Dynamic Public Transit Labeling

Mattia D'Emidio[1(✉)] and Imran Khan[2(✉)]

[1] Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, L'Aquila, Italy
mattia.demidio@univaq.it
[2] Gran Sasso Science Institute (GSSI), L'Aquila, Italy
imran.khan@gssi.it

Abstract. We study the journey planning problem in transit networks which, given the timetable of a *schedule-based transit system*, asks to answer to queries such as, e.g., "seek a journey that arrives at a given destination as early as possible". The state-of-the-art solution to such problem, in terms of query time, is *Public Transit Labeling* (PTL), proposed in [Delling et al., SEA 2015], that consists of three main ingredients: (i) a graph data structure for storing transit networks; (ii) a compact labeling-based representation of the transitive closure of such graph, computed via a time-consuming preprocessing routine; (iii) an efficient query algorithm exploiting both graph and precomputed data to answer quickly to queries of interest at runtime.

The major drawback of PTL is not being practical in *dynamic scenarios*, when the network's timetable can undergo updates (e.g. *delays*). In fact, even after a single change, precomputed data become outdated and queries can return incorrect results. Recomputing the labeling-based representation from scratch, after a modification, is not a viable option as it yields unsustainable time overheads. Since transit networks are inherently dynamic, the above represents a major limitation of PTL.

In this paper, we overcome such limit by introducing a *dynamic algorithm*, called D-PTL, able to update the preprocessed data whenever a delay affects the network, without recomputing it from scratch. We demonstrate the effectiveness of D-PTL through a rigorous experimental evaluation showing that its update times are orders of magnitude smaller than the time for recomputing the preprocessed data from scratch.

Keywords: Journey planning · Dynamic graph algorithms · 2-Hop-Cover labeling · Algorithm engineering · Massive datasets · Networks

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

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11619, pp. 103–117, 2019.
https://doi.org/10.1007/978-3-030-24289-3_9

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 ever traveled [2]. In particular, the *journey planning problem* asks, given an *input timetable* (i.e. the description, in terms of departure and arrival times, of transits of vehicles between stops within the system) to answer to natural queries such as, e.g., "What is the best journey from some stop A to some other stop B if I want to depart at time t?".

Despite its simple formulation, the problem is much more challenging w.r.t., for instance, to the route planning problem in road networks which has been the subject of many recent studies (see, e.g., [2,9,12]). This is due to the fact that schedule-based transit systems exhibit an inherent time-dependent component that requires more complex modeling assumptions to obtain meaningful results. For this reason, recently, transit companies have invested a lot of resources to develop software systems, called *journey planners* (see, e.g., *Google Transit*[1] or *bahn.de*[2]), to be able to answer efficiently to such kind of queries and to provide best journeys w.r.t. to some metric of interest. Depending on the considered metric and modeling assumptions, the problem can be specialized into a plethora of optimization problems [2].

The most common type of query is the *earliest arrival query*, which asks for computing a journey that minimizes the total traveling time from a given departure stop to a given arrival stop, if one departs at a distinguished 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 departure time can lie within a given range. Further types of queries can be obtained by considering multiple optimization criteria simultaneously or according to the abstraction at which the problem has to be solved. If, for instance, one wants to optimize the time required by a passenger for moving from one vehicle to another one within a stop (i.e., *transfer time*), then the problem is called *realistic* while it is referred to as *ideal* otherwise [6]. In this paper, we focus on the realistic scenario. To solve the mentioned variants, a great variety of models and techniques have been proposed in the literature [5,11,13–15]. We refer to the very recent survey of Bast et al. [2] for a comprehensive overview.

The state-of-the-art method, achieving the smallest query times on large sets of real-world inputs, is *Public Transit Labeling* (PTL, for short) a preprocessing-based approach that has been experimentally shown to outperform all other solutions, achieving order of milliseconds query times on average even in continental-sized networks [4,7,11,17]. Such approach essentially consists of three main ingredients: (i) a well-known graph data structure for storing transit networks, i.e. the *time–expanded graph*; (ii) a compact labeling-based 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

[1] https://maps.google.com/landing/transit/index.html.
[2] https://www.bahn.de.

and the precomputed data to answer quickly to queries of interest at runtime. 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 change to the network, queries can return incorrect results since the preprocessed data can become easily outdated and do not reflect properly the transitive closure. Recomputing the labeling-based data from scratch, after an update occurs, is not a viable option as it yields unsustainable time overheads, up to tens of hours [11]. 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 [1,8,10,16]. 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.

