of elementary connections in $\mathcal{T}$
$c_i = (Z, \text{S}_s, \text{S}_t, t_d, t_a)$	An elementary connection $c_i \in \mathcal{C}$
t_a	An arrival time
t_d	A departure time
TRIP_i	A trip served by a vehicle i
$\text{EA}(\text{S}_i, \text{S}_j, \tau)$	Earliest arrival query with departure stop S_i, arrival stop S_j and departure time at S_i no later than time τ
$\text{PQ}(\text{S}_i, \text{S}_j, \tau, \tau')$	Profile arrival query with departure stop S_i, arrival stop S_j and departure time at S_i in a range $[\tau, \tau']$
$\text{MC-EA}(\text{S}_i, \text{S}_j, \tau)$	Multi-criteria earliest arrival query with departure stop S_i, arrival stop S_j and a departure time no later than τ
RED-TE	Reduced–time–expanded
WRED-TE	Weighted–reduced–time–expanded
v^c_d	Departure vertex associated with the departure event of a connection c
v^c_a	Arrival vertex associated with the departure event of a connection c
$time(v)$	The time associated with a vertex v
$\text{DV}[i]$	The set of vertices with an associated departure stop as S_i
$\text{AV}[i]$	The set of vertices with an associated arrival stop as S_i
L	2-Hop Cover labeling of G
$\text{L}_{in}(v)$	In-label set associated to a vertex v
$\text{L}_{out}(v)$	Out-label set associated to a vertex v
SL	Stop labels
$\text{SL}_{in}(i)$	The set of in-stop labels associated to a stop S_i
$\text{SL}_{out}(i)$	The set of out-stop labels associated to a stop S_i

TABLE 4.1: List of frequently used notations important for understanding the rest of the thesis.

5

Dynamic Public Transit Labeling

In this Chapter, we provide a detailed description of the approach we proposed for maintaining the PTL structure, both basic and multi-criteria, for a given public transit network with randomly occurring delays. It is worthy to mention that the proposed approach is presented in [49, 50].

This Chapter is organized as follows: Section 5.1 explains in detail the approach we proposed for dynamically updating 2HC-R labeling, both standard, and compact form. Section 5.2 then extends the proposed approach for dynamically updating the multi-criteria PTL structure as well.

5.1 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.

It is easy to see how a delay can induce an arbitrary number of changes to both the graph and labelings [29, 41], 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 [28, 29, 83]) and then reflect all these changes on both L and SL by: i) detecting and removing obsolete label entries; and ii) adding newly updated label entries induced by the new graph, as done in other works on the subject [37, 105]. However, as mentioned in the earlier chapter, this results in a quite high computational effort, as shown by preliminary experimentation we conducted.

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 [105]. 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, resp., *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 6).

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, resp.) *violates* the RED-TE constraints whenever the associated time (the difference of the times of the endpoints, resp.) does not satisfy at least one of the inequalities imposed by the RED-TE model discussed in Section 3.2.2. 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 [105] 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 [105]. 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 6). In the next sections we describe in details the above sub-routines.

5.1.1 Removal phase

In this section we describe in details Algorithm REM-D-PTL, whose aim is removing from G vertices and arcs related with the delayed connection, and to update accordingly L (see Algorithm 1). In particular, assume we have a delay of $\delta > 0$ occurring on a connection c_m, of a trip, say $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$. This induces an increase in the departure and arrival times of c_m, and possibly of all connections c_j such that $m < j \leq k$, which, in turn, impacts on the structure of the graph and of the 2HC-R labeling. Hence, our update is performed once per connection c_j for $j = m, m+1, \ldots, k$. In details, for each connection c_j in the above order, the procedure starts by increasing of δ the time associated to the two vertices $v_d^{c_j}$ and $v_a^{c_j}$. Then, if we let S_s (S_t, resp.) be the departure (arrival, resp.) stop of c_j, concerning $v_d^{c_j}$, we must verify whether the increase in the times associated to the two vertices $v_d^{c_j}$ and $v_a^{c_j}$ induces a violation of constraints imposed on the arcs by the RED-TE graph.

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, i.e. 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, i.e. 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 Fig. 5.1.

As a final remark on this part, notice that (see Fig. 5.1) 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. From now onwards, for the sake of brevity, we denote this test by writing either "the graph has changed" or not.

5.1.2 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

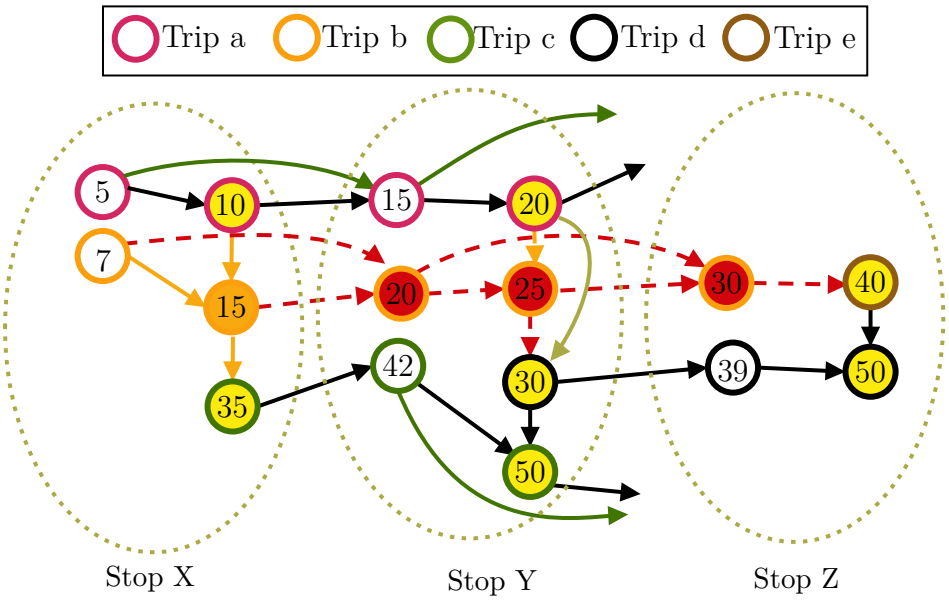

FIGURE 5.1: The RED-TE graph obtained after performing Algorithm REM-D-PTL (Algorithm 1) on the graph of Figure 3.1, as a consequence of a delay δ of 10 minutes occurring on the first connection of Trip β. The time associated to the departure vertex (filled in orange) of said connection is updated, but 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 light-brown, is added during the removal phase to connect departure vertices that remain in DV$[Y]$ to restore the RED-TE properties.

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 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.

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 3 which is called as sub-routine by Algorithm 2. In particular, if $v_d^{c_j}$ is the last vertex in DV$[s]$ (see Line 2 of Algorithm 3 – 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 AV$[s]$ that do not have any adjacent transfer arc and would not violate the RED-TE constraints, i.e. we add a vertex $v \in$ 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

Algorithm 1: Algorithm REM-D-PTL.

Input: RED-TE graph G, a delay $\delta > 0$ affecting a connection c_m, the trip 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

1 **for** $j = m, m+1, \ldots, k-1, k$ **do**
2 Let S_s and S_t be departure and arrival, resp., stops of c_j;
3 PRED $\leftarrow \infty$;
4 SUCC $\leftarrow \infty$;
5 $time(v_d^{c_j}) \leftarrow time(v_d^{c_j}) + \delta$;
6 $time(v_a^{c_j}) \leftarrow time(v_a^{c_j}) + \delta$;
7 **foreach** $v \in N_{out}^G(v_d^{c_j})$ **do** // Outgoing arcs (if any)
8 **if** $v \in$ DV$[s]$ **then** SUCC $\leftarrow v$; // Waiting arc in the graph
9 **if** $time(v_d^{c_j}) > time($SUCC$)$ **then**
10 **foreach** $v \in N_{in}^G(v_d^{c_j})$ **do** // Incoming arcs (if any)
11 **if** $v \in$ DV$[s]$ **then** PRED $\leftarrow v$; // Waiting arc in the graph
12 **if** $v \in$ AV$[s]$ **then** $\overline{A} \leftarrow \overline{A} \cup \{v\}$; // Transfer arc in stop S_s
13 $V \leftarrow V \setminus \{v_d^{c_j}\}$;
14 L $\leftarrow$ DEC-BU$(G, \text{L}, v_d^{c_j})$;
15 **if** PRED $\neq \infty \wedge$ SUCC $\neq \infty$ **then** $A \leftarrow A \cup \{($PRED, SUCC$)\}$; // Add waiting arc;
16 **foreach** $w \in \overline{A}$ **do** $A \leftarrow A \cup \{(w,$ SUCC$)\}$; // Add transfer arcs;
17 L $\leftarrow$ INC-BU$(G, \text{L},$ SUCC$)$;
18 **foreach** $v \in N_{out}^{out}(()v_a^{c_j})$ **do** // Outgoing arcs (if any)
19 **if** $v \in$ DV$[t] \wedge time(v) < time(v_a^{c_j}) +$ MTT$_t$ **then**
20 $V \leftarrow V \setminus \{v_a^{c_j}\}$;
21 L $\leftarrow$ DEC-BU$(G, \text{L}, v_a^{c_j})$;
22 **break**
23 **if** G *has not changed* **then break**;

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 DV$[s]$ (see Line 14 of Algorithm 3 – Sub-case I.b), i.e. there exists some waiting arc connecting $v_d^{c_j}$ to a vertex SUCC $\in$ 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 $time(v_d^{c_j}) \geq time(v) +$ 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 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).

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 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 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, resp.;
  PRED $\leftarrow \infty$;
  SUCC $\leftarrow \infty$;
  foreach $v \in N^G_{out}(v^{c_j}_d)$ **do** // Outgoing arcs (if any)
    if $v \in$ DV$[s]$ **then** SUCC $\leftarrow v$; // Waiting arc in the graph
  foreach $v \in N^G_{in}(v^{c_j}_d)$ **do** // Incoming arcs (if any)
    if $v \in$ DV$[s]$ **then** PRED $\leftarrow v$; // Waiting arc in the graph
  if $v^{c_j}_d \in V \wedge v^{c_j}_a \in V$ **then** // Case I) - Both not removed
    Execute RewireTransferDep$(G, v^{c_j}_d, \text{SUCC}, \mathrm{S}_s)$; // i.e. Algorithm 4
    if *G has changed* **then**
      L $\leftarrow$ INC-BU$(G, \mathrm{L}, v^{c_j}_d)$;
  else if $v^{c_j}_d \notin V \wedge v^{c_j}_a \notin V$ **then** // Case II) - Both Removed
    $V \leftarrow V \cup \{v^{c_j}_d\}$; // Add $v^{c_j}_d$ to G
    Execute RewireWaitingDep$(G, v^{c_j}_d, \mathrm{S}_s)$; // i.e. Algorithm 4
    Execute RewireTransferDep$(G, v^{c_j}_d, \text{SUCC}, \mathrm{S}_s)$; // i.e. Algorithm 3
    L $\leftarrow$ INC-BU$(G, \mathrm{L}, v^{c_j}_d)$;
    $V \leftarrow V \cup \{v^{c_j}_a\}$; $A \leftarrow A \cup \{(v^{c_j}_d, v^{c_j}_a)\}$; // Add $v^{c_j}_a$ and connection arc to G
    Execute RewireArr$(G, v^{c_j}_a, \text{TRIP}i, \mathrm{S}_t)$; // i.e. Algorithm 5
    L $\leftarrow$ INC-BU$(G, \mathrm{L}, v^{c_j}_a)$;
  else if $v^{c_j}_d \notin V \wedge v^{c_j}_a \in V$ **then** // Case III) - Only $v^{c_j}_d$ removed
    $V \leftarrow V \cup \{v^{c_j}_d\}$;
    $A \leftarrow A \cup \{(v^{c_j}_d, v^{c_j}_a)\}$; // Add $v^{c_j}_d$ and connection arc to G
    Execute RewireWaitingDep$(G, v^{c_j}_d, \mathrm{S}_s)$; // i.e. Algorithm 4
    Execute RewireTransferDep$(G, v^{c_j}_d, \text{SUCC}, \mathrm{S}_s)$; // i.e. Algorithm 3
    L $\leftarrow$ INC-BU$(G, \mathrm{L}, v^{c_j}_d)$;
  else // Case IV) - Only $v^{c_j}_a$ removed
    $V \leftarrow V \cup \{v^{c_j}_a\}$; $A \leftarrow A \cup \{(v^{c_j}_d, v^{c_j}_a)\}$; // Add $v^{c_j}_a$ and connection arc to G
    Execute RewireArr$(G, v^{c_j}_a, \text{TRIP}i, \mathrm{S}_t)$; // i.e. Algorithm 5
    L $\leftarrow$ INC-BU$(G, \mathrm{L}, v^{c_j}_a)$;

Algorithm 3: Algorithm RewireTransferDep.

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

```
1  if SUCC = ∞ then                                              // Sub-case I.a)
2      CANDIDATES ← ∅;
3      foreach v ∈ AV[s] do
4          TO_ADD ← true;
5          foreach u ∈ N^G_out(v) do                  // Outgoing transfer arcs (if any)
6              if u ∈ DV[s] then                                  // Has transfer arc
7                  TO_ADD ← false;
8                  break;
9          if TO_ADD ∧ time(v_d^{c_j}) ≥ time(v) + MTT_s then
10             CANDIDATES ← CANDIDATES ∪ {v}
11     foreach v ∈ CANDIDATES do
12         A ← A ∪ {(v, v_d^{c_j})}
13 else                                                          // Sub-case I.b)
14     T ← ∅;
15     foreach v ∈ AV[s] do
16         foreach u ∈ N^G_out(v) do                  // Outgoing transfer arcs (if any)
17             if u = SUCC ∧ time(v_d^{c_j}) ≥ time(v) + MTT_s then
18                 T ← T ∪ {(v, u)};
19     foreach (v, u) ∈ T do
20         A ← A \ {(v, u)};
21         A ← A ∪ {(v, v_d^{c_j})};
```

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 $v_d^{c_j}$

Algorithm 4: Algorithm RewireWaitingDep.

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

```
1  if DV[s] \ {v_d^{c_j}} ≠ ∅ then
2      m ← arg max_{v∈DV[s]} time(v);
3      if time(v_d^{c_j}) ≥ time(m) then                          // Add waiting arc
4          A ← A ∪ {(m, v_d^{c_j})};
5  else
6      Let m_1, m_2 ∈ DV[s] be such that time(m_1) ≤ time(v_d^{c_j}) ≤ time(m_2);
7      A ← A \ {(m_1, m_2)};                                  // Remove outdated waiting arc
8      A ← A ∪ {(m_1, v_d^{c_j}), (v_d^{c_j}, m_2)};                  // Add new waiting arcs
```

to V and to 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 DV$[s]$ and AV$[s]$ and on their times. In particular, if $time(v_d^{c_j}) \geq time(v)\ \forall\ v \in$ DV$[s]$ and DV$[s] \setminus \{v_d^{c_j}\} \neq \emptyset$, i.e. there is no waiting arc outgoing vertex $m = \arg\max_{v \in \text{DV}[s]} time(v)$ and there exists another departure vertex

besides $v_d^{c_j}$ in 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 $m_1, m_2 \in$ 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 $time(v_d^{c_j}) < time(v)\ \forall\ v \in$ 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 3, 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 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 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 TRIPi. 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 TRIPi (see Algorithm 5 for the pseudo-code of this phase). Again, we run INC-BU to update L (see Line 20).

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 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 3 and 4, discussed above. Algorithm INC-BU is also run to reflect changes on the 2HC-R labeling (see Line 26).

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 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 5. 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 Fig. 5.2. In addition, for the sake of simplicity in understanding, we show an example of execution of the procedures for: i): rewiring transfer arcs (Algorithm 3) and waiting arcs (Algorithm 4) to a departure vertex in Fig. 5.3; and ii): rewiring arcs to an arrival vertex (Algorithm 5) in Fig. 5.4.

After executing the insertion phase (preceded by the execution of the removal phase), G is a RED-TE graph (Lemma 5.1) and L is a 2HC-R labeling of G (Lemma 5.2).

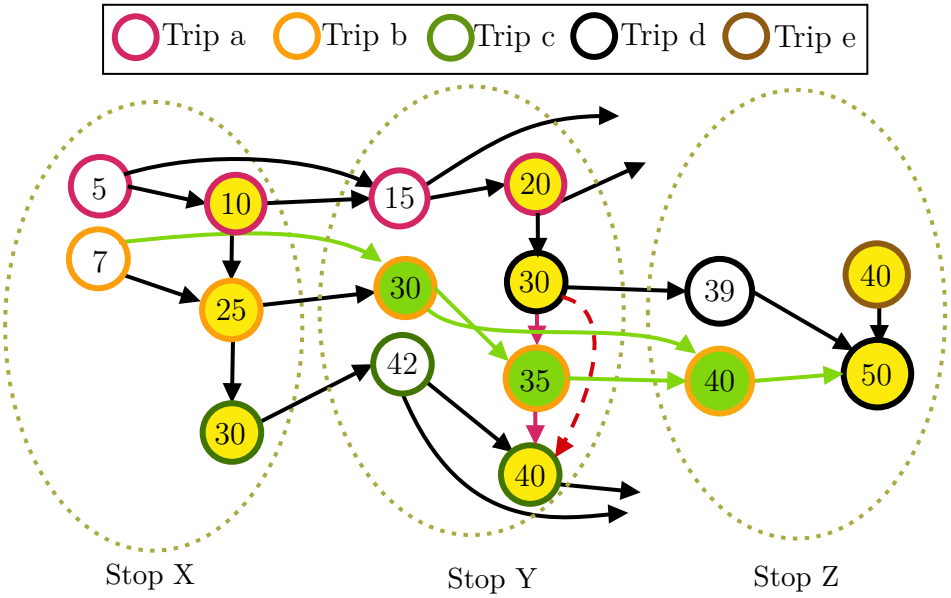

FIGURE 5.2: The RED-TE graph obtained after performing Algorithm INS-D-PTL (Algorithm 2) on the graph of Figure 5.1. Newly added vertices and arcs are drawn in green. The dashed arc drawn in red is the waiting arc of Fig. 5.1 that is removed in the insertion phase.

Algorithm 5: Algorithm REWIREARR.

