Efficient Route Planning on Public Transportation Networks: A Labelling Approach

Sibo Wang\textsuperscript{1} Wenqing Lin\textsuperscript{2} Yi Yang\textsuperscript{3} Xiaokui Xiao\textsuperscript{1} Shuigeng Zhou\textsuperscript{3}

\textsuperscript{1}Nanyang Technological University \textsuperscript{2} Institute for Infocomm Research, Singapore \textsuperscript{3}Fudan University

{wang0759, xkxiao}@ntu.edu.sg linw@i2r.a-star.edu.sg

\textsuperscript{3}Fudan University {yyang1, sgzhou}@fudan.edu.cn

ABSTRACT

A public transportation network can often be modeled as a timetable graph where (i) each node represents a station; and (ii) each directed edge \((u, v)\) is associated with a timetable that records the departure (resp. arrival) time of each vehicle at station \(u\) (resp. \(v\)). Several techniques have been proposed for various types of route planning on timetable graphs, e.g., retrieving the route from a node to another with the shortest travel time. These techniques, however, either provide insufficient query efficiency or incur significant space overheads.

This paper presents Timetable Labelling (TTL), an efficient indexing technique for route planning on timetable graphs. The basic idea of TTL is to associate each node \(u\) with a set of labels, each of which records the shortest travel time from \(u\) to some other node \(v\) given a certain departure time from \(u\); such labels would then be used during query processing to improve efficiency. In addition, we propose query algorithms that enable TTL to support three popular types of route planning queries, and investigate how we reduce the space consumption of TTL with advanced preprocessing and label compression methods. By conducting an extensive set of experiments on real world datasets, we demonstrate that TTL significantly outperforms the states of the art in terms of query efficiency, while incurring moderate preprocessing and space overheads.

Categories and Subject Descriptors

H.2.8 [Database Applications]: Spatial Databases and GIS

Keywords

Transportation Network; Path Queries; Algorithm

1. INTRODUCTION

Identifying fast routes in transportation networks is an important problem with applications in map services and navigation systems. This problem has been extensively studied in the past decades, yielding a plethora of indexing techniques that aim to improve query efficiency at reasonable costs of pre-computation and space. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SIGMOD’15, May 31 - June 04, 2015, Melbourne, VIC, Australia. Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.

Most of the existing techniques, however, consider private transportation, i.e., they assume that the traversal from one network location to another is carried out with a private vehicle without any service time constraint. In contrast, relatively less effort has been made for the (equally important) case of public transportation, where any traversal between different network locations relies on transportation services (e.g., buses, subways, ferries, etc.) that run on fixed routes with pre-defined schedules.

Route planning for public transportation drastically differs from that for private transportation, as the former needs to take into account additional spatiotemporal constraints that do not apply to the latter, such as the time required to wait for a bus at a station, or the feasibility of transferring from one bus to another given their respective schedules. Towards addressing the problem, a common approach is to model a public transportation network as a timetable graph [31], where (i) each node represents a station; and (ii) each directed edge \((u, v)\) is associated with a timetable that records the departure (resp. arrival) time of each vehicle at station \(u\) (resp. \(v\)). Three types of route planning queries on timetable graphs have been extensively studied:

1. Earliest Arrival Path (EAP): If we are at station \(A\) at time \(t_a\), what is the earliest time that we can arrive at station \(B\), and which is the corresponding route?

2. Latest Departure Path (LDP): If we plan to arrive at station \(B\) no later than time \(t_d\), what is the latest time that we should depart from station \(A\), and which route should we take?

3. Shortest Duration Path (SDP): If we are to depart from station \(A\) no sooner than time \(t_d\) and arrive at station \(B\) no later than time \(t_a\), which is the route that would minimize the duration of our trip from \(A\) to \(B\)?

As shown by Cooke et al. in [9], each of the above three types of queries can be solved with a modified version of the Dijkstra’s algorithm [16] that exploits the temporal information in the timetable graph \(G\). Cooke et al.’s method, however, is inefficient for the large transportation networks (with millions of temporal edges) commonly seen in practice. Therefore, numerous techniques (e.g., [6, 11, 15, 20]) are proposed to improve over Cooke et al.’s solution in terms of query efficiency. In particular, the state-of-the-art techniques either pre-arrange the temporal edges in \(G\) in a certain order [15] or augment \(G\) with information about certain important routes [20], and they accelerate query processing by utilizing the edge arrangement or the pre-computed information. Nevertheless, as we will show in our experiments, these techniques [15, 20] still incur considerable query overheads on sizable timetable graphs \(G\), since they require traversing a substantial portion of the temporal
We define the paths in \( HHL \) query processing. Nevertheless, \( TTL \) exploits the characteristics of timetable graphs to achieve significant improvements. \( G \) is shown to outperform the state of the art by up to \( k \) times larger than the optimal solution, where \( m \) is the number of edges in the timetable graph. On the practical side, we present a preprocessing algorithm motivated by our theoretical results, and show that it strikes a good balance between pre-computation cost and index quality. (Sections 5 and 6)

We improve the performance of \( TTL \) by devising several efficient techniques: (i) novel techniques for compressing the label sets of \( TTL \), which significantly reduce space consumption without substantial effects on query performance, and (ii) a simple and intuitive method to concisely represent the results of route planning queries on transportation networks, which not only is user friendly but also leads to higher query efficiency in \( TTL \). (Sections 7 and 8)