Our Contribution. In this paper, we overcome the above mentioned limit by presenting a new *dynamic 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 performing a recomputation it from scratch. Notice that, decreases in departure times are typically not allowed in transit networks, since, to preserve the integrity of pre-planned connections, if a vehicle arrives early at a given stop, it just waits [2]. Hence updating the information in such case is not necessary. However, our solution can be easily extended to manage such scenario.

We discuss the correctness of D-PTL and analyze its computational complexity in the worst case. Asymptotically speaking, the proposed solution is worse 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 D-PTL is able to update both the graph and the labeling structure orders of magnitude faster than the recomputation from scratch. 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 making D-PTL an effective approach to handle the journey planning problem in dynamic transit networks.

Structure of the Paper. The paper is organized as follows. Section 2 gives the notation, describe the basics of PTL and of the labeling technique [11]. In Sect. 3 we present our new dynamic algorithm and discuss its correctness and complexity. Section 4 describe our experimental study while Sect. 5 concludes paper and outlines possible future research directions.

2 Background

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

A *trip* $\textsc{trip}_i = (c_1, c_2, \ldots, c_k)$ is a sequence of k connections that: (i) are operated by a same vehicle; (ii) share pairwisely departure and arrival stop, i.e., formally, we have $c_{i-1} = (Z, s_i, s_j, t_d, t_a)$ and $c_i = (Z, s_j, s_k, t'_d, t'_a)$ with $t'_d > t_d$ for any $i \in [2, k]$. Clearly, connections in a trip are ordered in terms of the associated departure times, hence we say connection c_j *follows* connection c_{j-1} in a trip $\textsc{trip}_i$ 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 $\textsc{trip}_i$.

An *earliest arrival query* $\textsc{ea}(s_i, s_j, \tau)$ asks, given a triple s_i, s_j, τ of *source stop* s_i, *target stop* s_j and 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. A *journey* $J = (c_1, c_2, \ldots c_n)$ connecting two stops s_i to s_j is a sequence of n connections that: (i) can be serviced 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 stop s_h, then the departure time of c_{i+1} must be larger than the arrival time of c_i plus $\textsc{mttp}_h$. As well as trips, journeys are implicitly ordered by time according 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. A *profile query* $\textsc{pq}(s_i, s_j, \tau, \tau')$ asks for the *set of non-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-dominated journeys. A journey is non-dominated if and only if the departure time (arrival time, respectively) of its first connection is smaller than that of any other in the set.

Public Transit Labeling. The state-of-the art approach to answer the above queries is based on a graph representation of the transit network [11], and it is discussed in the following.

In particular, the input timetable $\mathcal{T} = (\mathcal{Z}, \mathrm{S}, \mathcal{C})$ associated to the transit network is modeled via a *reduced time–expanded graph* (RED-TE, for short) [6], that is a directed acyclic graph $G(V, A)$. Note that aperiodic timetables can be effectively represented as directed acyclic graphs [11]. Starting from initially empty sets V and A, the graph is built as follows. For each elementary connection $c = (Z, s_i, s_j, t_d, t_a)$ we add two vertices to V, namely a *departure vertex* v_d^c and *arrival vertex* v_a^c, respectively, each having an associated time $time(v_d^c)$ and $time(v_a^c)$, respectively, that is equal to the departure and arrival times of the connection, i.e. $time(v_d^c) = t_d$ 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). In addition, for each connection we also insert a directed *connection arc* (v_d^c, v_a^c) to A, connecting the corresponding two departure and arrival vertices. Moreover, for each trip $\mathrm{TRIP}_i = (c_0, c_1, \ldots, c_k)$ we add to A a *bypass arc* $(v_a^{c_i}, v_a^{c_{i+1}})$ connecting the two arrival vertices. Furthermore, for any pair of vertices $u, v \in \mathrm{DV}^i$, we add to A a *waiting arc* (u, v) if $time(v) \geq time(u)$ and there is no w in DV^i such that $time(v) \geq time(w) \geq time(u)$. Finally, for each $u \in \mathrm{AV}^i$ and for each $v \in \mathrm{DV}^i$, we add to A a *transfer arc* (u, v) if $time(v) \geq time(u) + \mathrm{MTTP}_i$ and there is no $w \in \mathrm{DV}^i$ such that $time(w) < time(v)$ and $time(w) \geq time(u) + \mathrm{MTTP}_i$.