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

1 MIN_NODE $\leftarrow \infty$;
2 MIN_NODE $\leftarrow \underset{v \in DV[t]\ ,\ time(v) \geq time(v_a^{c_j}) + MTT_t}{\arg\min} \{time(v)\}$;
3 **if** MIN_NODE $\neq \infty$ **then** // there exist a valid departure
4 $\quad A \leftarrow A \cup \{(v_a^{c_j}, \text{MIN\_NODE})\}$; // Add proper transfer arc
5 **if** $j > 0$ **then** // Not first connection of the trip
6 $\quad A \leftarrow A \cup \{(v_a^{c_{j-1}}, v_a^{c_j})\}$; // Add bypass arc
7 **if** $j < k$ **then** // Not last connection of the trip
8 $\quad A \leftarrow A \cup \{(v_a^{c_j}, v_a^{c_{j+1}})\}$; // Add bypass arc

Lemma 5.1. *Given a* RED-TE *graph* $G = (V, A)$ *of an input timetable* $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$. *Assume* $c_j \in \mathcal{C}$ *is a connection served in a trip* TRIP_i *such that* $v_d^{c_j} \in DV[s]$ ($v_a^{c_j} \in AV[t]$, *respectively), where* s *is a generic stop in* $\mathcal{S}$ (t *is a generic stop in* $\mathcal{S}$, *respectively). If* c_j *is delayed for* δ *amount of time with* $\delta > 0$, *then let the updated timetable be as* $\mathcal{T}' = (\mathcal{Z}, \mathcal{S}, \mathcal{C}')$, *and let* $G' = (V', A')$ *be the resultant graph obtained after the execution of Algorithm* REM-D-PTL *(see Algorithm 1) and* INS-D-PTL *(see Algorithm 2). We claim that* G' *is a* RED-TE *graph of the updated timetable* $\mathcal{T}'$.

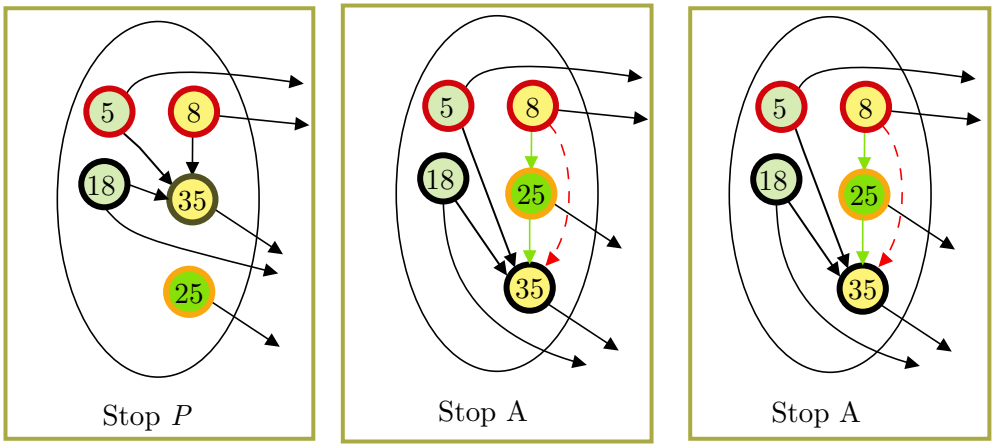

FIGURE 5.3: An example of execution of the procedure for rewiring transfer and waiting arcs given in Algorithm 3 and Algorithm 4 , respectively. On the left we show part of a sample graph, relative to a stop P with the assumption that $\text{MTT}_P = 5$ minutes, that is violating the RED-TE properties. 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 3 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 green are the newly added ones.

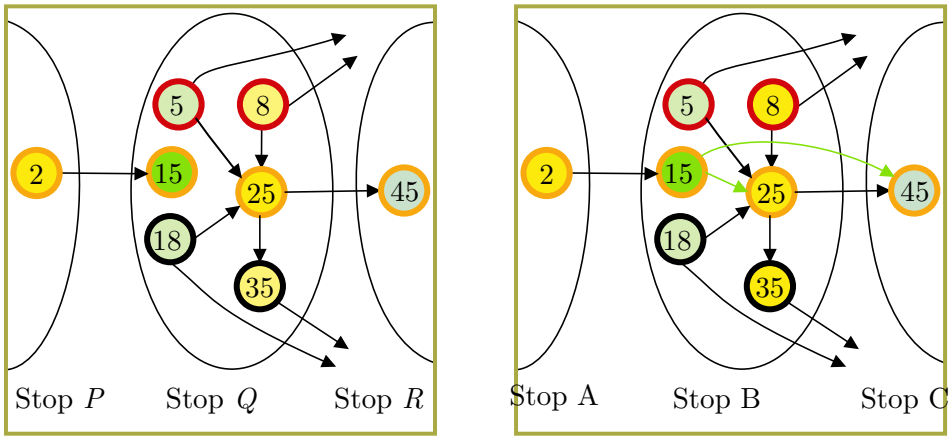

FIGURE 5.4: Part of a sample graph, relative to three stops, namely P, Q and R, is shown on the left side, where the minimum transfer time is assumed to be, for the sake of simplicity, $\text{MTT}_P = \text{MTT}_Q = \text{MTT}_R = 5$ minutes for all stops. Both transfer and bypass arcs for the vertex of $\text{AV}[Q]$ highlighted in green must be rewired, in order to restore RED-TE properties. To this end, Algorithm 5 is executed, and the result is shown on the right, with newly added arcs are highlighted in green.

Proof. For showing that the above hold, it is enough to show that G, after the execution of Algorithms 1 and 2, is a RED-TE graph in any of the two possible cases: A) c_j is the last connection in TRIP_i; and B) c_j is not the last connection in TRIP_i. Below, we discuss each of these cases in detail.

Discussion on Case A. In this case, occurring when c_j is the last connection in TRIP_i, we can have one of the four possible sub-cases: i) both $v_d^{c_j}$ and $v_a^{c_j}$ are affected by the delay of δ time units; ii) both $v_d^{c_j}$ and $v_a^{c_j}$ are not affected by the delay of δ time units; and iii) only $v_d^{c_j}$ is affected by the delay of δ time units; and iv) only $v_a^{c_j}$ is affected by the delay of δ time units.

In the sub-case i), occurring when both $v_d^{c_j}$ and $v_a^{c_j}$ are affected by the delay, both vertices are removed from G (Lines 13- 20 of Algorithm 1). Waiting and transfer arcs are then rewired using Algorithms 4 and 3 (respectively), called as a sub-routine by Algorithm 1. Note that, such arcs rewiring procedures (along with that shown in Algorithm 5) have been shown to result in correctly updating a given RED-TE graph (see [29]). After the removal of these vertices, Algorithm 2 adds new vertices for representing the delayed connection, and rewire the arcs once again via the procedures of [29], and therefore the RED-TE properties of G are restored. In the sub-case ii), occurring when both $v_d^{c_j}$ and $v_a^{c_j}$ are not affected by the delay, none of the vertices are removed from G during the execution of Algorithm 1. However, Algorithm 2 updates the transfer arcs using Algorithm 3 called as a sub-routine. In the sub-case iii), instead, Algorithm 1 remove $v_d^{c_j}$ from G, and rewire the waiting and transfer arcs using Algorithms 4 and 3 called as a sub-routine. In particular, it adds $(\text{PRED}, \text{SUCC})$ to A if both PRED and SUCC exist in $\text{DV}[s]$. It then rewire the transfer arcs using Algorithm 3, once again called as a sub-routine. Once the transfer arcs are rewired, Algorithm 2 then add a new vertex for representing the delayed arrival event, and wire arcs using the procedure in [29], thus restoring the RED-TE properties. Finally, in the sub-case iv), Algorithm 1 removes $v_a^{c_j}$ from G. Algorithm 2 then add a new vertex (let's say v) and a connection arc $(v_d^{c_j}, v)$ to G. It then rewire the rest of the arcs using the procedure in Algorithm 5 to restore the RED-TE properties. Following these, it is easy to conclude that G, after the execution of Algorithms 1 and 2, is a RED-TE graph.

Discussion on Case B. In this case, occurring when c_j is not the last connection in TRIP_i, we repeat the same removal and insertion phases for each such connection c_m, with $j \geq m \leq k$, in TRIP_i, where c_k is the last connection in TRIP_i. In this case, as connections are considered in a sequence, and that after each such cycle, the RED-TE properties are restored using the procedures of [29]. Therefore, it is easy to conclude that G is a RED-TE graph after executing both the Algorithm REM-D-PTL and INS-D-PTL. This completes our proof. □

Lemma 5.2. *Given a* RED-TE *graph* $G = (V, A)$ *of a timetable* $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$*, and a* 2HC-R *labeling* L *of* G*. If a connection* $c_i \in \mathcal{C}$ *is delayed for* δ *amount of time, then after the execution of Algorithm* REM-D-PTL *(see Algorithm 1) followed by the execution of Algorithm* INS-D-PTL *(see Algorithm 2), let* $G' = (V', A')$ *be the updated* RED-TE *graph, and* L$'$ *be the updated* 2HC-R *labeling. We claim that,* L$'$ *is a* 2HC-R *labeling of the* RED-TE *graph* G'*.*

Proof. Notice that, the above statement is based on the correctness of the approach in [105]. In particular, throughout Algorithms 1 and 2, 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 [105]. Hence, at the end of Algorithms 1 and 2, L$'$ is a 2HC-R labeling for G'. □

5.1.3 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 e.g. the routine in [41]. This computational effort is not large as that required for recomputing the 2HC-R labeling. However, we propose an 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 1–2, to compute two sets of so–called *updated stops*, denoted, resp., by US_{out} and US_{in}. These are defined as the stops $\text{S}_i \in \mathcal{S}$ such that vertices in DV$[i]$ (AV$[i]$, resp.) had their time value or forward label (backward label, resp.) changed during Algorithm REM-D-PTL or 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 1–2, after each update to times or labels.

Once this is done we update the stop labeling SL by recomputing only the entries of $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, resp.) for each $\text{S}_i \in \text{US}_{out}$ (for each $\text{S}_i \in \text{US}_{in}$, resp.). To this aim, for each stop $\text{S}_i \in \text{US}_{out}$ ($\text{S}_i \in \text{US}_{in}$, resp.) we first reset $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, resp.) to the empty set. Then, we scan departure (arrival, resp.) vertices in decreasing (increasing, resp.) order with respect to time and add entries to $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, resp.) accordingly. In particular, for all departure (arrival, resp.) vertices v of S_i in the above mentioned order, we add a pair $(u, \mathit{stoptime}_i(v))$ for each u in $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, resp.) if there is no pair $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, resp.) having u as hub vertex. This guarantees that each pair contains the latest departure (earliest arrival, resp.) 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 [41]. Details on how to update the stop labeling by executing the procedure are given in Algorithm 6. We are now ready to give the following results.

Algorithm 6: 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

```
1  foreach S_i ∈ US_out do
2      Q ← ∅;
3      SL_out(i) ← ∅;
4      while {DV[s] \ Q} ≠ ∅ do
5          m ← arg max_{v∈DV[s]} time(v);
6          foreach u ∈ L_out(m) do
7              if u ∉ SL_out(i) then
8                  SL_out(i) ← SL_out(i) ∪ {(u, time(m))};
9          Q ← Q ∪ {m};
10 Sort SL_out(i) with respect to vertices ids;
11 foreach S_i ∈ US_in do
12     Q ← ∅;
13     SL_in(i) ← ∅;
14     while {AV[s] \ Q} ≠ ∅ do
15         m ← arg min_{v∈DV[s]} time(v);
16         foreach u ∈ L_in(m) do
17             if u ∉ SL_in(i) then
18                 SL_in(i) ← SL_in(i) ∪ {(u, time(m))};
19         Q ← Q ∪ {m};
20 Sort SL_in(i) with respect to vertices ids;
```

Theorem 5.3 (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, i.e. 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$'$*.*

Proof. We prove the correctness of each of the above claims as follows. Concerning i), we have shown that after the execution of Algorithms 1 and 2, G' is a red-te graph (see Lemma 5.1). Concerning ii), we already proved that L$'$ is a 2hc-r labeling of G' (see Lemma 5.2). Finally, concerning iii), Algorithm 6 applies the definition of stop labeling, by updating the entry of a stop with the proper hub vertices and times values. In particular, Algorithm 6 recompute $\text{SL}_{in}(i)$ ($\text{SL}_{out}(j)$, respectively) for each $i \in \text{US}_{in}$ (for each $j \in \text{US}_{out}$, respectively). We know that both US_{in} and US_{out}, by definition, contain those stops in $\mathcal{S}$ whose label sets contain obsolete label entries. Using proof via

contradiction, we show that SL' is a stop labeling for L'. To this end, we assume that, SL'' is the stop labeling, computed from scratch, for L'. We also assume that there exist a stop $\text{S}_i \in \mathcal{S}$ such that $\text{SL}_{in}(i) \in \text{SL}''$ ($\text{SL}_{out}(i) \in \text{SL}''$, respectively) and $\text{SL}_{in}(i) \in \text{SL}'$ ($\text{SL}_{out}(i) \in \text{SL}'$, respectively) are not equivalent. We proceed by discussing at first the scenario where $\text{SL}_{in}(i) \in \text{SL}''$ and $\text{SL}_{in}(i) \in \text{SL}'$ are not equivalent. In this scenario, we can have two possible cases, that is: A) both label sets $\text{SL}_{in}(i) \in \text{SL}''$ and $\text{SL}_{in}(i) \in \text{SL}$ are equivalent; and B): both $\text{SL}_{in}(i) \in \text{SL}''$ and $\text{SL}_{in}(i) \in \text{SL}$ differs from each other in at least one of the label entry. In what follows, we discuss each of these cases in detail.

Discussion on Case A. In this case, occurring when both $\text{SL}_{in}(i) \in \text{SL}''$ and $\text{SL}_{in}(i) \in \text{SL}$ are equivalent, the following always hold: a) for all vertices $v \in \text{AV}[i]$, we have that $\text{L}_{in}(v) \in \text{L}$ and $\text{L}_{in}(v) \in \text{L}'$ are equivalent; and b) the time values associated with the vertices in $\text{AV}[i]$ have remained unchanged throughout the execution of the Algorithms 1 and 2. These two conditions essentially means that, US_{in} by definition cannot have S_i as its element. Therefore, $\text{SL}_{in}(i) \in \text{SL}'$ is never updated during the execution of the Algorithm 6, and hence it contradicts with our assumption that $\text{SL}_{in}(i) \in \text{SL}''$ is not equivalent to that in SL'.

Discussion on Case B. In this case, occurring when both $\text{SL}_{in}(i) \in \text{SL}''$ and $\text{SL}_{in}(i) \in \text{SL}$ differs from each other in at least one of the label entry, either one or both of the following must be true, that is: a) there exists at least one vertex $v \in \text{AV}[i]$ such that $\text{L}_{in}(v) \in \text{L}$ and $\text{L}_{in}(v) \in \text{L}'$ are not equivalent; and/or b) there exist at least one vertex in $\text{AV}[i]$ whose associated time value has been modified during the execution of the Algorithms 1 and 2. If one of these conditions hold, then US_{in} by definition must contain S_i as its element. We know that, if $\text{S}_i \in \text{US}_{in}$, then $\text{SL}_{in}(i)$ is recomputed from scratch, and hence this contradicts our assumption that $\text{SL}_{in}(i) \in \text{SL}''$ is not equivalent to that in SL'.

The above reasoning can be easily extended for scenario where $\text{SL}_{out}(i) \in \text{SL}''$ is not equivalent to that in SL'. This completes our proof. □

Theorem 5.4 (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 6. In what follows, we analyze separately the three algorithms.

Concerning Algorithm 1, we first bound the cost of executing Lines 1–21, i.e. 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 [105]. 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 the connection c_m to c_k, therefore in the worst case for all stops of the transit network, which are $|\mathcal{S}| \leq |\mathcal{C}|$. 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 takes $\mathcal{O}(|\mathcal{C}|)$, as $|\mathcal{C}| \geq \max\{|\mathcal{S}|, |\mathcal{Z}|\}$ 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 $|\mathcal{S}| \leq |\mathcal{C}|$ 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 6, it scans label entries of vertices in both US_{in} and US_{out} in non–increasing and non–decreasing order, resp. (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 6 is $\mathcal{O}(|\mathcal{C}| \log |\mathcal{C}|)$. If we sum up the complexities of Algorithms 1, 2, and 6, the claim follows. □

Notice that, Theorem 5.4 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 [36].

However, our experimental study, which is described in Chapter 6, clearly shows that D-PTL always outperforms PTL in practice.

5.1.4 Example

In this Section, to better clarify how basic D-PTL updates the RED-TE graph G, the 2HC-R labels L and its compact form SL for a given public transit network after a delay is introduced in one of the connections in $\mathcal{C}$, we discuss an example scenario. In particular, for the sake of clarity, we consider the timetable presented in Table 3.1, whose corresponding RED-TE graph G is shown in Fig. 3.1, in Chapter 3. Concerning L, for the sake of understanding, we assume that vertices in V are sorted using degree

in descending order (shown in Table 5.1), and that L is computed using BUTTERFLY algorithm (shown in Table 5.2).

For the sake of understanding, we show the RED-TE graph with the ordering information on each vertex $v_i \in V$ (see Fig. 5.5). It is noteworthy to mention that, we resolve ambiguities arise during ordering vertices at random. It is also noteworthy to mention that, when computing L, we assume that, for any two vertices v_i and v_j, with $j > i$ and $\text{DEG}(v_i) \leq \text{DEG}(v_j)$, we have that $l(v_i) < l(v_j)$. Concerning the compact form of L, we assume that SL is computed via the routine in [41] (see Table 5.3).

Concerning the delay, we assume a delay δ of ten minutes is added to the departure time of the last connection in TRIP served by vehicle with id β. More precisely, we assume the connection $c_k = (\beta, s_Y, s_Z, 25, 30)$ be now given as $c'_k = (\beta, s_Y, s_Z, 35, 40)$.

We are now ready to apply the basic D-PTL algorithm to our example scenario.

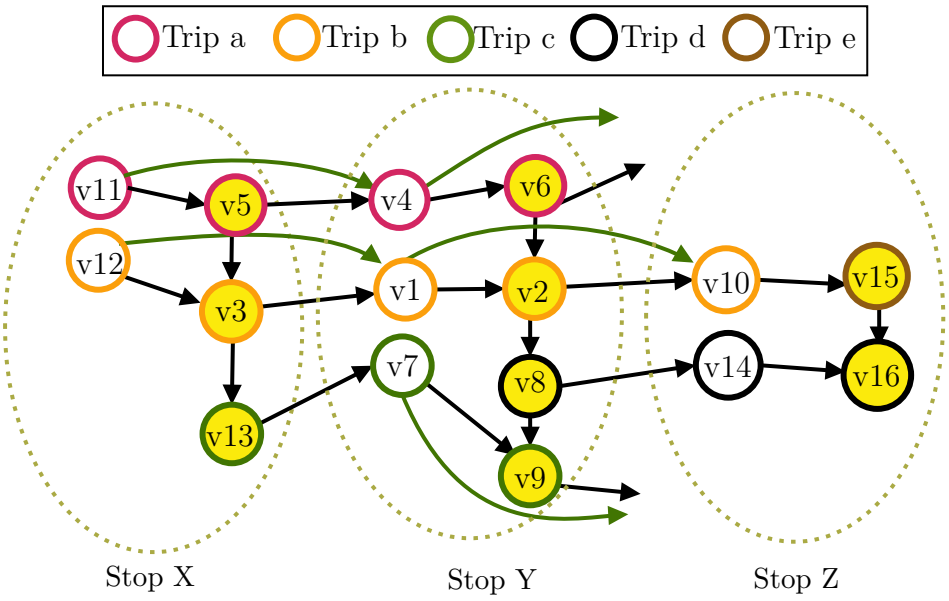

FIGURE 5.5: RED-TE graph, augmented with vertex ordering information, for the timetable in Table 3.1. In particular, a vertex v_i essentially means that it is the i^{th} vertex in V w.r.t the ordering.