Given a RED-TE graph $G = (V, A)$, we say a vertex u *is reachable from* (*reaches*, respectively) another vertex v if and only if there exists a path from v to u (from u to v, respectively) in G, i.e. a sequence of arcs $((v, v_1), (v_1, v_2), \ldots, (v_k, u))$. The *cost* of a path is the sum of weights of the arcs in the path, where the *weight* of an arc (i, j) is given by the positive (by construction) difference $time(j) - time(i)$ between the times of its endpoint vertices. It is easy to see how, in a RED-TE graph built as described above, all paths from a vertex u to a vertex v in G have the same cost [6,11], which is given by $time(v) - time(u)$ (again positive by construction). Given a graph G, any approach for computing a so–called 2-Hop-Cover *reachability labeling* L (2HCR labeling, for short) of G associates two labels to each vertex $v \in V$, namely a *backward label* $\mathrm{L}_{in}(v)$ and a *forward label* $\mathrm{L}_{out}(v)$ [7], where a label is a set of vertices of G. In particular, for any two vertices $u, v \in V$, $\mathrm{L}_{out}(u) \cap \mathrm{L}_{in}(v) \neq \emptyset$ if and only if there exists a path from u to v in G [7]. Vertices $\{h : h \in \mathrm{L}_{out}(u) \cap \mathrm{L}_{in}(v)\}$ 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 2HCR labeling is given by the sum of the sizes of the label entries and it is known that computing a 2HCR labeling of minimum size is NP-Hard [7]. However, numerous approaches have been presented to heuristically improve both the time to compute the labeling and its size [3,18,19]. Among them, the one in [19], called BUTTERFLY, has been

shown to exhibit superior performance for directed acyclic graphs. Moreover, the solution is suited for *dynamic graphs*, i.e. the authors also provide a dynamic algorithm that is able to update the 2HCR labeling L of a graph G to reflect changes occurring on G itself. In particular, given a graph G, a 2HCR labeling L of G, and an update operation occurring on G, the algorithm is able to compute another labeling L$'$ that is a 2HCR labeling for G', where G' is the graph obtained by applying the update on G. Note that, in this scenario, updates can be *incremental* (*decremental*, respectively), if they are additions (removals, respectively) of a vertex/arc. Throughout the paper, we denote by INC-BU(G, L, v) (DEC-BU(G, L, v), respectively) the result of the application of the dynamic algorithm of [19] to the labeling L of a graph G to handle an incremental (decremental, respectively) operation occurring on vertex/arc v (we refer the reader to [19] for more details on the above dynamic algorithms).

It is known that any 2HCR labeling, along with the above described RED-TE graph model, can be used to answer to queries of interest on timetable (we refer the reader to [11] for more details). However, in order to obtain very fast query time, compatible with modern applications, in [11] a customization of the general approach, tailored for RED-TE graphs, has been proposed under the name of *Public Transit Labeling* (PTL, for short).

The main idea underlying PTL is to compact labels, and to associate them to stops, rather than to vertices. In particular, PTL computes a RED-TE graph G, a 2HCR labeling L of G, and a set of *stop labels* SL of L [11]. For the sake of clarity we remark that the approach in [11] relies on a classic time–expanded graph. However, there is a one-to-one correspondence between RED-TE graphs and classic time–expanded graphs [6].

In details, we have a *forward stop label* $\mathrm{SL}_{out}(i)$ and a *backward stop label* $\mathrm{SL}_{in}(i)$ for each stop $s_i \in \mathrm{S}$. 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 (that reaches, respectively) from at least one vertex in DV^i (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 efficiency purposes, entries in $\mathrm{SL}_{out}(i)$ ($\mathrm{SL}_{in}(i)$, respectively) are stored as sorted arrays, in increasing order of hub vertices (according to distinct ids are assigned to vertices). The set of stop labels is usually referred to as *stop labeling* of G (or of L). Similarly to the general 2HCR labeling case, queries on the timetable can be answered via stop labels by scanning the entries associated to source and target stops. Query algorithms exploit the information in the stop labeling to discard dominated journeys to the stored hubs and to achieve query times of the order of milliseconds [11].