5.1.4.1 Applying DPTL

The basic D-PTL algorithm receives as an input the RED-TE graph G, L and the delayed connection c_k along with the amount of delay δ, and produces as an output the updated RED-TE graph G' and 2HC-R labeling L (both updated via the underlying removal and insertion phases), and the updated stop labels SL.

In what follows, we describe the working of the basic D-PTL algorithm in detail.

Stops	Trips	Departure/Arrival	Vertices	$time(v_*)$
Y	b	Arrival	v_1	20
Y	b	Departure	v_2	25
X	b	Departure	v_3	15
Y	a	Arrival	v_4	15
X	a	Departure	v_5	10
Y	a	Departure	v_6	20
Y	c	Arrival	v_7	42
Y	d	Departure	v_8	30
Y	c	Departure	v_9	50
Z	b	Arrival	v_{10}	30
X	a	Arrival	v_{11}	5
X	b	Arrival	v_{12}	7
X	c	Departure	v_{13}	35
Z	d	Arrival	v_{14}	39
Z	e	Departure	v_{15}	40
Z	d	Departure	v_{16}	50

TABLE 5.1: Ordering of vertices w.r.t degree (in+out) in descending order. The first column provides information about the stop. The second and third column provides information about trips and the type of event associated with the corresponding vertex in each row. The fourth column provides information about the ordering of each vertex $v \in V$. In particular, v_i essentially represents that it is at i^{th} position in V w.r.t the ordering. Finally, the last column provides information about the time value associated with each vertex.

Vertices	2HC-R Labeling (L)	
	$\mathrm{L}_{in}(v_*)$	$\mathrm{L}_{out}(v_*)$
v_1	$\{v_1\}$	$\{v_1\}$
v_2	$\{v_1, v_2\}$	$\{v_2\}$
v_3	$\{v_3\}$	$\{v_1, v_3\}$
v_4	$\{v_4\}$	$\{v_2, v_4\}$
v_5	$\{v_5\}$	$\{v_1, v_3, v_4, v_5\}$
v_6	$\{v_4, v_6\}$	$\{v_2, v_6\}$
v_7	$\{v_3, v_7\}$	$\{v_7\}$
v_8	$\{v_1, v_2, v_8\}$	$\{v_8\}$
v_9	$\{v_1, v_2, v_7, v_8, v_9\}$	$\{v_9\}$
v_{10}	$\{v_1, v_2, v_{10}\}$	$\{v_{10}\}$
v_{11}	$\{v_{11}\}$	$\{v_1, v_2, v_3, v_4, v_5, v_{11}\}$
v_{12}	$\{v_{12}\}$	$\{v_1, v_3, v_{12}\}$
v_{13}	$\{v_3, v_{13}\}$	$\{v_7, v_{13}\}$
v_{14}	$\{v_1, v_2, v_8, v_{14}\}$	$\{v_{14}\}$
v_{15}	$\{v_1, v_2, v_{10}, v_{15}\}$	$\{v_{15}\}$
v_{16}	$\{v_1, v_2, v_8, v_{10}, v_{14}, v_{15}, v_{16}\}$	$\{v_{16}\}$

TABLE 5.2: 2HC-R labeling for the RED-TE graph shown in Fig. 5.5 by following the vertex ordering presented in Table 5.1.

Stops	Stop Labeling (SL)	
	$\mathrm{SL}_{in}(*)$	$\mathrm{SL}_{out}(*)$
X	$\{(v_{11}, 5), (v_{12}, 7)\}$	$\{(v_5, 10), (v_3, 15), (v_4, 10), (v_1, 15), (v_{13}, 35), (v_7, 35)\}$
Y	$\{(v_3, 42), (v_4, 15), (v_1, 20), (v_7, 42)\}$	$\{(v_6, 20), (v_2, 25), (v_8, 30), (v_9, 50)\}$
Z	$\{(v_1, 30), (v_2, 30), (v_8, 39),$ $(v_{10}, 30), (v_{14}, 39)\}$	$\{(v_{15}, 40), (v_{16}, 50)\}$

TABLE 5.3: Stop labeling for the RED-TE graph shown in Fig. 5.5 built on-top of the 2HC-R labeling presented in Table 5.2

Removal Phase From our above discussion, we know that in this phase, the basic D-PTL procedure shown in Algorithm 1 removes the affected vertices, if any as a result of the delay, from G and accordingly update L. In our case, given δ and c_k, with v_2 as the departure vertex and v_{10} as the arrival vertex, the algorithm proceeds by updating $time(v_2)$ and $time(v_{10})$. In particular, it increments $time(v_2)$ and $time(v_{10})$ by adding δ to their respective values (Lines 5–6 of Algorithm 1). After which, it checks whether v_2 is affected by the delay or not. To this end, the algorithm computes a successor vertex (denoted using SUCC) of v_2 in DV$[Y]$ (Line 8 of Algorithm 1), i.e. v_8 in our case, and checks if $time(v_2)$ is greater than $time(v_8)$. In the affirmative case, it considers v_2 as affected. In our example, $time(v_2)$ ($time(v_8)$, respectively) has a value of 35 (30, respectively), therefore the delay effects v_2. The algorithm then computes the predecessor of v_2 (denoted using PRED) in DV$[Y]$ (Lines 9–12 of Algorithm 1), i.e. v_6 in our case. In addition to this, Lines 9–12 in the same procedure also compute a set of vertices $\overline{A}$, i.e. a subset of AV$[Y]$ such that for each vertex $v_i \in \overline{A}$, there exists an arc (v_i, v_2) in A . In our case, $\overline{A}$ contains only v_1 as it is the only such vertex in AV$[Y]$ with a transfer arc incident to v_2. The algorithm then removes v_2 from G, and update L (shown in Table 5.4) using DEC-BU called as a sub-routine (Line 14 of Algorithm 1).

For restoring the RED-TE properties, the algorithm adds a waiting arc (PRED, SUCC) to A (Line 15 of Algorithm 1), i.e. in our case is given as (v_6, v_8). In addition to this, the algorithm also adds transfer arcs (v, SUCC) for each $v \in \overline{A}$. In our case, the transfer arc (v_1, v_8) is added to A (see the dotted arcs in Fig. 5.6). The algorithm then update L (shown in Table 5.5) using INC-BU called as a sub-routine (Line 17 of Algorithm 1).

Vertices	2HC-R Labeling (L)	
	$L_{in}(v_*)$	$L_{out}(v_*)$
v_1	$\{v_1\}$	$\{v_1\}$
$v_2{}^-$	$\{v_1, v_2\}^-$	$\{v_2\}^-$
v_3	$\{v_3\}$	$\{v_1, v_3\}$
v_4	$\{v_4\}$	$\{v_4\}^\star$
v_5	$\{v_5\}$	$\{v_1, v_3, v_4, v_5\}$
v_6	$\{v_4, v_6\}$	$\{v_6\}^\star$
v_7	$\{v_3, v_7\}$	$\{v_7\}$
v_8	$\{v_8\}^\star$	$\{v_8\}$
v_9	$\{v_3, v_7, v_8, v_9\}^\star$	$\{v_9\}$
v_{10}	$\{v_1, v_{10}\}^\star$	$\{v_{10}\}$
v_{11}	$\{v_{11}\}$	$\{v_1, v_3, v_4, v_5, v_{11}\}^\star$
v_{12}	$\{v_{12}\}$	$\{v_1, v_3, v_{12}\}$
v_{13}	$\{v_3, v_{13}\}$	$\{v_7, v_{13}\}$
v_{14}	$\{v_8, v_{14}\}^\star$	$\{v_{14}\}$
v_{15}	$\{v_1, v_{10}, v_{15}\}^\star$	$\{v_{15}\}$
v_{16}	$\{v_1, v_8, v_{10}, v_{14}, v_{15}, v_{16}\}^\star$	$\{v_{16}\}$

TABLE 5.4: Updated 2HC-R labeling after v_2 is removed from the RED-TE graph and DEC-BU(G, v_2, L) is executed. In details, labels and vertices with superscript "$\star$" are those updated in the decremental update procedure, while those with superscript "$-$" represent newly added labels accordingly.

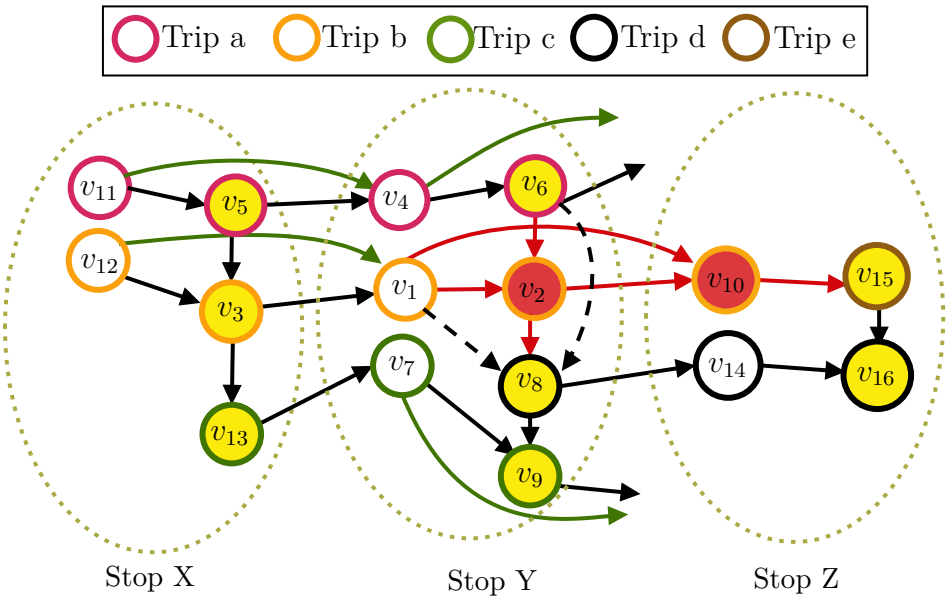

FIGURE 5.6: Visualizing the effects of basic D-PTL's removal phase on the RED-TE graph shown in Fig. 5.5. Vertices filled with Red color are the affected vertices and are therefore removed along with the arcs colored in Red. Dotted arcs are the newly added arcs.

Concerning the arrival vertex, that in our case is v_{10}, the algorithm performs a check to verify if it is affected by the delay. In particular, the algorithm conducts a search for a transfer arc adjacent to v_{10} (Line 18 of Algorithm 1), i.e. (v_{10}, v_{15}), and checks if $time(v_{10}) > time(v_{15}) + \text{MTT}_Z$ (Line 19). In our case, $time(v_{10}) > time(v_{15}) + \text{MTT}_Z$ holds, therefore v_{10} is affected by the delay. The algorithm then removes v_{10} from G (shown in Fig. 5.6), and update L (shown in Table 5.6) using DEC-BU called as a sub-routine (Line 21 of Algorithm 1). The resultant RED-TE graph, at the end of the removal phase, is shown in Fig. 5.7.

Insertion Phase After the execution of the removal phase, the basic D-PTL's insertion phase (see Algorithm 2) inserts new vertices and arcs in G and update accordingly L. In what follows, we describe the description of the basic D-PTL insertion phase applied to our example scenario.

From our discussion in Section 5.1.4.1, we know that, both the departure and arrival vertices, i.e. v_2 and v_{10} (respectively), were affected by the delay and were therefore removed from G in the removal phase. For adding the representation of c'_k (given as $c'_k = (\beta, \text{S}_Y, \text{S}_Z, 35, 40)$) in G, the algorithm inserts new vertices and arcs into G in a way to restore the RED-TE properties, and update accordingly L (Lines 13–20).

In details, for representing the updated departure event, the algorithm adds a new vertex to V. For the sake of simplicity, we assume the newly added vertex be the last vertex

Vertices	2HC-R Labeling (L)	
	$L_{in}(v_*)$	$L_{out}(v_*)$
v_1	$\{v_1\}$	$\{v_1\}$
v_3	$\{v_3\}$	$\{v_1, v_3\}$
v_4	$\{v_4\}$	$\{\ v_4\ \}$
v_5	$\{v_5\}$	$\{v_1, v_3, v_4, v_5\}$
v_6	$\{v_4, v_6\}$	$\{v_6\ \}$
v_7	$\{v_3, v_7\}$	$\{v_7\}$
v_8	$\{v_1, v_4, v_6, v_8\}^\star$	$\{v_8\}$
v_9	$\{v_1, v_4, v_7, v_8, v_9\}^\star$	$\{v_9\}$
v_{10}	$\{\ v_1,\ v_{10}\ \}$	$\{v_{10}\}$
v_{11}	$\{v_{11}\}$	$\{\ v_1,\ v_3,\ v_4,\ v_5,\ v_{11}\ \}$
v_{12}	$\{v_{12}\}$	$\{v_1, v_3, v_{12}\}$
v_{13}	$\{v_3, v_{13}\}$	$\{v_7, v_{13}\}$
v_{14}	$\{v_1, v_4, v_6, v_8, v_{14}\}^\star$	$\{v_{14}\}$
v_{15}	$\{v_1, v_{10}, v_{15}\}^\star$	$\{v_{15}\}$
v_{16}	$\{v_1, v_4, v_6, v_8, v_{10}, v_{14}, v_{15}, v_{16}\}^\star$	$\{v_{16}\}$

TABLE 5.5: 2HC-R labels, after the new waiting arc (v_6, v_8) and the transfer (v_1, v_8) are added to G and INC-BU(G, v_8, L) is executed. The label sets marked with superscript "$\star$" are those modified by the incremental update procedure.

Vertices	2HC-R Labeling (L)	
	$L_{in}(v_*)$	$L_{out}(v_*)$
v_1	$\{v_1\}$	$\{v_1\}$
v_3	$\{v_3\}$	$\{v_1, v_3\}$
v_4	$\{v_4\}$	$\{\ v_4\ \}$
v_5	$\{v_5\}$	$\{v_1, v_3, v_4, v_5\}$
v_6	$\{v_4, v_6\}$	$\{v_6\ \}$
v_7	$\{v_3, v_7\}$	$\{v_7\}$
v_8	$\{\ v_1,\ v_4,\ v_6,\ v_8\ \}$	$\{v_8\}$
v_9	$\{v_1,\ v_4,\ v_7,\ v_8,\ v_9\ \}$	$\{v_9\}$
$v_{10}{}^-$	$\{v_1, v_{10}\}^-$	$\{v_{10}\}^-$
v_{11}	$\{v_{11}\}$	$\{\ v_1,\ v_3,\ v_4,\ v_5,\ v_{11}\ \}$
v_{12}	$\{v_{12}\}$	$\{v_1, v_3, v_{12}\}$
v_{13}	$\{v_3, v_{13}\}$	$\{v_7, v_{13}\}$
v_{14}	$\{v_1,\ v_4,\ v_6,\ v_8,\ v_{14}\ \}$	$\{v_{14}\}$
v_{15}	$\{v_{15}\}^\star$	$\{v_{15}\}$
v_{16}	$\{v_1, v_4, v_6, v_8, v_{14}, v_{15}, v_{16}\}^\star$	$\{v_{16}\}$

TABLE 5.6: Updated 2HC-R labeling after v_{10} is removed from the RED-TE graph (see Fig.5.7) and DEC-BU(G, v_{10}, L) is executed. In details, labels and vertices with superscript "$\star$" are those updated in the decremental update procedure, while those with superscript "$-$" represent newly added labels accordingly.

in V w.r.t the ordering (shown in Table 5.1) we used for computing L. It is worthy to mention that, in practice, for the sake of obtaining minimized labeling size after it is updated, it is useful to update the ordering of existing vertices, if required after the addition of a new vertex. To this end, specific to 2HC-R labeling, various techniques have been presented in [105], and are experimentally shown to be effective. For details about such techniques, we refer the interested reader to [105], and references therein. To proceed with our discussion, we assume the newly added vertex be v_{17}.

The algorithm, after adding v_{17} to V, assigns $time(v_{17})$ a value equal to the departure time given in c'_k, i.e. 35. Then, waiting arcs are rewired using the procedure shown in

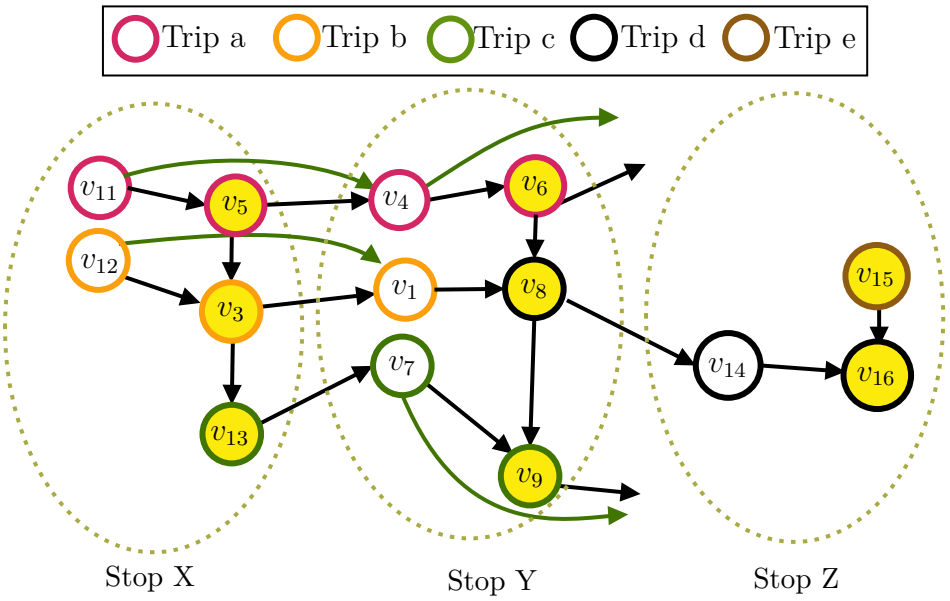

FIGURE 5.7: The RED-TE graph (corresponding to that shown in Fig. 5.5) after executing basic D-PTL's removal phase. The affected vertices v_2 and v_{10} are removed, and the RED-TE properties are restored by adding a new waiting arc (v_6, v_8) and a new transfer arc (v_1, v_8) to G.