3 Dynamic Public Transit Labeling

In this section, we introduce *Dynamic Public Transit Labeling* (D-PTL, for short), a new dynamic algorithm able to update a RED-TE graph $G = (V, A)$, the corresponding 2HCR labeling L and stop labeling SL, as a consequence of a delay

affecting a connection of the transit network. Formally, a *delay* is an increase in the departure time of an elementary connection of a finite quantity $\delta > 0$. It is easy to see how a delay can induce an arbitrary number of changes to both the graph and labelings [6,11], depending on the structure of the trip the connection belongs to, thus in turn inducing arbitrarily wrong answers to queries.

A general dynamic strategy to achieve the purpose of updating both G, the 2HCR 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 [5,6,15]) and then reflect all these changes on both L and SL by: (i) detecting and removing obsolete label entries; and (ii) adding new updated label entries induced by the new graph, as done in other works on the subject [10,19]. However, this results in a quite high computational effort, as shown by preliminary experimentation we conducted. Thus, in order to minimize the number of changes to both L and SL, we exploit the specific structure of the RED-TE graph and propose a dynamic algorithm that interleaves phases of update of the graph with phases of update of the labeling L via the dynamic algorithms of [19]. 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 w.r.t. departure time, by executing two subroutines, called, respectively, *removal phase* (Algorithm REM-D-PTL for short) and *insertion phase* (Algorithm INS-D-PTL for short) that update L along with the graph. The former, described in Algorithm 1, takes care of removing from G vertices and arcs, associated with the delayed connection, that *violate* the RED-TE constraints.

We say a vertex (arc, respectively) *violates* the RED-TE constraints whenever the associated time (the difference of the times of the endpoints, respectively) does not satisfy at least one of the inequalities imposed by the RED-TE model discussed in Sect. 2. Note that, vertex and arcs of the above kind can be: (a) departure and arrival vertices of the delayed connection; (b) departure and arrival vertices following the delayed connection in the same trip; (c) arcs adjacent to vertices in (a) and (b).

The latter, instead, described in Algorithm 2, adds to G vertices and arcs, according to the delayed connection, in such a way G is a RED-TE graph properly representing the updated timetable. Then, it also updates accordingly L.

The behavior of algorithm is customized depending on the effect of the delay on the original G. In particular, we have a different update path by Algorithm 2 depending on whether the vertices associated to the connection are removed or not from the graph by Algorithm 1. We distinguish four cases:

Algorithm 1. Algorithm REM-D-PTL.

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

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

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

In more details Algorithm 2 incorporates specific sub-routines, described in Algorithms 3, 4 and 5, whose purpose is to update of specific parts of the graph and to handle the mentioned different topological cases.

Finally, the two above phases are then followed by a bundle update of SL by a suitable procedure, described in Sect. 3.1.

Algorithm 2. Algorithm INS-D-PTL.

Input: Graph G, delay $\delta > 0$, delayed connection c_m, trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$

Output: RED-TE graph G including vertices of connections affected by the delay, 2HCR labeling L of G

foreach c_j, $j = m$ ***to*** k **do**
  Let s_s and s_t be departure and arrival, respectively, stops of c_j;
  $\text{PRED} \leftarrow \infty$; $\text{SUCC} \leftarrow \infty$;
  foreach $v \in N_{out}(v_d^{c_j})$ **do** // Outgoing arcs (if any)
    if $v \in \text{DV}^s$ **then** // Waiting arc in the graph
      $\text{SUCC} \leftarrow v$;
  foreach $v \in N_{in}(v_d^{c_j})$ **do** // Incoming arcs (if any)
    if $v \in \text{DV}^s$ **then** // Waiting arc in the graph
      $\text{PRED} \leftarrow v$;
  if $v_d^{c_j} \in V$ ***and*** $v_a^{c_j} \in V$ **then** // Case a) - Both not removed
    Call RewireTransferDep$(G, v_d^{c_j}, \text{SUCC}, s_s)$;
    if G *has changed* **then**
      $\text{L} \leftarrow \text{INC-BU}(G, \text{L}, v_d^{c_j})$;
  else if $v_d^{c_j} \notin V$ ***and*** $v_a^{c_j} \notin V$ **then** // Case b) - Both Removed
    $V \leftarrow V \cup \{v_d^{c_j}\}$; // Add $v_d^{c_j}$ to G
    Call RewireWaitingDep$(G, v_d^{c_j}, s_s)$;
    Call RewireTransferDep$(G, v_d^{c_j}, \text{SUCC}, s_s)$;
    $\text{L} \leftarrow \text{INC-BU}(G, \text{L}, v_d^{c_j})$;
    $V \leftarrow V \cup \{v_a^{c_j}\}$; $A \leftarrow A \cup \{(v_d^{c_j}, v_a^{c_j})\}$; // Add $v_a^{c_j}$ and connection arc to G
    Call RewireArr$(G, v_a^{c_j}, \text{TRIP}_i, s_t)$;
    $\text{L} \leftarrow \text{INC-BU}(G, \text{L}, v_a^{c_j})$;
  else if $v_d^{c_j} \notin V$ ***and*** $v_a^{c_j} \in V$ **then** // Case c) - Only $v_d^{c_j}$ removed
    $V \leftarrow V \cup \{v_d^{c_j}\}$;
    $A \leftarrow A \cup \{(v_d^{c_j}, v_a^{c_j})\}$; // Add $v_d^{c_j}$ and connection arc to G
    Call RewireWaitingDep$(G, v_d^{c_j}, s_s)$;
    Call RewireTransferDep$(G, v_d^{c_j}, \text{SUCC}, s_s)$;
    $\text{L} \leftarrow \text{INC-BU}(G, \text{L}, v_d^{c_j})$;
  else // Case d) - Only $v_a^{c_j}$ removed
    $V \leftarrow V \cup \{v_a^{c_j}\}$;
    $A \leftarrow A \cup \{(v_d^{c_j}, v_a^{c_j})\}$; // Add $v_a^{c_j}$ and connection arc to G
    Call RewireArr$(G, v_a^{c_j}, \text{TRIP}_i, s_t)$;
    $\text{L} \leftarrow \text{INC-BU}(G, \text{L}, v_a^{c_j})$;

3.1 Updating the Stop Labeling

Once both the graph and the 2HCR 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 the scratch, via e.g. the routine

Algorithm 3. Algorithm RewireTransferDep.

Input: RED-TE graph G, vertices $v_d^{c_j}$ and SUCC, stop s_s
Output: (Partially updated) RED-TE graph G

if SUCC $= \infty$ **then** // Case a.1)
  CANDIDATES $\leftarrow \emptyset$;
  foreach $v \in \text{AV}^s$ **do**
    TO_ADD $\leftarrow$ *true*;
    foreach $u \in N_{out}(v)$ **do** // Outgoing transfer arcs (if any)
      if $u \in \text{DV}^s$ **then** // Has transfer arc
        TO_ADD $\leftarrow$ *false*;
        break;
    if TO_ADD ***and*** $time(v_d^{c_j}) \geq time(v) + \text{MTTP}_s$ **then**
    CANDIDATES $\leftarrow$ CANDIDATES $\cup \{v\}$;
  foreach $v \in$ CANDIDATES **do** $A \leftarrow A \cup \{(v, v_d^{c_j})\}$;
else // Case a.2)
  $T \leftarrow \emptyset$;
  foreach $v \in \text{AV}^s$ **do**
    foreach $u \in N_{out}(v)$ **do** // Outgoing transfer arcs (if any)
      if $u =$ SUCC ***and*** $time(v_d^{c_j}) \geq time(v) + \text{MTTP}_s$ **then**
      $T \leftarrow T \cup \{(v, u)\}$;
  foreach $(v, u) \in T$ **do**
    $A \leftarrow A \setminus \{(v, u)\}$;
    $A \leftarrow A \cup \{(v, v_d^{c_j})\}$;

Algorithm 4. Algorithm RewireWaitingDep.