Algorithm 4 called as a sub-routine by Algorithm 2. In details, the procedure in Algorithm 4 performs a test to check if v_{17} is the last vertex in DV$[Y]$ (Line 3 of Algorithm 4). In our example, v_{17} is not the last vertex in DV$[Y]$, therefore the algorithm computes two vertices, referred to as m_1 and m_2, such that $time(m_1) \leq time(v_d^{c_j}) \leq time(m_2)$ (Line 6 of Algorithm 4). In our case, m_1 and m_2 are computed as v_8 and v_9, respectively. It then removes the waiting arc (v_8, v_9) from A, and add two new waiting arcs (v_8, v_{17}) and (v_{17}, v_9) in A. Waiting arcs are then rewired using Algorithm 4 called as sub-routine by Algorithm 2. In details, the procedure in Algorithm 4 performs a test to check if v_{17} is the last vertex in DV$[Y]$ (Line 3 of Algorithm 4). In our example, v_{17} is not the last vertex in DV$[Y]$, therefore the algorithm compute two vertices, referred to as m_1 and m_2, such that $time(m_1) \leq time(v_d^{c_j}) \leq time(m_2)$ (Lines 6 of Algorithm 4). In our case, m_1 and m_2 are calculated as v_8 and v_9, respectively. It then replace the waiting arc (v_8, v_9) in A with two new waiting arcs (v_8, v_{17}) and (v_{17}, v_9). This is followed by rewiring transfer arcs incident at v_{17} using Algorithm 3 called as a sub-routine by Algorithm 2.

In details, the procedure for rewiring transfer arcs in our specific example do not make any changes to G. In more details, the algorithm performs a test to verify if v_{17} is last vertex in DV$[Y]$ (Line 1 of Algorithm 3). In the negative case, which is also the case with our example, it scans the transfer arcs incident at v_9, where v_9 is a successor of v_{17} in DV$[Y]$ (Line 14 of Algorithm 3). Notice that, we have only one such arc (v_7, v_9), incident at v_9, in A. It then verify if this arc can be replaced by (v_7, v_{17}) subjected

to the condition that $time(v_7) + \text{MTT}_Y \leq time(v_{17})$, which cannot be satisfied for our example. Therefore, the basic D-PTL algorithm update L (see Table 5.7) using INC-BU as sub-routine call(Line 17 of Algorithm 2).

Finally, concerning the addition of the arrival vertex in V, the basic D-PTL procedure adds a new vertex to V. Similar to the case with departure vertex v_{17} (added earlier), we assume the new arrival vertex be the last vertex w.r.t the ordering in V, i.e. v_{18}. Once v_{18} is added to V, the algorithm proceeds by restoring the RED-TE properties. In particular, the algorithm adds a new connection arc (v_{17}, v_{18}) in A (Line 18 of Algorithm 2). Bypass and the transfer arcs are then rewired using Algorithm 5 as a sub-routine call by Algorithm 2.

In detail, Lines 1–2 of Algorithm 5 computes a vertex, referred to as MIN_NODE, such that MIN_NODE $\in$ AV$[Z]$ and $time(\text{MIN\_NODE}) \geq time(v_{18}) + \text{MTT}_Z$. In our example, MIN_NODE corresponds to v_{16}. Once MIN_NODE is computed, the algorithm then add a transfer arc (v_{18}, v_{16}), and proceeds by adding bypass arcs with the arrival vertices of the preceeding connections and following connections of c'_k in the same trip, i.e. the one served by vehicle β (Lines 4–8 of Algorithm 5). However, as c'_k is the last connection served by the vehicle with id β, therefore the algorithm adds only a transfer arc (v_1, v_{18}) in A(see Fig. 5.8). Note that v_1 is the arrival vertex associated with the connection preceding c'_k in the same trip.

After updating the graph, Algorithm 2 updates L (shown in Table 5.8) using INC-BU once again called as sub-routine. The resultant RED-TE graph after the execution of the insertion phase is shown in Fig. 5.9.

Vertices	2HC-R Labeling (L)	
	$\text{L}_{in}(v_*)$	$\text{L}_{out}(v_*)$
v_1	$\{v_1\}$	$\{v_1\}$
v_3	$\{v_3\}$	$\{v_1, v_3\}$
v_4	$\{v_4\}$	$\{\ v_4\ \}$
v_5	$\{v_5\}$	$\{v_1, v_3, v_4, v_5\}$
v_6	$\{v_4, v_6\}$	$\{v_6\ \}$
v_7	$\{v_3, v_7\}$	$\{v_7\}$
v_8	$\{\ v_1,\ v_4,\ v_6,\ v_8\ \}$	$\{v_8\}$
v_9	$\{v_1,\ v_4,\ v_7,\ v_8,\ v_9\ \}$	$\{v_9\}$
v_{11}	$\{v_{11}\}$	$\{\ v_1,\ v_3,\ v_4,\ v_5,\ v_{11}\ \}$
v_{12}	$\{v_{12}\}$	$\{v_1, v_3, v_{12}\}$
v_{13}	$\{v_3, v_{13}\}$	$\{v_7, v_{13}\}$
v_{14}	$\{v_1,\ v_4,\ v_6,\ v_8,\ v_{14}\ \}$	$\{v_{14}\}$
v_{15}	$\{\ v_{15}\ \}$	$\{v_{15}\}$
v_{16}	$\{\ v_1,\ v_4,\ v_6,\ v_8,\ v_{14},\ v_{15},\ v_{16}\ \}$	$\{v_{16}\}$
$v_{17}{}^+$	$\{v_1, v_4, v_6, v_8, v_{17}\}^+$	$\{v_9, v_{17}\}^+$

TABLE 5.7: Updated 2HC-R labeling for the RED-TE graph shown in Fig. 5.8 after executing INC-BU(G, v_{17}, L). In details, newly added label sets are those with superscript "$\star$".

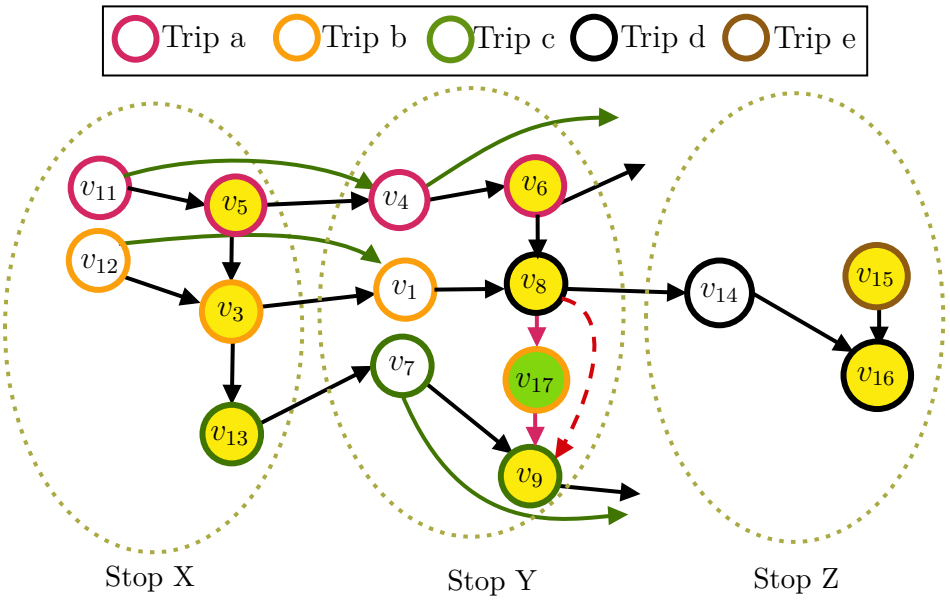

FIGURE 5.8: RED-TE graph after adding the departure vertex having updated time value. The newly added vertex is filled with Green, while the newly added arcs are colored in Magenta. The removed arc, however, is shown as dotted arc.

Vertices	2HC-R Labeling (L)	
	$L_{in}(v_*)$	$L_{out}(v_*)$
v_1	$\{v_1\}$	$\{v_1\}$
v_3	$\{v_3\}$	$\{v_1, v_3\}$
v_4	$\{v_4\}$	$\{ v_4 \}$
v_5	$\{v_5\}$	$\{v_1, v_3, v_4, v_5\}$
v_6	$\{v_4, v_6\}$	$\{v_6 \}$
v_7	$\{v_3, v_7\}$	$\{v_7\}$
v_8	$\{ v_1, v_4, v_6, v_8 \}$	$\{v_8\}$
v_9	$\{v_1, v_4, v_7, v_8, v_9 \}$	$\{v_9\}$
v_{11}	$\{v_{11}\}$	$\{ v_1, v_3, v_4, v_5, v_{11} \}$
v_{12}	$\{v_{12}\}$	$\{v_1, v_3, v_{12}\}$
v_{13}	$\{v_3, v_{13}\}$	$\{v_7, v_{13}\}$
v_{14}	$\{v_1, v_4, v_6, v_8, v_{14} \}$	$\{v_{14}\}$
v_{15}	$\{ v_{15} \}$	$\{v_{15}\}$
v_{16}	$\{ v_1, v_4, v_6, v_8, v_{14}, v_{15}, v_{16} \}$	$\{v_{16}\}$
v_{17}	$\{ v_1, v_4, v_6, v_8, v_{17} \}$	$\{v_9, v_{16}, v_{17}\}^{\star}$
$v_{18}{}^{+}$	$\{v_1, v_4, v_6, v_8, v_{17}, v_{18}\}^{+}$	$\{v_{16}, v_{18}\}^{+}$

TABLE 5.8: Updated 2HC-R labeling for the RED-TE graph shown in Fig. 5.9 after executing INC-BU(G, v_{18}, L). In details, newly added label sets (modified label sets, respectively) are those with superscript "+" ("⋆", respectively).

Updating Stop Labels Given the updated 2HC-R labels L, the D-PTL procedure for dynamically updating stop labels (Algorithm 6) reflect the changes made in L onto SL. In details, specific to the example we have considered, the procedure considers as input the updated L (shown in Table 5.8), SL (shown in Table 5.3), and two sets of stops US_{in} and US_{out}. Concerning US_{in} and US_{out}, as mentioned earlier in Section 5.1.3, both of these sets are maintained throughout the execution of the removal and insertion phases. In more details , a stop S_i is added to US_{in} (US_{out}, respectively) only if there is a vertex

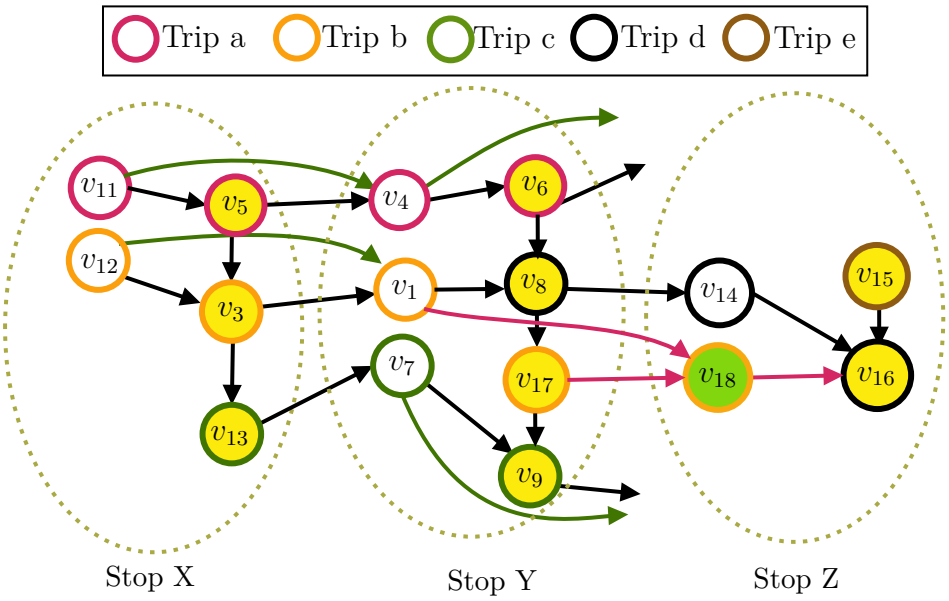

FIGURE 5.9: RED-TE graph after adding the arrival vertex having updated time value. The newly added vertex is filled with Green, while the newly added arcs are colored in Magenta.

$v \in \text{AV}[i]$ such that $\text{L}_{in}(v)$ ($\text{L}_{out}(i)$, respectively) was modified in either of the removal or insertion phase or $time(v)$ was modified.

For our example, we analyze the changes made to L throughout the removal and insertion phases (i.e. whose details are shown in Tables 5.2–5.8), and present a summary of such an analysis in Table 5.9. From the results shown in Table 5.9, it can be easily seen that US_{out} contains only Y and US_{in} contains only Z.

Stops	AV[∗]	DV[∗]
X	$\{v_{11}, v_{12}\}$	$\{v_5, v_3, v_{13}\}$
Y	$\{v_4, v_1, v_7\}$	$\{v_6{}^{\star}, v_2{}^{-}, v_8, v_{17}{}^{+}, v_9\}$
Z	$\{v_{10}{}^{-}, v_{14}{}^{\star}, v_{18}{}^{+}\}$	$\{v_{15}, v_{16}\}$

TABLE 5.9: Listing vertices in AV[i] (DV[i], respectively) for each $\text{S}_i \in \mathcal{S}$ in the second column (third column, respectively). The color of the vertices in second column can be interpreted as follows. A vertex v_j is with superscript: i) "+", if v_j is new vertex added to G in the insertion phase; ii) "$\star$", if $\text{L}_{in}(v_j)$ was modified either in the removal or insertion phase; iii) "$-$", if $\text{L}_{in}(v_j)$ was removed either in the removal or insertion phase. The same hold for the entries in the third column, however, instead of $\text{L}_{in}(*)$, we considered $\text{L}_{out}(*)$.

Given US_{out} and US_{in}, as a first step, the algorithm recompute $\text{SL}_{out}(s)$ for each such $s \in \text{US}_{out}$ (Lines 1–9 of Algorithm 6). In our case, it recomputes $\text{SL}_{out}(Y)$ as Y is the only element in US_{out}. It then order the entries in $\text{SL}_{out}(Y)$ w.r.t the time values associated with each hub. The algorithm considers US_{in}, and computes $\text{SL}_{in}(Z)$ as Z is the only element in US_{in} (Lines 11–19 of Algorithm 6), and accordingly order the entries in $\text{SL}_{in}(Z)$. The updated stop labels obtained at the end of such execution is shown in

Table 5.10. This conclude the basic D-PTL's procedure for updating G, L and SL as a result of the delay δ in c_k.

Stops	Stop Labeling (SL)	
	$\text{SL}_{in}(*)$	$\text{SL}_{out}(*)$
X	$\{(v_{11}, 5), (v_{12}, 7)\}$	$\{(v_5, 10), (v_3, 15), (v_4, 10), (v_1, 15), (v_{13}, 35), (v_7, 35)\}$
Y	$\{(v_3, 42), (v_4, 15), (v_1, 20), (v_7, 42)\}$	$\{(v_6, 20), (v_8, 30), (v_{17}, 35), (v_9, 50), (v_{16}, 35)\}^\star$
Z	$\{(v_4, 39), (v_1, 39), (v_6, 39), (v_8, 39), (v_{17}, 40), (v_{18}, 40)\}^\star$	$\{(v_{15}, 40), (v_{16}, 50)\}$

TABLE 5.10: Updated SL after G and L are modified in the removal and in insertion phases. Recomputed stop labels during the execution of Algorithm 6 are associated a superscript "$\star$".

5.2 Dynamic Multi-Criteria Public Transit Labeling

In this Section, we extend the basic D-PTL algorithm to handle the multi-criteria setting and refer to the extended version as multi-criteria D-PTL (presented in [50]). 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 [105]. 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 [37] and the incremental algorithm INCPLL of [2], 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 work 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 detail, 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 [37] 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 5.2.1. *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 5.2.1. *For any vertex $v \in \text{RIGHT}_x$, a label entry (h, δ_{hv}) in $\text{L}_{in}(v)$ maybe affected only if $h \in \text{LEFT}_x$ or if $v = x$.*

Property 5.2.2. *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 5.2.2. *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 5.5. *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 5.2.1–5.2.2. 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 BFS 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.

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)};
```

5.2.1 Forward Update

The procedure processes vertices in LEFT_x in decreasing order w.r.t a topological ordering T of G. Assume we are processing a given vertex, say v. If v has a maximum value in T as compared to that for the rest of vertices in LEFT_x, then we know by the definition of T that 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 5.2.1). 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 5.2.2). 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 [105], 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 5.5). 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 9).

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 8). 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 [31]. 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 precedes v in the established order (we refer the reader to [31,37] for more details). As discussed in earlier sections, we denote by $l(v)$ the position of a vertex $v \in V$ according to the established order.

Algorithm 8: 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

```
1  foreach v ∈ LEFT_x in decreasing order of t(v) do
2      C_out(v) ← ∅ ;
3      foreach (h, δ_vh) ∈ L_out(v) do
4          if h ∈ RIGHT_x or h = x then
5              L_out(v) ← L_out(v) \ {(h, δ_vh)} ;
6      foreach u ∈ N^G_out(v) do
7          foreach (h, δ_uh) ∈ L_out(u) do
8              if h ∈ RIGHT_x and h ∉ C_out(v) then
9                  C_out(v) ← C_out(v) ∪ {h};
10     A_v ← customBFS(G, v, RIGHT_x, LEFT_x);
11     foreach w ∈ C_out(v) in increasing order of t(w) do
12         d ← min_{(u,δ_vu)∈A_v} {δ_vu + QUERY(u, w, L)} ;
13         if MQUERY(v, w, L) > d then
14             if l(v) > l(w) then
15                 L_out(v) ← L_out(v) ∪ {(w, d)};
16             else
17                 if (v, δ_vw) ∈ L_in(w) then
18                     δ_vw ← d;
19                 else
20                     L_in(w) ← L_in(w) ∪ {(v, d)};
21                 Mark (v, δ_vw) in L_in(h);
```

Algorithm 9: 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
```

5.2.2 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 5.5). However, in this case, there may be marked label entries, e.g. $(h, \delta_{vh}) \in \text{L}_{in}(v)$ 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 5.5). 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 $\text{L}_{in}(v)$, we can compute correct values of $d_{G'}(w, v)$ for any $w \in V'$ such that $w \notin \text{LEFT}_x$, via a query $\text{QUERY}(w, v, \text{L})$. However, to restore the cover property for other vertices, we may need to add new label entries to $\text{L}_{in}(v)$ and possibly to forward label sets of some of the vertices in LEFT_x. To this end, symmetrically to the forward update case, we compute the superset of hubs, this time for the backward label $\text{L}_{in}(v)$, denoted as $\text{C}_{in}(v)$, as the union of the hub vertices, belonging to LEFT_x, for all backward label sets of all vertices in $N^{G'}_{in}(v)$, that

is:

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

Observe that, for any $w \in \textsc{c}_{in}(v)$, 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 $\textsc{query}(w, v, \textsc{l})$. In particular, it is given by the minimum value we obtain for $\textsc{query}(w, u, \textsc{l}) + \textsc{query}(u, v, \textsc{l})$ among all vertices $u \in N_{in}^{G'}(v)$. If this value is not encoded in the labeling, we add (u, δ_{uv}) to $\textsc{l}_{in}(v)$ if $l(v) > l(u)$, otherwise we add (v, δ_{uv}) to $\textsc{l}_{out}(u)$.

Algorithm 10: Procedure BACKWARD.