Input: (Partially updated) RED-TE graph G, vertex $v_d^{c_j}$, stop s_s
Output: (Partially updated) RED-TE graph G

if $\text{DV}^s \setminus \{v_d^{c_j}\} \neq \emptyset$ **then**
  $m \leftarrow argmax_{v \in \text{DV}^s} time(v)$;
  if $time(v_d^{c_j}) \geq time(m)$ **then** // Add waiting arc
    $A \leftarrow A \cup \{(m, v_d^{c_j})\}$;
else
  Let $m_1, m_2 \in \text{DV}^s$ be such that $time(m_1) \leq time(v_d^{c_j}) \leq time(m_2)$;
  $A \leftarrow A \setminus \{(m_1, m_2)\}$; // Remove outdated waiting arc
  $A \leftarrow A \cup \{(m_1, v_d^{c_j}), (v_d^{c_j}, m_2)\}$; // Add new waiting arcs

in [11]. This computational effort is not large as that required for recomputing the 2HCR labeling. However, we propose a dynamic routine that is incorporated in D-PTL and avoids (and it is faster than) the recomputation from scratch of the stop labeling as well, described in what follows. Our routine requires, during the execution of Algorithms 1, 2, to compute two sets of so–called *updated stops*, denoted, respectively, by US_{out} and US_{in}. These are defined as the stops $s_i \in \text{S}$ such that vertices in DV^i (AV^i, respectively) had their time value or forward label (backward label, respectively) changed during Algorithm REM-D-PTL or during

Algorithm 5. Algorithm REWIREARR.

Input: (Partially updated) RED-TE graph G, vertex $v_a^{c_j}$, trip $\text{TRIP}_i = (c_0, c_1, \ldots, c_m, \ldots, c_k)$, stop s_t.
Output: (Partially updated) RED-TE graph G

```
1  TEMP_NODE ← ∞;
2  TEMP_TIME ← ∞;
3  foreach v ∈ DV^t do
4      if time(v) ≥ time(v_a^{c_j}) + MTTP_t then          // Search for minimum
5          if time(v) ≤ TEMP_TIME then
6              TEMP_TIME ← time(v);
7              TEMP_NODE ← v;
8  A ← A ∪ {(v_a^{c_j}, TEMP_NODE)};                      // Add proper transfer arc
9  if j > 0 then                                          // Not first connection of the trip
10     A ← A ∪ {(v_a^{c_{j-1}}, v_a^{c_j})};               // Add bypass arc
11 if j < k then                                          // Not last connection of the trip
12     A ← A ∪ {(v_a^{c_j}, v_a^{c_{j+1}})} ;              // Add bypass arc
```

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 time or labels.

Once this is done we update the stop labeling SL by recomputing only the entries of $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) for each $s_i \in \text{US}_{out}$ (for each $s_i \in \text{US}_{in}$, respectively). To this aim, for each stop $s_i \in \text{US}_{out}$ ($s_i \in \text{US}_{in}$, respectively) we first reset $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) to the emptyset. Then, we scan departure (arrival, respectively) vertices in decreasing (increasing, respectively) order w.r.t. time and add entries to $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) accordingly. In particular, for all departure (arrival, respectively) vertices v of s_i in the above mentioned order, we add a pair $(u, \mathit{stoptime}_i(v))$ for each u in $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) only if there is no pair $\text{SL}_{out}(i)$ ($\text{SL}_{in}(i)$, respectively) having u as hub vertex. This guarantees that each pair contains latest departure (earliest arrival, respectively) times. After updating the stop labels, we sort both $\text{SL}_{out}(i)$ and $\text{SL}_{in}(i)$ to restore the ordering according to the hub vertices [11]. 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, whose proof, due to space limitations, is deferred to the full version of the paper.

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

Algorithm 6. Algorithm for updating the stop labeling.

Input: Outdated stop labeling SL, 2HCR labeling L of G, sets US_{out}, US_{in}
Output: Stop labeling SL of L