Input: Directed Acyclic Graph G, 2HC-SP labeling $\textsc{l}$ of G, vertex x to be removed from G, sets $\textsc{left}_x$ and $\textsc{right}_x$.

```
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_in^G(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_in^G(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)};
```

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

Theorem 5.6 (Correctness of DAG-DECPLL)**.** *Let G be a DAG, let $\textsc{l}$ be a 2HC-SP labeling of G, and let x be a vertex of G. Let $G' = G \setminus \{x\}$ and $\textsc{l}'$ be the output of algorithm DAG-DECPLL when applied to G, $\textsc{l}$ and x. Then: a) $G' = G \setminus \{x\}$ is a DAG and b) $\textsc{l}'$ 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, we show, in the following, that: A):

all of the outdated label entries are removed from L, i.e. those label entries that induce paths that include the removed vertex x; and B): the cover property holds for all pairs of vertices in V'.

Concerning A), notice that as a first step in updating L, we compute two sets RIGHT_x and LEFT_x. We then remove the label entries (h, δ_{vh}), with $h \in \text{RIGHT}_x$, from $\text{L}_{out}(v)$ for all $v \in \text{LEFT}_x$. Moreover, we also remove the label entries (h, δ_{hv}), with $h \in \text{LEFT}_x$, from $\text{L}_{in}(v)$ for all $v \in \text{RIGHT}_x$. By doing so, we essentially eliminates all of the induced paths, containing x, in L (see Corollary 5.2.1 and 5.2.2). Moreover, for any vertex $v \in V'$, if $v \in \text{LEFT}_x$ ($v \in \text{RIGHT}_x$, respectively), we know that $\text{L}_{in}(v)$ ($\text{L}_{out}(v)$, respectively) cannot contain affected label entries (see Property 5.2.1 and 5.2.2). Alongside these, it is worthy to mention that, for any vertex v in $V' \setminus (\text{RIGHT}_x \cup \text{LEFT}_x)$, none of the label entries in both $\text{L}_{out}(v)$ and in $\text{L}_{in}(v)$ are affected by the removal of x from G. Furthermore, for any pair of vertices u, v in V', excluding the pairs where both $u \in \text{LEFT}_x$ and $v \in \text{RIGHT}_x$, L after the removal of affected label entries can still be used for computing shortest paths in G' (see Lemma 5.5).

Finally, concerning B), i.e. related to the cover property, we test the property for all and only the vertices that are affected by the removal of x (sets RIGHT_x and 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. Therefore our claim follows. □

Theorem 5.7 (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 w.r.t. $|V|$, 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. □

5.2.3 Extensions

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.

5.2.4 Compact 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 $\text{S}_i \in \mathcal{S}$ 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 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 E-SL$_{out}(i)$;
2. there exists an entry $(h, stoptime_i(h), transfers_i(h))$ in E-SL$_{out}(i)$ but $\delta_{vh} <$ MT where
$$\text{MT} = \min_{(h, stoptime_i(h), transfers_i(h)) \in \text{E-SL}_{out}(i)} transfers_i(h).$$

To compute E-SL$_{in}(i)$, symmetrically, we process vertices in 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 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 [41] for the stop labeling. The detailed procedure for computing the extended stop labeling is shown in Algorithm 11.

5.2.4.1 Multi-criteria Query Algorithm

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 $stoptime_i(v) \geq \tau$.

Algorithm 11: Algorithm E-SL Computation.

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

```
1  foreach S_i ∈ 𝒮 do
2      E-SL_in(i) ← ∅;
3      E-SL_out(i) ← ∅;
4      foreach v ∈ V do
5          Visit[v] = 0;
6          MT[v] = ∞;
7      foreach v ∈ AV[i] in increasing order of arrival time do
8          foreach (h, δ_hv) ∈ L_in(v) do
9              if Visit[v] = 0 then
10                 E-SL_in(i) ← E-SL_in(i) ∪ {(h, time(v), δ_hv)};
11                 MT[v] ← δ_hv;
12                 Visit[v] ← 1;
13             else if MT[v] > δ_hv then
14                 E-SL_in(i) ← E-SL_in(i) ∪ {(h, time(v), δ_hv)};
15                 MT[v] ← δ_hv;
16     Sort E-SL_in(i) w.r.t. first, second and third field in each label entry;
17     foreach v ∈ V do
18         Visit[v] = 0;
19         MT[v] = ∞;
20     foreach v ∈ DV[i] in decreasing order of departure time do
21         foreach (h, δ_vh) ∈ L_out(v) do
22             if Visit[v] = 0 then
23                 E-SL_out(i) ← E-SL_out(i) ∪ {(h, time(v), δ_hv)};
24                 MT[v] ← δ_vh;
25                 Visit[v] ← 1;
26             else if MT[v] > δ_vh then
27                 E-SL_out(i) ← E-SL_out(i) ∪ {(h, time(v), δ_vh)};
28                 MT[v] ← δ_hv;
29     Sort E-SL_in(i) w.r.t. first, second and third field in each label entry;
30 return E-SL;
```

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 w.r.t 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). We are now ready to provide to state the following

Lemma.

Theorem 5.8. *(Correctness of Multi-criteria query answering procedure using* E-SL*) Given a* WRED-TE *graph* $G = (V, A, w)$ *of a timetable* $\mathcal{T} = (\mathcal{Z}, \mathcal{S}, \mathcal{C})$*, and a* 2HC-SP *labeling* L *of* G*. If* E-SL *is the compact form of* L *computed using the procedure shown in Algorithm 11, then an answer to a generic multi-criteria query* MC-EA$(\mathrm{S}_i, \mathrm{S}_j, \tau)$ *computed using* L *via the multi-criteria* PTL *can be also computed using* E-SL*.*

Proof. For showing that the above is correct, we need to show that, one can correctly answers a multi-criteria query using E-SL in all possible scenarios. In details, when computing an answer to a multi-criteria query MC-EA$(\mathrm{S}_i, \mathrm{S}_j, \tau)$, one of the following two cases can occur: A) none of the journeys between S_s and S_t has a departure time at S_s later than or equal to τ; and B) there exist at least one journey J between S_s and S_t having a departure time at S_s no earlier than τ. Below, we discuss the correctness of the above statement in both of these cases in detail.

Discussion on Case A. In this case, occurring when no journey exist between S_s and S_t with the departure time at S_s later than or equal to τ, clearly for each pair (v, u) of vertices with $v \in \mathrm{DV}[s]$ and $u \in \mathrm{AV}[t]$ such that $time(v) \geq \tau$, we have that $\mathrm{L}_{out}(v) \cap \mathrm{L}_{in}(u) = \emptyset$ (see Definition 3.2). We prove that, in this case, the procedure for computing MC-EA$(\mathrm{S}_s, \mathrm{S}_t, \tau)$ using E-SL always returns an empty set. In particular, using proof through contradiction, we show that if we consider only those label entries $(h, time(v), \delta_{vh})$ in E-SL$_{out}(s)$ for which $time(v) \geq \tau$, then E-SL$_{out}(s) \cap$ E-SL$_{in}(t) = \emptyset$. To this end, we assume that the procedure for answering MC-EA$(\mathrm{S}_s, \mathrm{S}_t, \tau)$ using E-SL returns at least one journey, say J. This in-turn means that the following must be true: i) there exists a vertex $v \in \mathrm{DV}[s]$ with $time(v) \geq \tau$ for which $(h, \delta_{vh}, time(v)) \in$ E-SL$_{out}(s)$, that is $(h, \delta_{vh}) \in \mathrm{L}_{out}(v)$; and ii) there exists a vertex $u \in \mathrm{AV}[t]$ for which $(h, \delta_{hu}, time(u))$ in E-SL$_{in}(t)$, that is $(h, \delta_{hu}) \in \mathrm{L}_{in}(u)$. Notice that, for i), we have that $time(h) \geq time(v) \geq \tau$ (by definition of E-SL). For ii), instead, we have that $\tau \leq time(h) \leq time(u)$ (once again by definition of E-SL). The presence of these label entries in E-SL$_{out}(s)$ (E-SL$_{in}(t)$, respectively) essentially means that there exists a path between v and u in G with $v \in \mathrm{DV}[s]$ and $u \in \mathrm{AV}[t]$. This negates the fact that $\mathrm{L}_{out}(v) \cap \mathrm{L}_{in}(u) = \emptyset$ for all vertices $v \in \mathrm{DV}[s]$ with $time(v) \geq \tau$ and for all vertices $u \in \mathrm{AV}[s]$. Hence, it contradicts our assumption, and therefore our claim follows.

Discussion on Case B. In this case, occurring when there exists at least one journey J between S_s and S_t with departure time at S_s no earlier than τ, clearly $\mathrm{L}_{out}(v) \cap \mathrm{L}_{in}(u) \neq \emptyset$ for a vertex $v \in \mathrm{DV}[s]$ with $time(v) \geq \tau$ and a vertex $u \in \mathrm{AV}[t]$ (see Definition 3.2 and that of 2HC-SP labeling). For showing the correctness of the query answering procedure operating on E-SL, we consider at first those cases where J is unique. To proceed with

our discussion, assume h be the hub vertex between the vertices v and u, then when computing E-SL, h must be inserted in one of the label entries in E-SL$_{out}(s)$ and also in E-SL$_{in}(t)$ (Lines 7–15 and Lines 20–28 of Algorithm 11). Hence, the query answering procedure must consider label entries with h. For scenarios where J is not unique, that is there exist multiple journeys between S$_s$ and S$_t$ with departure time at S$_s \geq \tau$, it can be easily seen that the same reasoning also hold as we allow multiple label entries having the same hub entries with decreasing distances (Lines 13–15 and Lines 26–28 of Algorithms 11).

By following our discussion for both of the above cases A) and B), it can be concluded that the procedure we proposed for answering multi-criteria query using E-SL results in correctly answering any generic multi-criteria query. □

5.2.4.2 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 7–10, 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$_i \in \mathcal{S}$ 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 5.9 (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. Concerning i), using the proof we provided for Lemma 5.1 and by replacing the RED-TE graph with WRED-TE graph, it can be easily shown that 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 5.6 or the proof in [105]). Hence, at the end of Algorithm 2, L is a 2HC-SP labeling for G. Finally, note that the algorithm described in section 5.2.4.2 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 5.2.4.2, E-SL is still an extended stop labeling of L. □

Theorem 5.10 (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 5.4. 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 $|\mathcal{S}| \leq |\mathcal{C}|$. Since $|V| \in \mathcal{O}(|\mathcal{C}|)$ for any WRED-TE graph, the claim follows. □

5.2.5 Example

In this section, we describe the working of the multi-criteria D-PTL using an example scenario. Moreover, for the sake of developing a better understanding of the strategy DAG-DECPLL uses, we describe in detail how the multi-criteria D-PTL update L using DAG-DECPLL after a decremental update is introduced in G. Alongside this, we describe the procedure we proposed for computing E-SL and that for dynamically updating it. For the sake of presentation, here we use D-PTL when referring to the multi-criteria D-PTL. Nonetheless, we differentiate the basic D-PTL from the multi-criteria D-PTL wherever required.

Concerning the description of our example scenario, we assume as input the timetable presented in Table 3.1, whose corresponding WRED-TE graph is shown in Fig. 3.2. Moreover, we need to have at our disposal a description of the following parameters:

- a 2HC-SP labeling L of G;
- an extended stop labels E-SL computed on top of L; and
- a delay δ introduced in a connection $c_i \in \mathcal{C}$.

In what follows, we describe each of the above mentioned parameters in detail.

2HC-SP Labeling Concerning the 2HC-SP labeling L, as discussed earlier in Section 2.2.6.2, one can easily compute it using e.g. PLL given an ordered set V of vertices. In our case, we assume that the vertices in V are sorted using degree in descending order (shown in Table 5.1 and in Fig. 5.10). It is worthy to remark that, we assume the vertices in V are given as $V = \{v_1, v_2, \ldots, v_{16}\}$ such that a vetex v_i appears before a vertex v_j only if $i < j$, in which case, we say that $l(v_j) > l(v_i)$. In our case, we assume L is computed using PLL (see Table 5.11) given the WRED-TE graph $G(V, A, w)$ with V as an ordered set of vertices.

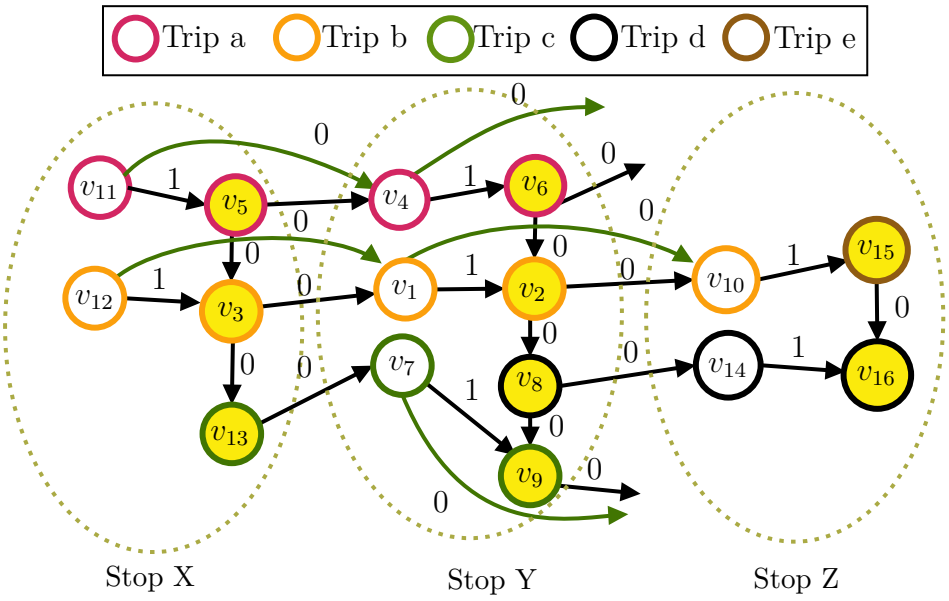

FIGURE 5.10: An example of weighted RED-TE graph for a timetable with three stops X, Y and Z, five trips $\{a, b, c, d, e\}$ reported in Table 3.1.

Extended Stop Labels Given the 2HC-SP labeling L of G, we compute E-SL using the procedure shown in Algorithm 11. In particular, the Algorithm considers as input the WRED-TE graph G (shown in Fig. 5.10), and a 2HC-SP labeling L of G (shown in Table 5.11), and compute E-SL (see Table 5.12) as follows. For each $i \in \mathcal{S}$ with $\mathcal{S} = \{X, Y, Z\}$ in our case, the algorithm computes E-SL$_{in}(i)$ by scanning the in-label sets of vertices in AV$[i]$, while possibly adding new hubs in E-SL$_{in}(i)$. In details, the algorithm scans each vertex v in AV$[i]$, and update E-SL$_{in}(i)$ by adding a new label entry. In particular, for each such $(h, \delta_{hv}) \in$ L$_{in}(v)$ scanned by the algorithm, it adds $(v_h, \delta_{hv}, time(v))$ to E-SL$_{in}(i)$ subject to existing label entries in E-SL$_{in}(i)$ (Lines 7–15 of Algorithm 11). For computing E-SL$_{out}(i)$ for all such $i \in \mathcal{S}$, instead, the algorithm considers each vertex

Vertex	2HC-SP Labeling	
	$\text{L}_{in}(v_*)$	$\text{L}_{out}(v_*)$
v_1	$\{(v_1,0)\}$	$\{(v_1,0)\}$
v_2	$\{(v_1,1),(v_2,0)\}$	$\{(v_2,0)\}$
v_3	$\{(v_3,0)\}$	$\{(v_1,0),(v_3,0)\}$
v_4	$\{(v_4,0)\}$	$\{(v_2,1),(v_4,0)\}$
v_5	$\{(v_5,0)\}$	$\{(v_1,0),(v_3,0),(v_4,0),(v_5,0)\}$
v_6	$\{(v_4,1),(v_6,0)\}$	$\{(v_2,0),(v_6,0)\}$
v_7	$\{(v_3,0),(v_7,0)\}$	$\{(v_7,0)\}$
v_8	$\{(v_1,1),(v_2,0),(v_8,0)\}$	$\{(v_8,0)\}$
v_9	$\{(v_1,1),(v_2,0),(v_7,1),(v_8,0),(v_9,0)\}$	$\{(v_9,0)\}$
v_{10}	$\{(v_1,0),(v_2,0),(v_{10},0)\}$	$\{(v_{10},0)\}$
v_{11}	$\{(v_{11},0)\}$	$\{(v_1,1),(v_2,1),(v_3,1),(v_4,0),(v_5,1),(v_{11},0)\}$
v_{12}	$\{(v_{12},0)\}$	$\{(v_1,0),(v_3,1),(v_{12},0)\}$
v_{13}	$\{(v_3,0),(v_{13},0)\}$	$\{(v_7,0),(v_{13},0)\}$
v_{14}	$\{(v_1,1),(v_2,0),(v_8,0),(v_{14},0)\}$	$\{(v_{14},0)\}$
v_{15}	$\{(v_1,1),(v_2,1),(v_{10},1),(v_{15},0)\}$	$\{(v_{15},0)\}$
v_{16}	$\{(v_1,1),(v_2,1),(v_8,1),(v_{10},1),(v_{14},1),(v_{15},0),(v_{16},0)\}$	$\{(v_{16},0)\}$

TABLE 5.11: 2HC-SP labeling for the RED-TE graph shown in Fig. 5.10 by following the vertex ordering presented in Table 5.1.

v in DV$[i]$, and add a label $(v_h, \delta_{vh}, time(v))$ to E-SL$_{out}(i)$ if $(h, \delta_{vh}) \in \text{L}_{out}(v)$ (Lines 20–28 of Algorithm 11). This is followed by ordering the entries in E-SL$_{out}(i)$ and E-SL$_{in}(i)$, for all $i \in \mathcal{S}$, w.r.t the time of hubs.

Delay Concerning the delay, we assume a delay δ of ten minutes added to the departure time of the last connection in TRIP served by vehicle with id β. More precisely, we assume the connection $c_k = (\beta, \text{S}_Y, \text{S}_Z, 25, 30)$ be now given as $c'_k = (\beta, \text{S}_Y, \text{S}_Z, 35, 40)$. Notice that this delay is similar to that we considered earlier in our example for the basic D-PTL case. We do so in order to focus on the description of how the multi-criteria D-PTL update L, i.e. DECPLL algorithm, instead of how G is rewired which we explained earlier in Section 5.1.4.

Stops	Extended stop labeling (E-SL)	
	E-SL$_{in}(*)$	E-SL$_{out}(*)$
X	$\{(v_{11},5,0),(v_{12},7,0)\}$	$\{(v_5,10,0),(v_3,15,0),(v_4,10,0),(v_1,15,0),(v_{13},35,0),(v_7,35,0)\}$
Y	$\{(v_4,15,0),(v_3,42,0),(v_1,20,0),(v_7,42,0)\}$	$\{(v_6,0,20),(v_2,0,25),(v_8,0,30),(v_9,0,50)\}$
Z	$\{(v_1,30,0),(v_2,30,0),(v_8,39,0),(v_{10},30,0),(v_{14},39,0)\}$	$\{(v_{15},40,0),(v_{16},50,0)\}$

TABLE 5.12: Extended Stop labeling for the WRED-TE graph shown in Fig. 5.10 built on-top of the 2HC-SP labeling presented in Table 5.11

5.2.5.1 Applying Multi-criteria DPTL

Given the WRED-TE graph G, the 2HC-SP labeling L of G, and the connection c_i with δ as the amount of incurred delay, here we describe how D-PTL update G and L. It is worthy to remark that, for updating G and L, the multi-criteria D-PTL algorithm uses the same removal and deletion phases as used by the basic D-PTL (see Algorithm 1 and in Algorithm 2, respectively). However, for updating L after decremental and incremental updates to G, as explained in earlier sections, the multi-criteria D-PTL uses DAG-DECPLL and INCPLL Algorithms, respectively.

We proceed by discussing at first the removal phase, followed by a discussion about the insertion phase.

Removal Phase Given the delayed connection c_i with v_2 as the departure vertex and v_{10} as the arrival vertex, the algorithm proceeds by considering at first v_2, and add δ to $time(v_2)$. For checking if v_2 is affected by the delay, it compares $time(v_2)$ with $time(v_8)$ (Lines 9-12 of Algorithm 1). In our case, $time(v_2) > time(v_8)$, and therefore the algorithm computes PRED and SUCC vertices of v_2. It also compute $\overline{A}$ in exactly the same way we described in the basic D-PTL case. The algorithm then removes v_2 from G (shown in Fig. 5.11) and update L using DAG-DECPLL (see Algorithm 7) as follows.

As a first step, the DAG-DECPLL procedure computes the topological order T of G (Line 1 of Algorithm 7), and is given as follows:

$$T = \{v_{11}, v_5, v_4, v_6, v_{12}, v_3, v_1, v_2, v_8, v_{10}, v_{14}, v_{15}, v_{16}, v_{13}, v_7, v_9\}$$

Once T is computed, the algorithm then compute LEFT_{v_2} and RIGHT_{v_2} via backward and forward BFS rooted at v_2 (Line 3 of Algorithm 7). In our case, LEFT_{v_2} and RIGHT_{v_2} are as follows:

$$\text{LEFT}_{v_2} = \{v_{11}, v_5, v_4, v_6, v_{12}, v_3, v_1\},$$
$$\text{RIGHT}_{v_2} = \{v_8, v_9, v_{10}, v_{14}, v_{15}, v_{16}\}$$

After computing LEFT_{v_2} and RIGHT_{v_2}, the algorithm performs a check to verify if either of the LEFT_{v_2} or RIGHT_{v_2} is an empty set (Line 5 of Algorithm 7). However, in our case, both LEFT_{v_2} and RIGHT_{v_2} are not empty sets. Therefore, the algorithm proceeds towards updating L via the forward phase (see Algorithm 8) followed by the backward phase (see Algorithm 10) as follows.

The procedure described in Algorithm 8 process the vertices in LEFT_{v_2} in decreasing topological order T (Line 1 of Algorithm 8). In our case, the algorithm proceeds by picking at first v_1, and remove all affected label entries from $\text{L}_{out}(v_1)$ (Lines 3–5 of Algorithm 8). Recall that, a label entry $(h, \delta_{v_1 h}) \in \text{L}_{out}(v_1)$, given v_2 is removed from G, is affected if $h \in \text{RIGHT}_{v_2} \cup \{v_2\}$. Observe that $\text{L}_{out}(v_1)$ contains $(v_1, 0)$ only (see Table 5.11) and that $v_1 \notin \text{RIGHT}_{v_2} \cup \{v_2\}$. This essentially means that none of the label entries in $\text{L}_{out}(v_1)$ are affected by the removal of v_2 from G.

The algorithm proceeds by computing $\text{C}_{out}(v_1)$, that is a super-set of labels for $\text{L}_{out}(v_1)$, by scanning the label entries in $\text{L}_{out}(u)$ for all $u \in N^G_{out}(v_1)$. In particular, for each (h, δ_{uh}) the algorithm scans, it add h to $\text{C}_{out}(v_1)$ only if $h \notin \text{LEFT}_{v_2} \cup \{v_2\}$ (Lines 6–9 of Algorithm 8). In our case, $N^G_{out}(v_1)$, after we have removed v_2 from G (shown in Fig. 5.11), is composed of v_{10} only. Therefore, the algorithm scans label entries in $\text{L}_{out}(v_{10})$ (shown in Table 5.11), and eventually add v_{10} to $\text{C}_{out}(v_1)$.

Once $\text{C}_{out}(v_1)$ is computed, the algorithm then compute another set A_{v_1} using Algorithm 9 called as a sub-routine (Line 10 of Algorithm 8). In particular, Algorithm 9 performs a pruned forward BFS rooted at v_1, and for each vertex h it visits during the BFS, the algorithm add $(h, \delta_{v_1 h})$ to A_v if $h \in \text{RIGHT}_{v_1}$. We recall the reader that the BFS is pruned whenever such a vertex h is scanned. In our case, the algorithm starts the BFS rooted at v_1 and visits v_{10}. As $v_{10} \in \text{RIGHT}_{v_2}$, therefore the algorithm adds $(v_{10}, 0)$ to A_{v_1} with 0 as the shortest distance between v_1 and v_{10} in G (see Fig. 5.11). It then prune the BFS at v_{10}, and as no other vertices are visited by the BFS, the algorithm returns A_{v_1}.

Given A_{v_1}, the procedure in Algorithm 8 proceeds by considering the vertices in $\text{C}_{out}(v_1)$ in increasing order w.r.t the ordering shown in Table 5.12, and accordingly update L (Lines 11–21 of Algorithm 8), as follows.

Given that $\text{C}_{out}(v_1)$ is composed of v_{10} only, the algorithm picks v_{10} and compute the $\delta_{v_1 v_{10}}$ using A_{v_1} and L. In particular, the algorithm computes $\delta_{v_1 v_{10}}$ as $\delta_{v_1 v_{10}} + \text{QUERY}(v_{10}, v_{10}, \text{L})$ (Line 12 of Algorithm 8). Notice that $\delta_{v_1 v_{10}}$ is contained in A_{v_1}. It can be easily seen that $\delta_{v_1 v_{10}}$ is assigned a value of 0. The algorithm then performs a check to verify that $\text{MQUERY}(v_1, v_{10}, \text{L})$ results in a value greater than that of $\delta_{v_1 v_{10}}$ (Line 17 of Algorithm 8). In our case, $\text{L}_{out}(1) \cap \text{L}_{in}(10)$ contains v_1 as the common hub vertex (see Table 5.11). However, as $v_1 \in \text{LEFT}_{v_2}$ and the fact that $\text{MQUERY}(v_1, v_{10}, \text{L})$ do not consider such hubs, therefore $\text{MQUERY}(v_1, v_{10}, \text{L})$ simply returns ∞. The algorithm proceeds by checking if a label entry $(v_1, 0)$ already exists in $\text{L}_{in}(v_{10})$, that is the case with our example, and therefore it updates the distance information in $\text{L}_{in}(v_{10})$ (Line 21, Algorithm 8). Finally, for future iterations, the algorithm marks $(v_1, 0)$ in $\text{L}_{in}(v_{10})$.

For the sake of completion, the algorithm repeats the above-mentioned steps for all of the vertices in $\textsc{left}_{v_2}$. For the sake of simplicity in presentation, we do not explicitly describe the above process for all of the vertices in $\textsc{left}_{v_2}$. Nonetheless, we report the resultant labeling obtained at the end of the forward phase in Table 5.14.

Vertex	2HC-SP Labeling	
	$\textsc{L}_{in}(v_*)$	$\textsc{L}_{out}(v_*)$
v_1	$\{(v_1,0)\}$	$\{(v_1,0)\}$
v_2	$\{(v_1,1),(v_2,0)\}$	$\{(v_2,0)\}$
v_3	$\{(v_3,0)\}$	$\{(v_1,0),(v_3,0)\}$
v_4	$\{(v_4,0)\}$	$\{(v_4,0)\}^\star$
v_5	$\{(v_5,0)\}$	$\{(v_1,0),(v_3,0),(v_4,0),(v_5,0)\}$
v_6	$\{(v_4,1),(v_6,0)\}$	$\{(v_6,0)\}^\star$
v_7	$\{(v_3,0),(v_7,0)\}$	$\{(v_7,0)\}$
v_8	$\{(v_1,1),(v_2,0),(v_8,0)\}$	$\{(v_8,0)\}$
v_9	$\{(v_1,1),(v_2,0),(v_7,1),(v_8,0),(v_9,0)\}$	$\{(v_9,0)\}$
v_{10}	$\{(v_1,0)^\diamond,(v_2,0),(v_{10},0)\}$	$\{(v_{10},0)\}$
v_{11}	$\{(v_{11},0)\}$	$\{(v_1,1),(v_3,1),(v_4,0),(v_5,1),(v_{11},0)\}^\star$
v_{12}	$\{(v_{12},0)\}$	$\{(v_1,0),(v_3,1),(v_{12},0)\}$
v_{13}	$\{(v_3,0),(v_{13},0)\}$	$\{(v_7,0),(v_{13},0)\}$
v_{14}	$\{(v_1,1),(v_2,0),(v_8,0),(v_{14},0)\}$	$\{(v_{14},0)\}$
v_{15}	$\{(v_1,1),(v_2,1),(v_{10},1),(v_{15},0)\}$	$\{(v_{15},0)\}$
v_{16}	$\{(v_1,1),(v_2,1),(v_8,1),(v_{10},1),(v_{14},1),(v_{15},0),(v_{16},0)\}$	$\{(v_{16},0)\}$

TABLE 5.13: Update 2HC-SP labeling after executing the Forward–Update phase, assuming that v_2 is removed from the WRED-TE graph G. Modified label sets are those with superscript "$\star$".

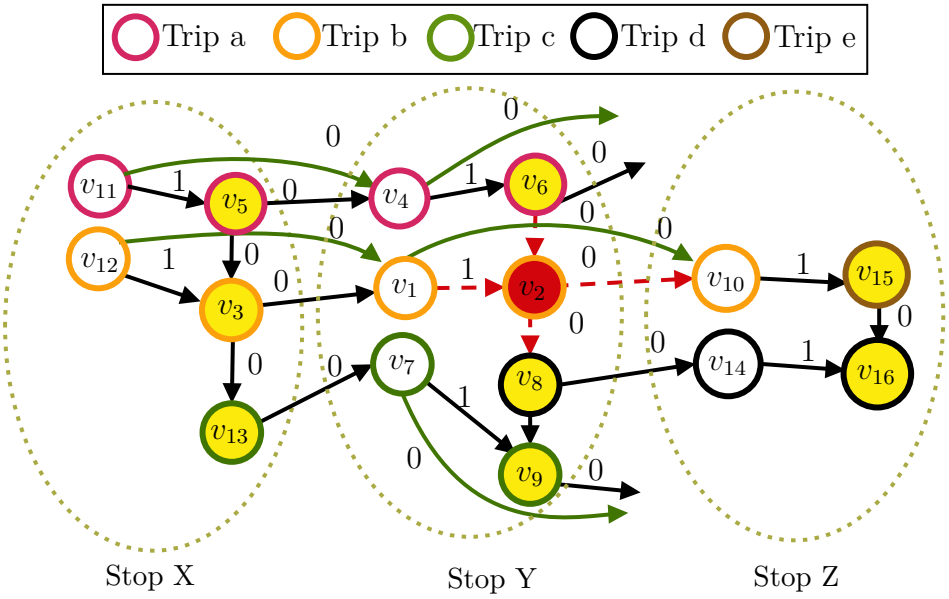

FIGURE 5.11: The updated WRED-TE graph G after removing v_2. In details, vertices and arcs (dotted) in red are those removed from G.

The DAG-DECPLL's backward update phase (see Algortihm 10) is the final phase in the process of updating L after the removal of a vertex from the graph. As an application

to our example scenario, below we describe the working of the backward update phase in details.

Given the 2HC-SP labeling L (shown in Table 5.14), the algorithm proceeds by considering the vertices in RIGHT_{v_2} in increasing order w.r.t T, and update L as follows. It considers at first v_8 and update $\text{L}_{in}(v_8)$ by removing the affected label entries (Lines 3– 6 of Algortihm 10). Observe that the label entries in $\text{L}_{in}(v_8)$ (see Table 5.14) with v_1 and v_2 as hubs are affected, that is $v_1 \in \text{LEFT}_x$ and v_2 is the removed vertex itself. The algorithm then computes $\text{C}_{in}(v_8)$ using the in-label sets of the vertices in $\text{L}_{in}(v_8)$ (Lines 7–10 of Algorithm 10). Notice that $N^G_{in}(v_8)$ in the updated graph (shown in Fig. 5.11) is an empty set, as a result of which, the algorithm do not add any vertex to $\text{C}_{in}(v_8)$. This completes the process of updating $\text{L}_{in}(v_8)$.

The algorithm then considers v_9, and proceeds by removing the affected label entries from $\text{L}_{in}(v_9)$. In particular, the algorithm remove label entries containing v_1 and v_2 in $\text{L}_{in}(v_9)$. It then compute $\text{C}_{in}(v_9)$ by scanning the label entries in $\text{L}_{in}(u)$ for all such $u \in N^G_{in}(v_9)$. In our case, $N^G_{in}(v_9)$ is given as $\{v_7, v_8\}$ (see Fig.5.11). Notice that in $N^G_{in}(v_9)$, we have that $v_7 \notin \text{LEFT}_{v_2} \cup \text{RIGHT}_{v_2}$ and $v_8 \in \text{RIGHT}_{v_2}$. Given $N^G_{in}(v_9)$, the algorithm scans $\text{L}_{in}(v_7)$, and add v_3 in $\text{C}_{in}(v_9)$. It then scan $\text{L}_{in}(v_8)$, however, no new vertex is added in $\text{C}_{in}(v_9)$.

Given $\text{C}_{in}(v_9)$, the algorithm considers the vertices in $\text{C}_{in}(v_9)$, and for each vertex $v \in \text{C}_{in}(v_9)$, it compute δ_{vv_9}. As in our case, v_3 is the only vertex in $\text{C}_{in}(v_9)$, therefore the algorithm computes $\delta_{v_3v_9}$ as the minimum value between: i) $\text{QUERY}(v_3, v_7, \text{L}) + \text{QUERY}(v_7, v_9, \text{L})$; and ii) $\text{QUERY}(v_3, v_8, \text{L}) + \text{QUERY}(v_8, v_9, \text{L})$. In details, i) evaluates into 1 while ii) evaluates into ∞ as $\text{QUERY}(v_3, v_8, \text{L})$ simply returns ∞. Following these, the algorithm add $(v_3, 1)$ in $\text{L}_{in}(v_9)$.

For the sake of completion, the algorithm repeats the above steps for the rest of vertices in RIGHT_{v_2}, and accordingly update L. In Table 5.14, we report the resultant labeling obtained after executing the backward update phase.

Once we have executed the forward and backward update phases for updating L, the DAG-DECPLL's procedure then remove $\text{L}_{in}(v_2)$ and $\text{L}_{out}(v_2)$ from L (Line 10 of Algorithm 7). This conclude the process of dynamically updating L after we have removed v_2 from G.

For restoring the WRED-TE properties, the D-PTL's procedure in Algorithm 1 adds a transfer arc (v_1, v_8) with a weight of value equal to 1 in A, and update L using INCPLL originally presented in [2]. In our case, the INCPLL procedure performs a pruned BFS from the hubs in label sets associated with v_1 and v_8, and update L (see Table 5.15)

Vertex	2HC-SP Labeling	
	$L_{in}(v_*)$	$L_{out}(v_*)$
v_1	$\{(v_1,0)\}$	$\{(v_1,0)\}$
v_2	$\{(v_1,1),(v_2,0)\}$	$\{(v_2,0)\}$
v_3	$\{(v_3,0)\}$	$\{(v_1,0),(v_3,0)\}$
v_4	$\{(v_4,0)\}$	$\{(v_4,0)\}$
v_5	$\{(v_5,0)\}$	$\{(v_1,0),(v_3,0),(v_4,0),(v_5,0)\}$
v_6	$\{(v_4,1),(v_6,0)\}$	$\{(v_6,0)\}$
v_7	$\{(v_3,0),(v_7,0)\}$	$\{(v_7,0)\}$
v_8	$\{(v_8,0)\}^\star$	$\{(v_8,0)\}$
v_9	$\{(v_3,1),(v_7,1),(v_8,0),(v_9,0)\}^\star$	$\{(v_9,0)\}$
v_{10}	$\{(v_1,0)^\diamond,(v_{10},0)\}^\star$	$\{(v_{10},0)\}$
v_{11}	$\{(v_{11},0)\}$	$\{(v_1,1),(v_3,1),(v_4,0),(v_5,1),(v_{11},0)\}$
v_{12}	$\{(v_{12},0)\}$	$\{(v_1,0),(v_3,1),(v_{12},0)\}$
v_{13}	$\{(v_3,0),(v_{13},0)\}$	$\{(v_7,0),(v_{13},0)\}$
v_{14}	$\{(v_8,0),(v_{14},0)\}^\star$	$\{(v_{14},0)\}$
v_{15}	$\{(v_1,1),(v_{10},1),(v_{15},0)\}^\star$	$\{(v_{15},0)\}$
v_{16}	$\{(v_1,1),(v_8,1),(v_{10},1),(v_{14},1),(v_{15},0),(v_{16},0)\}^\star$	$\{(v_{16},0)\}$

TABLE 5.14: Update 2HC-SP labeling after executing the backward Update phase, assuming that v_2 is removed from the WRED-TE graph G. In details, newly added label sets (marked labels, respectively) are those with superscript "+" ("$\diamond$", respectively).

by adding the missing label entries. For details about how INCPLL works, we refer the reader to [2].

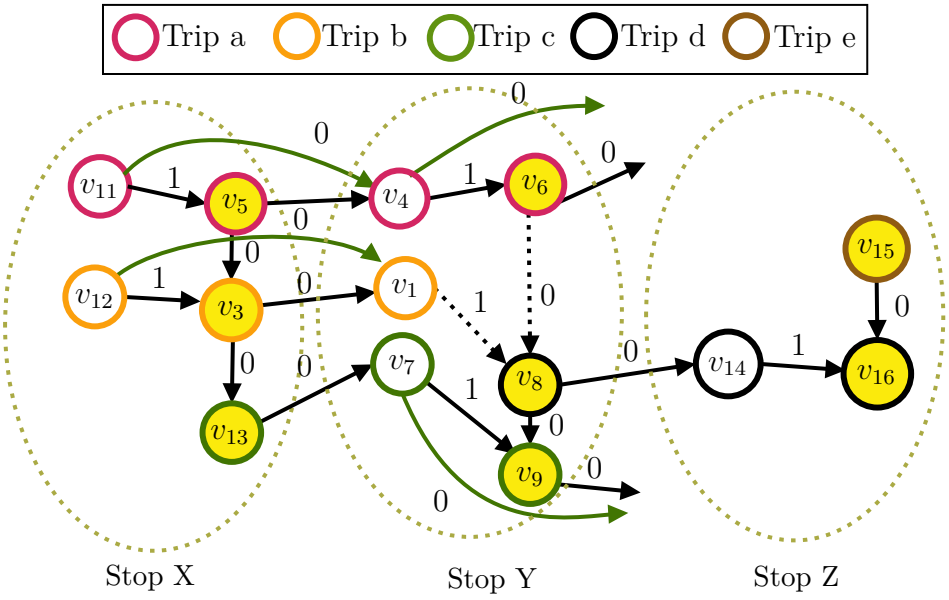

FIGURE 5.12: Updated WRED-TE graph after rewiring arcs. In particular, newly added arcs are those with dotted representation.