foreach $s_i \in \text{US}_{out}$ **do**
  $Q \leftarrow \emptyset$;
  $\text{SL}_{out}(i) \leftarrow \emptyset$;
  while $\{\text{DV}^s \setminus Q\} \neq \emptyset$ **do**
    $m \leftarrow argmax_{v \in \text{DV}^s}\, time(v)$;
    foreach $u \in \text{L}_{out}(m)$ **do**
      if $u \notin \text{SL}_{out}(i)$ **then**
        Add $(u, time(m))$ to $\text{SL}_{out}(i)$;
    $Q \leftarrow Q \cup \{m\}$;
Sort $\text{SL}_{out}(i)$ w.r.t. hub vertices;
foreach $s_i \in \text{US}_{in}$ **do**
  $Q \leftarrow \emptyset$;
  $\text{SL}_{in}(i) \leftarrow \emptyset$;
  while $\{\text{AV}^s \setminus Q\} \neq \emptyset$ **do**
    $m \leftarrow argmin_{v \in \text{DV}^s}\, time(v)$;
    foreach $u \in \text{L}_{in}(m)$ **do**
      if $u \notin \text{SL}_{in}(i)$ **then**
        Add $(u, time(m))$ to $\text{SL}_{in}(i)$;
    $Q \leftarrow Q \cup \{m\}$;
Sort $\text{SL}_{in}(i)$ w.r.t. hub vertices;

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

Notice that, Theorem 2 implies that D-PTL is slower than the recomputation from scratch via PTL in the worst case. However, our experimental study shows D-PTL always outperforms PTL in practice.

4 Experimental Study

In this section, we present our experimental study to assess the performance of D-PTL. In particular, we implemented, in C++, both PTL and D-PTL, and developed a simulation environment to test the two algorithms on given input transit networks. We conducted experiments as follows.

For each input, we build the RED-TE graph G and execute PTL to compute both the 2HCR labeling L and the stop labeling SL. Then, we select a connection c_j of the timetable uniformly at random and delay it by δ minutes, where δ is randomly chosen within $[5, time(m) - time(v_d^{c_j}) + 10]$ and $m = argmax_{v \in \text{DV}^s}\, time(v)$. Finally, we run D-PTL to update both the graph and the labelings. In parallel, we run PTL to recompute graph and labelings from scratch. We repeat the above for 50 connections After each execution, we measure both the update time of D-PTL and the running time of PTL. 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 labelings while the latter is obtained by computing the average time to answer to 100 000 queries, of both earliest arrival and profile type via the query algorithms of [11]. This is done to evaluate the quality of the obtained data structures w.r.t. relevant factors like size and query time. 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 [15].

As inputs to our experiments, we considered real-world transit networks whose data is publicly available [3], as done in other studies of this kind [5,6,11,14]. Details of the considered inputs are given in Table 1 where we report, for each network, the number of stops, the size of the corresponding RED-TE graph $G = (V, A)$, the time for preprocessing the network to compute the 2HCR labeling L and the stop labeling SL, resp., and the size of both L and SL, in megabytes. All our code has been compiled with GNU g++ v.5 (O3 opt. level) under Linux (Kernel 4.4.0-47). All tests have been executed on a workstation equipped with an Intel Xeon© CPU and 128 GB of main memory.

Table 1. Details of input datasets: preprocessing time is expressed in seconds, sizes 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

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

In Table 2 we can observe that D-PTL is able to update L and SL in a time that is orders of magnitude smaller than that taken by the recomputation (up to more than 600 times smaller). Moreover, the speed-up seem to increase as the network size increases, thus suggesting that D-PTL scales well against input size. Furthermore, our experiments show that graphs and labelings updated via D-PTL and those recomputed from scratch are equivalent in terms of both query times and space overhead, thus confirming that the use of D-PTL does not induce any degradation in the performance of the data structures. Results supporting these claims are postponed to the full version of the paper due to space constraints.

[3] Public Transit Feeds Archive – https://transitfeeds.com/.

All above observations are a strong evidence of the fact that 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.

Table 2. Comparison between D-PTL and PTL in terms of computational time, in seconds, to update and recompute from scratch, resp., the data structures.

Network	D-PTL		PTL		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

5 Conclusion

We have studied the journey planning problem and have presented D-PTL, the first dynamic algorithm that enables the use of the state-of-the-art solution (PTL) in dynamic scenarios. We have shown, through extensive experimentation, that D-PTL is orders of magnitude faster than the recomputation from scratch, while at the same time preserving the performance in terms of query time and space overhead. Several research directions deserve further investigation. Perhaps the most relevant one is to adapt D-PTL to support multi-criteria queries. In [11] such queries are handled by encoding transfers as arc costs, by computing shortest path labels based on these costs, and by adjusting the query algorithm to compute Pareto optimal solutions. Unfortunately updating shortest path labels is a challenging task, computationally speaking [1,8,10] and it would be interesting to adapt one known technique to the scenario considered in this paper. Another worth future work could be to extend the experimentation to larger and more diverse inputs, to strengthen the obtained conclusions.