Finally, the arrival vertex of the delayed connection, that in our case is given is v_{10}, is removed from G (shown in Fig. 5.12). The algorithm then update L (see Table 5.16) via a sub-procedure call to DAG-DECPLL. This concludes the removal phase.

Vertex	2HC-SP Labeling	
	$\mathrm{L}_{in}(v_*)$	$\mathrm{L}_{out}(v_*)$
v_1	$\{(v_1,0)\}$	$\{(v_1,0)\}$
v_3	$\{(v_3,0)\}$	$\{(v_1,0),(v_3,0)\}$
v_4	$\{(v_4,0)\}$	$\{(v_4,0)\}$
v_5	$\{(v_5,0)\}$	$\{(v_1,0),(v_3,0),(v_4,0),(v_5,0)\}$
v_6	$\{(v_4,1),(v_6,0)\}$	$\{(v_6,0)\}$
v_7	$\{(v_3,0),(v_7,0)\}$	$\{(v_7,0)\}$
v_8	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)\}^\star$	$\{(v_8,0)\}$
v_9	$\{(v_1,1),(v_4,1),(v_6,0),(v_3,1)$ $(v_7,1),(v_8,0),(v_9,0)\}^\star$	$\{(v_9,0)\}$
v_{10}	$\{\ (v_1,0),\ (v_{10},0)\ \}$	$\{(v_{10},0)\}$
v_{11}	$\{(v_{11},0)\}$	$\{(v_1,1),(v_3,1),(v_4,0),(v_5,1),$ $(v_{11},0)\}$
v_{12}	$\{(v_{12},0)\}$	$\{(v_1,0),(v_3,1),(v_{12},0)\}$
v_{13}	$\{(v_3,0),(v_{13},0)\}$	$\{(v_7,0),(v_{13},0)\}$
v_{14}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)$ $(v_{14},0)\}^\star$	$\{(v_{14},0)\}$
v_{15}	$\{(v_1,1),(v_{10},1),(v_{15},0)\}$	$\{(v_{15},0)\}$
v_{16}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,1)$ $(v_{10},1),(v_{14},1),(v_{15},0),(v_{16},0)\}^\star$	$\{(v_{16},0)\}$

TABLE 5.15: Updated 2HC-SP labeling after executing INCPLL procedure as a result of the addition of new arcs: i): (v_1, v_8) with a weight of value 1; and ii): (v_6, v_8) with a weight of value 0; in A. Newly added label sets (modified label sets, respectively) are those with superscript "+" ("$\star$", respectively).

Vertex	2HC-SP Labeling	
	$\mathrm{L}_{in}(v_*)$	$\mathrm{L}_{out}(v_*)$
v_1	$\{(v_1,0)\}$	$\{(v_1,0)\}$
v_3	$\{(v_3,0)\}$	$\{(v_1,0),(v_3,0)\}$
v_4	$\{(v_4,0)\}$	$\{(v_4,0)\}$
v_5	$\{(v_5,0)\}$	$\{(v_1,0),(v_3,0),(v_4,0),(v_5,0)\}$
v_6	$\{(v_4,1),(v_6,0)\}$	$\{(v_6,0)\}$
v_7	$\{(v_3,0),(v_7,0)\}$	$\{(v_7,0)\}$
v_8	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)\}$	$\{(v_8,0)\}$
v_9	$\{(v_1,1),(v_4,1),(v_6,0),(v_3,1)$ $(v_7,1),(v_8,0),(v_9,0)\}$	$\{(v_9,0)\}$
v_{11}	$\{(v_{11},0)\}$	$\{(v_1,1),(v_3,1),(v_4,0),(v_5,1),$ $(v_{11},0)\}$
v_{12}	$\{(v_{12},0)\}$	$\{(v_1,0),(v_3,1),(v_{12},0)\}$
v_{13}	$\{(v_3,0),(v_{13},0)\}$	$\{(v_7,0),(v_{13},0)\}$
v_{14}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)$ $(v_{14},0)\}^\star$	$\{(v_{14},0)\}$
v_{15}	$\{(v_{15},0)\}^\star$	$\{(v_{15},0)\}$
v_{16}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,1)$ $(v_{14},1),(v_{15},0),(v_{16},0)\}^\star$	$\{(v_{16},0)\}$

TABLE 5.16: Updated 2HC-SP labeling after executing DECPLL procedure as a result of the removal of v_{10} from G. In details, newly added label sets (modified label sets, respectively) are those with superscript "+" ("$\star$", respectively).

Insertion Phase The insertion phase (see Algorithm 2) as discussed in earlier sections restore the WRED-TE properties of G by inserting new vertices and arcs (if required). As our example is similar to that we have considered in the basic D-PTL case (see Section 5.1.4), therefore here will not go into details of how the arcs are rewired.

Given the updated WRED-TE graph G (shown in Fig 5.12) and the corresponding 2HC-SP labeling L (shown in Table 5.15), the algorithm inserts new vertices and arcs for representing the delayed connection in G. In particular, the algorithm proceeds by inserting at first the departure vertex, say v_{17} (following our assumption we discussed in Section 5.1.4). It then updates the waiting arcs, that is it removes (v_8, v_9) from A and add two new arcs (v_8, v_{17}) and (v_{17}, v_9) in A. This is followed by updating L (see Table 5.17) via INCPLL called as a sub-routine by Algorithm 2.

Vertex	2HC-SP Labeling	
	$\mathrm{L}_{in}(v_*)$	$\mathrm{L}_{out}(v_*)$
v_1	$\{(v_1, 0)\}$	$\{(v_1, 0)\}$
v_3	$\{(v_3, 0)\}$	$\{(v_1, 0), (v_3, 0)\}$
v_4	$\{(v_4, 0)\}$	$\{(v_4, 0)\}$
v_5	$\{(v_5, 0)\}$	$\{(v_1, 0), (v_3, 0), (v_4, 0), (v_5, 0)\}$
v_6	$\{(v_4, 1), (v_6, 0)\}$	$\{(v_6, 0)\}$
v_7	$\{(v_3, 0), (v_7, 0)\}$	$\{(v_7, 0)\}$
v_8	$\{(v_1, 1), (v_4, 1), (v_6, 0), (v_8, 0)\}$	$\{(v_8, 0)\}$
v_9	$\{(v_1, 1), (v_4, 1), (v_6, 0), (v_3, 1)$ $(v_7, 1), (v_8, 0), (v_9, 0)\}$	$\{(v_9, 0)\}$
v_{11}	$\{(v_{11}, 0)\}$	$\{(v_1, 1), (v_3, 1), (v_4, 0), (v_5, 1),$ $(v_{11}, 0)\}$
v_{12}	$\{(v_{12}, 0)\}$	$\{(v_1, 0), (v_3, 1), (v_{12}, 0)\}$
v_{13}	$\{(v_3, 0), (v_{13}, 0)\}$	$\{(v_7, 0), (v_{13}, 0)\}$
v_{14}	$\{(v_1, 1), (v_4, 1), (v_6, 0), (v_8, 0)$ $(v_{14}, 0)\}^{\star}$	$\{(v_{14}, 0)\}$
v_{15}	$\{(v_{15}, 0)\}$	$\{(v_{15}, 0)\}$
v_{16}	$\{(v_1, 1), (v_4, 1), (v_6, 0), (v_8, 1)$ $(v_{14}, 1), (v_{15}, 0), (v_{16}, 0)\}$	$\{(v_{16}, 0)\}$
${v_{17}}^{+}$	$\{(v_1, 1), (v_4, 1), (v_6, 0), (v_8, 0)$ $(v_{17}, 0)\}^{+}$	$\{(v_9, 0), (v_{17}, 0)\}^{+}$

TABLE 5.17: Updated 2HC-SP labeling after executing INCPLL procedure as a result of the addition of v_{17} (along with the arcs) in G. In details, newly added label sets (modified label sets, respectively) are those with superscript "+" ("$\star$", respectively).

Finally, for representing the arrival event of the delayed connection in G, the algorithm adds a new vertex, we refer to as v_{18}, in G. It then update A by adding arcs such as: i) a connection arc (v_{17}, v_{18}); ii) a transfer arc (v_{18}, v_{16}); and iii) a bypass arc (v_1, v_{18}) (see Fig. 5.13). The algorithm then update L (see Table 5.18) using INCPLL once again called as a sub-procedure by Algorithm 2. This conclude the steps required for updating G and L in order to reflect the changes caused by the delayed connection c_i. The changes to L are reflected in E-SL are described in the following.

Updating Extended Stop Labels After we have updated L, the D-PTL procedure, discussed in Section 5.2.4.2, dynamically update E-SL to reflect the changes made to L onto E-SL. The main idea is to recompute only those stop labels which possibly contain affected label entries. Here, we describe how E-SL is dynamically updated for our specific example.

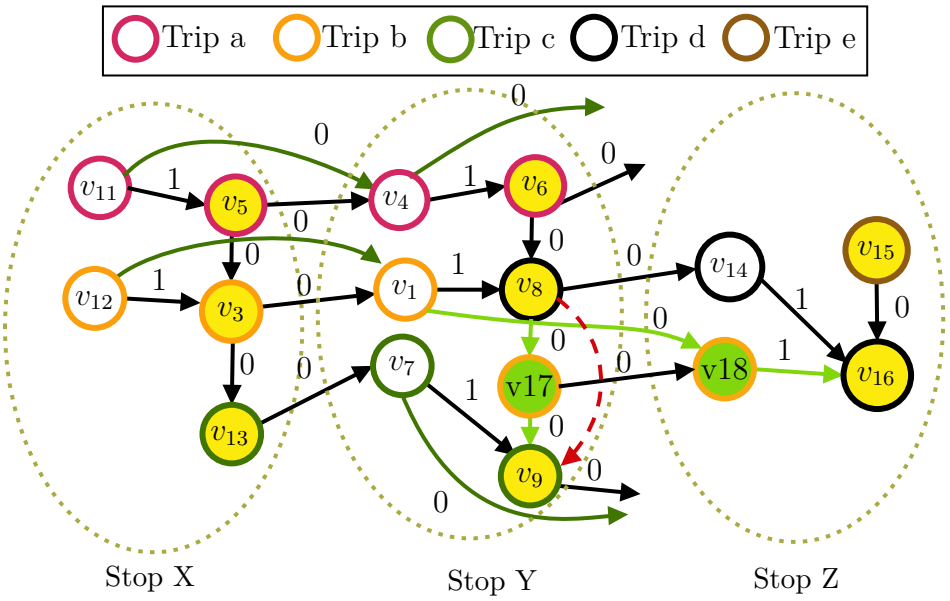

FIGURE 5.13: Updated WRED-TE graph after we have added new departure and arrival vertices. Newly added vertices and arcs are colored in green. Dashed arc in red is the removed waiting arc.

Vertex	2HC-SP Labeling	
	$\text{L}_{in}(v_*)$	$\text{L}_{out}(v_*)$
v_1	$\{(v_1,0)\}$	$\{(v_1,0)\}$
v_3	$\{(v_3,0)\}$	$\{(v_1,0),(v_3,0)\}$
v_4	$\{(v_4,0)\}$	$\{(v_4,0)\}$
v_5	$\{(v_5,0)\}$	$\{(v_1,0),(v_3,0),(v_4,0),(v_5,0)\}$
v_6	$\{(v_4,1),(v_6,0)\}$	$\{(v_6,0)\}$
v_7	$\{(v_3,0),(v_7,0)\}$	$\{(v_7,0)\}$
v_8	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)\}$	$\{(v_8,0)\}$
v_9	$\{(v_1,1),(v_4,1),(v_6,0),(v_3,1)$ $(v_7,1),(v_8,0),(v_9,0)\}$	$\{(v_9,0)\}$
v_{11}	$\{(v_{11},0)\}$	$\{(v_1,1),(v_3,1),(v_4,0),(v_5,1),$ $(v_{11},0)\}$
v_{12}	$\{(v_{12},0)\}$	$\{(v_1,0),(v_3,1),(v_{12},0)\}$
v_{13}	$\{(v_3,0),(v_{13},0)\}$	$\{(v_7,0),(v_{13},0)\}$
v_{14}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)$ $(v_{14},0)\}^{\star}$	$\{(v_{14},0)\}$
v_{15}	$\{(v_{15},0)\}$	$\{(v_{15},0)\}$
v_{16}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,1)$ $(v_{14},1),(v_{15},0),(v_{16},0)\}$	$\{(v_{16},0)\}$
v_{17}	$\{(v_1,1),(v_4,1),(v_6,0),(v_8,0)$ $(v_{17},0)\}$	$\{(v_9,0),(v_{17},0)\}$
$v_{18}{}^{+}$	$\{(v_1,0),(v_4,1),(v_6,0),(v_8,0)$ $(v_{17},0),(v_{18},0)\}^{+}$	$\{(v_{16},1),(v_{18},0)\}^{+}$

TABLE 5.18: Updated 2HC-SP labeling after executing INCPLL procedure as a result of the addition of v_{18} (along with the arcs) in G. In details, newly added label sets (modified label sets, respectively) are those with superscript "+" ("$\star$", respectively).

The procedure considers as input the updated 2HC-SP labeling L (shown in Table 5.18) and the two sets of stops, namely US_{out} and US_{in}. US_{out} and US_{in}, as discussed in earlier sections, are maintained throughout the process of updating L. In our case, we analyzed

the changes made to L, and report the summary in Table 5.19. From this, it is easy to see that US_{out} (US_{in}, respectively) contains Y (respectively, contains Z) only.

Stops	AV[∗]	DV[∗]
X	$\{v_{11}, v_{12}\}$	$\{v_5, v_3, v_{13}\}$
Y	$\{v_4, v_1, v_7\}$	$\{{v_6}^{\star}, {v_2}^{-}, v_8, {v_{17}}^{+}, v_9\}$
Z	$\{{v_{10}}^{-}, {v_{14}}^{\star}, {v_{18}}^{+}\}$	$\{v_{15}, v_{16}\}$

TABLE 5.19: Listing vertices in AV[i] (DV[i], respectively) for each $\text{S}_i \in \mathcal{S}$ in the second column (third column, respectively). The color of the vertices in second column can be interpreted as follows. A vertex v_j is with superscript: i) "+", if v_j is new vertex added to G in the insertion phase; ii) "$\star$", if $\text{L}_{in}(v_j)$ was modified either in the removal or insertion phase; and iii) "$-$", if $\text{L}_{in}(v_j)$ was removed either in the removal or insertion phase. The same hold for the entries in the third column, however, instead of $\text{L}_{in}(*)$, we considered $\text{L}_{out}(*)$.

Given US_{out} and US_{in}, the procedure recompute E-$\text{SL}_{out}(Y)$ and order the entries in E-$\text{SL}_{out}(Y)$ w.r.t the time of hubs. After which, the procedure recompute E-$\text{SL}_{in}(Z)$ and accordingly order the entries in E-$\text{SL}_{in}(Z)$. This conclude the process of dynamically updating E-SL. In Table 5.20, we report the updated E-SL obtained after applying the above steps.

Stops	Extended stop labeling (E-SL)	
	E-$\text{SL}_{in}(*)$	E-$\text{SL}_{out}(*)$
X	$\{(v_{11}, 5, 0), (v_{12}, 7, 0)\}$	$\{(v_5, 10, 0), (v_3, 15, 0), (v_4, 10, 0), (v_1, 15, 0), (v_{13}, 35, 0), (v_7, 35, 0)\}$
Y	$\{(v_4, 15, 0), (v_3, 42, 0), (v_1, 20, 0), (v_7, 42, 0)\}$	$\{(v_6, 0, 20), (v_8, 0, 30), (v_{17}, 35, 0), (v_9, 0, 50)\}^{\star}$
Z	$\{(v_1, 35, 0), (v_1, 39, 1), (v_4, 39, 1), (v_6, 39, 0), (v_8, 39, 0), (v_{18}, 35, 0), (v_{17}, 35, 0), (v_{14}, 39, 0)\}^{\star}$	$\{(v_{15}, 40, 0), (v_{16}, 50, 0)\}$

TABLE 5.20: Updated E-SL after G and L are modified in the removal and in insertion phases. Recomputed extended stop labels during the execution of Algorithm 6 are those with superscript "$\star$".

6

Experiments

In this Chapter, we present our experimental study to assess the performance of D-PTL, both basic and multi-criteria. In particular, Section 6.1 discusses the technical details of the environment, in terms of the underlying software and computational resources we have used for running our experiments. Section 6.2 discusses in details the experiments in terms of the assumptions we have made. Finally, Section 6.3 analyzes the results we obtained from our experiments.

6.1 Implementation Details

We implemented, in C++, both PTL and D-PTL, and developed a simulation environment to test the two algorithms on given input transit networks. All our code has been compiled with GNU g++ v.4.8.5 (O3 opt. level) under Linux (Kernel 4.4.0-148). All tests have been executed on a workstation equipped with an Intel Xeon© CPU and 128 GB of main memory.

6.2 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 work, 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 $m = \arg\max_{v \in \mathrm{DV}[s]} time(v)$. 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 Section 3.2. 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 [83]. We repeat the above process for 50 connections, in order to compute average values and collect statistically significant results.

6.2.1 Datasets

As inputs to our experiments, we considered, as other studies of this kind [28,29,41,52], real-world transit networks whose data is publicly available (see, Public Transit Feeds Archive – https://transitfeeds.com/). Details of the inputs considered for basic PTL are given in Table 6.1 while those of the inputs considered for the multi-criteria setting are given in Table 6.2. We remark that we were forced to use smaller inputs in the latter case with respect to the basic one, since the preprocessing phase is much more time consuming with respect to basic case and 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.

Network	**# stops**	**Graph**		**Preprocessing Time**		**Labeling Size**	
		$\|V\|$	$\|A\|$	L	SL	L	SL
London	5 221	3 066 852	5 957 246	4 494.00	5.19	5 856	529
Madrid	4 698	3 971 870	7 859 375	10 559.10	13.66	12 295	2 653
Rome	9 273	5 502 796	10 893 752	17 081.05	30.18	18 531	5 262
Melbourne	27 237	9 757 352	18 389 454	3 774.00	12.79	8 293	1 136

TABLE 6.1: 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	E-SL	L	E-SL
Palermo	1 714	563 064	1 112 110	7 828.00	3.43	4 687	372
Barcelona	3 232	1 201 256	2 075 005	14 207.00	7.99	4 219	359
Luxembourg	2 802	1 239 870	2 438 413	42 701.70	11.75	30 491	1129
Prague	4 940	1 755 078	2 475 801	29 288.60	18.57	4 243	694
Venice	2 173	1 373 674	2 526 500	19 114.03	5.87	7 426	189

TABLE 6.2: Details of input datasets for the multi-criteria setting: preprocessing time is expressed in seconds, labeling size in megabytes.

6.3 Analysis

The main results of our experiments are summarized in Tables 6.3–6.4, 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.

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	10495.40	14.20	416.55
Rome	12.36	14.49	16847.00	29.50	628.55
Melbourne	4.08	7.25	3807.00	11.50	337.03

TABLE 6.3: 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	(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	7 807.14	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	2 9435.90	20.84	40.86

TABLE 6.4: 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.

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 [37], 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 6.5–6.7 and Tables 6.6-6.8). 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 [31, 36, 37]. On top of that, a further consideration that can be done by analyzing the data in Table 6.7 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 [41] for a detailed comparison with the reduction provided by the original stop labeling). To summarize,

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 6.5: 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.

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;

Network	(Basic) D-PTL **Avg. Space** *(MB)*		(Basic) PTL **Avg. Space** *(MB)*	
	L	SL	L	SL
London	5 894	533	5 902	532
Madrid	12 391	2 669	12 395	2 663
Rome	1 8531	5 260	18 512	5 252
Melbourne	8 298	1 140	8 300	1 138

TABLE 6.6: 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.

- 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.

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 6.7: 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. Space** *(MB)*		(Multi-criteria) PTL **Avg. Space** *(MB)*	
	L	E-SL	L	E-SL
Palermo	4 764	372	4 737	373
Barcelona	4 253	360	4 219	360
Luxembourg	30 557	1 129	30 519	1 126
Venice	7 443	190	7 413	190
Prague	4 250	694	4 251	695

TABLE 6.8: 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 work (cf Section 5.2.4.2) while the recomputation from scratch is done by Algorithm 11, that is also not originally included in the PTL framework.

7

Conclusion and Future Research Directions

In this Chapter, we provide conclusive remarks and discuss potential research directions worthy to be investigated in the future. In particular, Section 7.1 concludes our work and provide a summary of what we done so far, while Section 7.2 discusses open research questions one can investigate in the future.

7.1 Conclusion

In this work, we have studied the journey planning problem in the context of 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 w.r.t. 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, for the basic PTL (that is for answering

earliest arrival and profile queries), 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 the basic PTL.

For the multi-criteria PTL (that is for answering multi-criteria queries), we extended the basic D-PTL framework to handle dynamic updates to 2-Hop Cover shortest path (2HC-SP) labeling. In this regard, we presented a new algorithm, named *DAG–Decremental-Pruned-Landmark-Labeling* (DAG-DECPLL), for dynamically updating 2HC-SP labelings in directed acyclic graphs as a consequence of decremental updates. The new method has been shown to be, empirically, much faster than the only known solution DECPLL [37]. For the sake of fairness, we recall that the latter works also for general graphs.

We integrated DAG-DECPLL and INCPLL [2] in the basic D-PTL framework to obtain what we called multi-criteria D-PTL. Moreover, for optimizing query times and space consumption when dealing with multi-criteria queries, we presented a new compact version of the data structure employed by the basic PTL. We have described an approach for answering to multi-criteria queries using the new compact version of the data structure we presented. Not only these, but we also presented a procedure for dynamically updating the compact version of the data structure employed by PTL in the multi-criteria setting. We have provided strong experimental evidences of the effectiveness of the compact representation in reducing the required query times.

We have experimentally shown the effectiveness of the multi-criteria D-PTL once again using real-world networks. Similar to the case with the basic D-PTL, the results we obtained for experimentally evaluating the multi-criteria D-PTL suggests that the multi-criteria D-PTL is, on average, always at least an order of magnitude smaller than that required by the recomputation from scratch, in the multi-criteria PTL.

7.2 Future Work

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.

Concerning the transitive closure of a graph, a research direction worthy to investigate might be that of extending the DECPLL to handle incremental updates specifically in DAG's, as we observed in our experiments that INCPLL, against our expectation, is comparatively slower than DECPLL. One way to achieve this could be that of extending the incremental Algorithm of [105].

Other research directions that might be pursued is that of presenting a new graph-based model for representing a timetable in a more compact way. This could be useful in many ways. In particular, the presence of a more compact graph-based representation of a timetable might help in reducing the time required for computing the 2-Hop-Cover labeling, and also the time required for dynamically updating it. Furthermore, the compact graph-based representation might help minimize the query times as the number of labels scanned for answering a query could be minimized.

Bibliography

[1] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. Hierarchical hub labelings for shortest paths. In *Proceeds of the European Symposium on Algorithms*, pages 24–35. Springer, 2012.

[2] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *Proceeds of the International World Wide Web Conference (WWW14)*, pages 237–248. ACM, 2014.

[3] Noga Alon, Paul Seymour, and Robin Thomas. Planar separators. *SIAM Journal on Discrete Mathematics*, 7(2):184–193, 1994.

[4] Bastian Amberg, Boris Amberg, and Natalia Kliewer. Robust efficiency in urban public transportation: Minimizing delay propagation in cost-efficient bus and driver schedules. *Transportation Science*, 53(1):89–112, 2019.

[5] Leonid Antsfeld and Toby Walsh. Finding multi-criteria optimal paths in multimodal public transportation networks using the transit algorithm. In *Proceeds of the the 19th ITS World Congress*, page 32, 2012.

[6] Renato Oliveira Arbex and Claudio Barbieri da Cunha. Efficient transit network design and frequencies setting multi-objective optimization by alternating objective genetic algorithm. *Transportation Research Part B: Methodological*, 81:355–376, 2015.

[7] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In *Proceeds of the International Symposium on Experimental Algorithms*, pages 55–66. Springer, 2013.

[8] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceeds of the European Symposium on Algorithms*, pages 290–301. Springer, 2010.

[9] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. Springer, 2016.

[10] Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-robustness of transfer patterns in public transportation route planning. In *Proceeds of the 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, 2013.

[11] Holger Bast, Stefan Funke, and Domagoj Matijević. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*, 2006.

[12] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566–566, 2007.

[13] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. Unlimited transfers for multi-modal route planning: An efficient solution. *arXiv preprint arXiv:1906.04832*, 2019.

[14] Elisabetta Bergamini and Henning Meyerhenke. Approximating betweenness centrality in fully dynamic networks. *Internet Mathematics*, 12(5):281–314, 2016.

[15] Annabell Berger, Andreas Gebhardt, Matthias Müller-Hannemann, and Martin Ostrowski. Stochastic delay prediction in large train networks. In *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[16] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In *Proceeds of the Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2011.

[17] Adi Botea. Fast, optimal pathfinding with compressed path databases. In *SOCS*, 2012.

[18] Adi Botea and Daniel Harabor. Path planning with compressed all-pairs shortest paths data. In *Proceeds of the Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[19] Adi Botea and Daniel Harabor. Path planning with compressed all-pairs shortest paths data. In *Proceeds of the Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

[20] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[21] Ulrik Brandes, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Travel planning with self-made maps. In *Proceeds of the Workshop on Algorithm Engineering and Experimentation*, pages 132–144. Springer, 2001.

[22] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3–15, 2004.

[23] Francesco Bruera, Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, and Daniele Frigioni. Dynamic multi-level overlay graphs for shortest paths. *Mathematics in Computer Science*, 1(4):709–736, 2008.

[24] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD13)*, pages 193–204. ACM, 2013.

[25] Boris V Cherkassky, Loukas Georgiadis, Andrew V Goldberg, Robert E Tarjan, and Renato F Werneck. Shortest-path feasibility algorithms: An experimental evaluation. *Journal of Experimental Algorithmics (JEA)*, 14:2–7, 2010.

[26] Boris V Cherkassky, Andrew V Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.

[27] Serafino Cicerone, Mattia D'Emidio, and Daniele Frigioni. On mining distances in large-scale dynamic graphs. In *Proceeds of the 19th Italian Conference on Theoretical Computer Science (ICTCS18)*, volume 2243 of *CEUR Workshop Proceedings*, pages 77–81. CEUR-WS.org, 2018.

[28] Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos D. Zaroliagis. Engineering graph-based models for dynamic timetable information systems. In *Proceeds of the Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS14)*, volume 42 of *OASICS*, pages 46–61. Schloss Dagstuhl, 2014.

[29] Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos D. Zaroliagis. Engineering graph-based models for dynamic timetable information systems. *Journal of Discrete Algorithms*, 46-47:40–58, 2017.

[30] Edith Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.

[31] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[32] Feliciano Colella, Mattia D'Emidio, and Guido Proietti. Simple and practically efficient fault-tolerant 2-hop cover labelings. In *Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic*, volume 1949 of *CEUR Workshop Proceedings*, pages 51–62. CEUR-WS.org, 2017.

[33] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[34] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *Proceeds of the International Symposium on Mathematical Foundations of Computer Science*, pages 722–731. Springer, 1998.

[35] G. D'Angelo, M. D'Emidio, and D. Frigioni. Fully dynamic update of arc-flags. *Networks*, 63(3):243–259, 2014.

[36] G. D'Angelo, M. D'Emidio, and D. Frigioni. Distance queries in large-scale fully dynamic complex networks. In *Proceeds of the International Workshop on Combinatorial Algorithms (IWOCA16)*, volume 9843 of *Lecture Notes in Computer Science*, pages 109–121. Springer, 2016.

[37] Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. Fully dynamic 2-hop cover labeling. *ACM Journal of Experimental Algorithmics*, 24(1):1.6:1–1.6:36, 2019.

[38] Gianlorenzo d'Angelo, Mattia d'Emidio, Daniele Frigioni, and Camillo Vitale. Fully dynamic maintenance of arc-flags in road networks. In *Proceeds of the International Symposium on Experimental Algorithms*, pages 135–147. Springer, 2012.

[39] George Bernard Dantzig. *Linear programming and extensions*. Princeton university press, 1998.

[40] Daniel Delling, Julian Dibbelt, and Thomas Pajor. Fast and exact public transit routing with restricted pareto sets. In *Proceeds of the Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 54–65. SIAM, 2019.

[41] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public transit labeling. In *Proceeds of the International Symposium on Experimental Algorithms (SEA15)*, volume 9125 of *Lecture Notes in Computer Science*, pages 273–285. Springer, 2015.

[42] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster transit routing by hyper partitioning. In *Proceeds of the Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[43] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2015.

[44] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[45] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17:4–4, 2012.

[46] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering time-expanded graphs for faster timetable information. In *Robust and Online Large-Scale Optimization*, pages 182–206. Springer, 2009.

[47] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2015.

[48] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In *Proceeds of the International Workshop on Experimental and Efficient Algorithms*, pages 52–65. Springer, 2007.

[49] Mattia D'Emidio and Imran Khan. Dynamic public transit labeling. In *International Conference on Computational Science and Its Applications*, pages 103–117. Springer, 2019.

[50] Mattia D'Emidio, Imran Khan, and Daniele Frigioni. Journey planning algorithms for massive delay-prone transit networks. *Algorithms*, 13(1):2, 2020.

[51] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *Proceeds of the International Symposium on Experimental Algorithms*, pages 43–54. Springer, 2013.

[52] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *ACM Journal of Experimental Algorithmics*, 23:1.7:1–1.7:56, 2018.

[53] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[54] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In *Proceeds of the International Workshop on Experimental and Efficient Algorithms*, pages 347–361. Springer, 2008.

[55] Twan Dollevoet, Dennis Huisman, Marie Schmidt, and Anita Schöbel. Delay propagation and delay management in transportation networks. In *Handbook of Optimization in the Railway Industry*, pages 285–317. Springer, 2018.

[56] MC Er. A parallel computation approach to topological sorting. *The Computer Journal*, 26(4):293–295, 1983.

[57] Cem Evrendilek. Vertex separators for partitioning a graph. *Sensors*, 8(2):635–657, 2008.

[58] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[59] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[60] Meng-yin Fu, Jie Li, and Pei-de Zhou. Design and implementation of bidirectional dijkstra algorithm. *JOURNAL-BEIJING INSTITUTE OF TECHNOLOGY-ENGLISH EDITION-*, pages 366–370, 2003.

[61] Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos Zaroliagis. Multimodal dynamic journey-planning. *Algorithms*, 12(10):213, 2019.

[62] Marc Goerigk, Sacha Heße, Matthias Müller-Hannemann, Marie Schmidt, and Anita Schöbel. Recoverable robust timetable information. In *Proceeds of the 13th Workshop on algorithmic approaches for transportation modelling, optimization, and systems*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[63] Andrew V Goldberg. Point-to-point shortest path algorithms with preprocessing. In *Proceeds of the International Conference on Current Trends in Theory and Practice of Computer Science*, pages 88–102. Springer, 2007.

[64] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R Sturtevant, Robert C Holte, and Jonathan Schaeffer. Enhanced partial expansion a. *J. Artif. Intell. Res.(JAIR)*, 50:141–187, 2014.

[65] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[66] Moritz Hilger, Ekkehard Köhler, Rolf H Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41–72, 2009.

[67] Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, and Novica Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *2012 proceedings of the 35th international convention MIPRO*, pages 1811–1815. IEEE, 2012.

[68] Sungwon Jung and Sakti Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, 2002.

[69] Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In *The Shortest Path Problem*, pages 19–40, 2006.

[70] Kamesh Madduri, David A Bader, Jonathan W Berry, and Joseph R Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *Proceeds of the of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, 2007.

[71] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *Journal of Experimental Algorithmics (JEA)*, 14:3–2, 2010.

[72] T Mellouli and L Suhl. Passenger online routing in dynamic networks. *Informationsprobleme in Transport und Verkehr*, 4:17–30, 2006.

[73] Rolf H Möhring. Verteilte verbindungssuche im öffentlichen personenverkehr graphentheoretische modelle und algorithmen. In *Angewandte Mathematik, insbesondere Informatik*, pages 192–220. Springer, 1999.

[74] Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria pareto search. In *Algorithmic Methods for Railway Optimization*, pages 246–263. Springer, 2007.

[75] Matthias Müller-Hannemann and Mathias Schnee. Finding all attractive train connections by multi-criteria pareto search. In *Algorithmic Methods for Railway Optimization*, pages 246–263. Springer, 2007.

[76] Matthias Müller-Hannemann and Mathias Schnee. Efficient timetable information in the presence of delays. In *Robust and Online Large-Scale Optimization*, pages 249–272. Springer, 2009.

[77] Matthias Müller-Hannemann and Karsten Weihe. On the cardinality of the pareto set in bicriteria shortest path problems. *Annals of Operations Research*, 147(1):269–286, Oct 2006.

[78] David Munoz, Frantz Bouchereau Lara, Cesar Vargas, and Rogerio Enriquez-Caldera. *Position location techniques and applications*. Academic Press, 2009.

[79] Karl Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

[80] Giacomo Nannicini, Daniel Delling, Dominik Schultes, and Leo Liberti. Bidirectional a* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012.

[81] Peng Ni, Masatoshi Hanai, Wen Jun Tan, Chen Wang, and Wentong Cai. Parallel algorithm for single-source earliest-arrival problem in temporal graphs. In *Proceeds of the 46th International Conference on Parallel Processing (ICPP)*, pages 493–502. IEEE, 2017.

[82] Duc-Minh Phan and Laurent Viennot. Fast public transit routing with unrestricted walking through hub labeling. *arXiv preprint arXiv:1906.08971*, 2019.

[83] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

[84] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics (JEA)*, 12:2–4, 2008.

[85] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D Zaroliagis. Experimental comparison of shortest path approaches for timetable information. In *ALENEX/ANALC*, pages 88–99. Citeseer, 2004.

[86] Yongrui Qin, Quan Z. Sheng, Nickolas J. G. Falkner, Lina Yao, and Simon Parkinson. Efficient computation of distance labeling for decremental updates in large dynamic graphs. *World Wide Web*, 20(5):915–937, 2017.

[87] Matteo Riondato and Evgenios M Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2016.

[88] Peter Sanders and Dominik Schultes. Robust, almost constant time shortest-path queries in road networks. In *The Shortest Path Problem*, pages 193–218, 2006.

[89] Dominik Schultes. Fast and exact shortest path queries using highway hierarchies. *Master-Arbeit, Universität des Saarlandes, Saarbrücken*, 2005.

[90] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: an empirical case study from public railroad transport. *Journal of Experimental Algorithmics (JEA)*, 5:12, 2000.

[91] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using multi-level graphs for timetable information in railway systems. In *Proceeds of the Workshop on Algorithm Engineering and Experimentation*, pages 43–59. Springer, 2002.

[92] Raimund Seidel. On the all-pairs-shortest-path problem. In *Proceeds of the of the twenty-fourth annual ACM symposium on Theory of computing*, pages 745–749, 1992.

[93] Ben Strasser and Dorothea Wagner. Connection scan accelerated. In *Proceeds of the the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 125–137. SIAM, 2014.

[94] Gintaras Vaira and Olga Kurasova. Parallel bidirectional dijkstra's shortest path algorithm. *Databases and Information Systems VI, Frontiers in Artificial Intelligence and Applications*, 224:422–435, 2011.

[95] Dorothea Wagner and Thomas Willhalm. Drawing graphs to speed up shortest-path computations. In *ALENEX/ANALCO*, pages 17–25, 2005.

[96] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric containers for efficient shortest-path computation. *Journal of Experimental Algorithmics (JEA)*, 10:1–3, 2005.

[97] Dorothea Wagner and Tobias Zündorf. Public transit routing with unrestricted walking. In *Proceeds of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[98] Dorothea Wagner and Tobias Zündorf. Public transit routing with unrestricted walking. In *Proceeds of the17th Workshop on Algorithmic Approaches for*

Transportation Modelling, Optimization, and Systems (ATMOS 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[99] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceeds of the 2015 ACM International Conference on Management of Data (SIGMOD15)*, pages 967–982. ACM, 2015.

[100] Arthur Warburton. Approximation of pareto optima in multiple-objective, shortest-path problems. *Oper. Res.*, 35(1):70–79, February 1987.

[101] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *Proceedings of the22nd ACM International Conference on Information & Knowledge Management (CIKM13)*, pages 1601–1606. ACM, 2013.

[102] Hilmi Yıldırım, Vineet Chaoji, and Mohammed J Zaki. Grail: a scalable index for reachability queries in very large graphs. *The VLDB Journal*, 21(4):509–534, 2012.

[103] Fang Zhao. Large-scale transit network optimization by minimizing user cost and transfers. *Journal of Public Transportation*, 9(2):6, 2006.

[104] Fang Zhao and Xiaogang Zeng. Optimization of user and operator cost for large-scale transit network. *Journal of Transportation Engineering*, 133(4):240–251, 2007.

[105] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD14)*, pages 1323–1334. ACM, 2014.