References

1. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Proceedings of 23rd International World Wide Web Conference (WWW 14), pp. 237–248. ACM (2014)
2. Bast, H., et al.: Route planning in transportation networks. In: Kliemann, L., Sanders, P. (eds.) Algorithm Engineering. LNCS, vol. 9220, pp. 19–80. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49487-6_2
3. Cheng, J., Huang, S., Wu, H., Fu, A.W.C.: Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In: Proceedings of 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD 2013), pp. 193–204. ACM (2013)

4. Cicerone, S., D'Emidio, M., Frigioni, D.: On mining distances in large-scale dynamic graphs. In: Aldini, A., Bernardo, M., (eds.) Proceedings of the 19th Italian Conference on Theoretical Computer Science, Urbino, Italy. CEUR Workshop Proceedings, vol. 2243, pp. 77–81. CEUR-WS.org, 18–20 September 2018. http://ceur-ws.org/Vol-2243/paper6.pdf
5. Cionini, A., D'Angelo, G., D'Emidio, M., Frigioni, D., Giannakopoulou, K., Paraskevopoulos, A., Zaroliagis, C.D.: Engineering graph-based models for dynamic timetable information systems. In: Proceedings of 14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS14) (2014)
6. Cionini, A., D'Angelo, G., D'Emidio, M., Frigioni, D., Giannakopoulou, K., Paraskevopoulos, A., Zaroliagis, C.D.: Engineering graph-based models for dynamic timetable information systems. J. Discrete Algorithms **46–47**, 40–58 (2017)
7. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. **32**(5), 1338–1355 (2003)
8. D'Angelo, G., D'Emidio, M., Frigioni, D.: Distance queries in large-scale fully dynamic complex networks. In: Mäkinen, V., Puglisi, S.J., Salmela, L. (eds.) IWOCA 2016. LNCS, vol. 9843, pp. 109–121. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44543-4_9
9. D'Angelo, G., D'Emidio, M., Frigioni, D.: Fully dynamic update of arc-flags. Networks **63**(3), 243–259 (2014). https://doi.org/10.1002/net.21542
10. D'Angelo, G., D'Emidio, M., Frigioni, D.: Fully dynamic 2-hop cover labeling. ACM J. Exp. Algorithmics **24**(1), 1.6:1–1.6:36 (2019). https://doi.org/10.1145/3299901
11. Delling, D., Dibbelt, J., Pajor, T., Werneck, R.F.: Public transit labeling. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 273–285. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20086-6_21
12. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning in road networks. Transp. Sci. **51**(2), 566–591 (2017). https://doi.org/10.1287/trsc.2014.0579
13. Delling, D., Pajor, T., Werneck, R.F.: Round-based public transit routing. Transp. Sci. **49**(3), 591–604 (2015)
14. Dibbelt, J., Pajor, T., Strasser, B., Wagner, D.: Connection scan algorithm. ACM J. Exp. Algorithmics **23**, 1.7:1–1.7:56 (2018)
15. Pyrga, E., Schulz, F., Wagner, D., Zaroliagis, C.: Efficient models for timetable information in public transportation systems. ACM J. Exp. Algorithmics **12**(2.4), 1–39 (2008)
16. Qin, Y., Sheng, Q.Z., Falkner, N.J.G., Yao, L., Parkinson, S.: Efficient computation of distance labeling for decremental updates in large dynamic graphs. World Wide Web **20**(5), 915–937 (2017)
17. Wang, S., Lin, W., Yang, Y., Xiao, X., Zhou, S.: Efficient route planning on public transportation networks: a labelling approach. In: Proceedings of 2015 ACM International Conference on Management of Data (SIGMOD 2015), pp. 967–982. ACM (2015)
18. Yano, Y., Akiba, T., Iwata, Y., Yoshida, Y.: Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In: Proceedings of 22nd ACM International Conference on Information & Knowledge Management (CIKM 2013), pp. 1601–1606. ACM (2013)
19. Zhu, A.D., Lin, W., Wang, S., Xiao, X.: Reachability queries on large dynamic graphs: a total order approach. In: Proceedings of 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD 2014), pp. 1323–1334. ACM (2014)