- We experimentally evaluate \( TTL \) against the states of the art, using 11 real transportation networks with up to 4 million edges. Our results demonstrate that \( TTL \) significantly outperforms competitors in terms of query efficiency, while incurring moderate preprocessing and space overheads. (Section 10)

2. PROBLEM DEFINITION

Let \( G \) be a multigraph with \( n \) nodes, such that (i) each node represents a station in a public transportation network, and (ii) each edge \( e \) from a node \( u \) to another node \( v \) is associated with a vehicle \( b \) and two timestamps \( t_d \) and \( t_a \), which indicate that there is a vehicle \( b \) that departs from \( u \) at \( t_d \) and arrives at \( v \) at \( t_a \), without any stops in between. (Note that there may exist multiple edges from \( u \) to \( v \).) For convenience, we represent \( e \) as a tuple \( (u, v, t_d, t_a, b) \). We refer to \( b \), \( t_d \), and \( t_a \) as the vehicle, departure time, and arrival time of \( e \), respectively. We define the duration of \( e \) as \( t_a - t_d \). In addition, we say that \( e \) is an outgoing edge of \( u \) and an incoming edge of \( v \). We define the out-degree (resp. in-degree) of \( v \) as the number of its outgoing (resp. incoming) edges, denoted by \( d_{out}(v) \) (resp. \( d_{in}(v) \)). We define the paths in \( G \) as follows:

**Definition 1 (Paths).** A path \( P \) in \( G \) is a sequence \( \langle e_1, e_2, \ldots, e_k \rangle \) of edges in \( G \), such that for any \( i \in [1, k-1] \), the starting node of \( e_{i+1} \) equals the ending node of \( e_i \), and the departure time of \( e_{i+1} \) is after the arrival time of \( e_i \).

We define the starting node (resp. departure time) of \( P \) as that of \( e_1 \), the ending node (resp. arrival time) of \( P \) as that of \( e_k \), and the duration of \( P \) as the difference between \( P \)'s starting and arrival time. In addition, if all edges on \( P \) have the same vehicle \( b \) (i.e., \( P \) does not contain a transfer), then we refer to \( b \) as the vehicle of \( P \); otherwise, we define the vehicle of \( P \) as \( \text{null} \).

We consider three types of route planning queries on \( G \), as defined in the following:

**Definition 2 (Earliest Arrival Path Query).** Given two nodes \( u \) and \( v \) in \( G \) and a starting timestamp \( t \), an earliest arrival path (EAP) query asks for the path with the earliest arrival time among those that (i) start from \( u \) no sooner than \( t \) and (ii) end at \( v \).

**Definition 3 (Latest Departure Path Query).** Given two nodes \( u \) and \( v \) in \( G \) and an ending timestamp \( t' \), a latest departure path (LDP) query asks for the path with the latest departure time among those that (i) start from \( u \) and (ii) end at \( v \) no later than \( t' \).

**Definition 4 (Shortest Duration Path Query).** Given two nodes \( u \) and \( v \) in \( G \), a starting timestamp \( t \), and an ending timestamp \( t' \), a shortest duration path (SDP) query asks for the path with the shortest duration those that (i) start from \( u \) no sooner than \( t \) and (ii) end at \( v \) no later than \( t' \).

For simplicity, we use the term path queries to refer to EAP, LDP, and SDP queries. We illustrate Definitions 2-4 with an example.

**Example 1.** Figure 1 shows a timetable graph \( G \) containing 6 nodes \( v_1, v_2, \ldots, v_6 \) and 14 edges. The ordered pair \( (t_d, t_a) \) on each edge \( e \) indicates that the departure (resp. arrival) time of the edge equals \( t_d \) (resp. \( t_a \)). We use solid, dashed, and dotted lines to represent the edges correspond to vehicles \( b_1, b_2 \), and \( b_3 \), respectively. Suppose that we are at \( v_2 \) at time \( t = 5 \), and plan to travel to \( v_1 \). In that case, the earliest time that we can arrive at \( v_1 \) is...
3. TIMETABLE LABELLING

This section presents timetable labelling (TTL), our index structure for route planning on timetable graphs. At a high level, a TTL index pre-computes two sets of labels, $L_{\text{in}}(v)$ and $L_{\text{out}}(v)$, for each node $v$ in $G$, such that each label in $L_{\text{in}}(v)$ (resp. $L_{\text{out}}(v)$) is a tuple concerning a "fast" path that ends at (starts from) $v$. The construction of the label sets is governed by a strict total order on the nodes in $G$, which, intuitively, defines the relative importance of each node with respect to the others. We refer to $o$ as the node order, and use $o(v)$ to denote the rank of a node $v$ in $o$. Without loss of generality, we assume that $o(v)$ is an integer in $[1, n]$, and say that $v$ ranks lower than another node $u$ if $o(v) > o(u)$. Given a timetable graph $G$ and a node order $o$, the TTL index can be uniquely constructed. Specifically, TTL is defined based on the concepts of canonical paths.

**Definition 5 (Canonical Paths).** Given a node order $o$ and two nodes $u$ and $v$ in $G$, a path $P$ from $u$ to $v$ is a **canonical path**, if it satisfies the following constraints:

1. **Dominance Constraint**: There does not exist a path from $u$ to $v$ with (i) departure time later than $P$'s and arrival time no later than that of $P$, or (ii) departure time no earlier than $P$'s and arrival time earlier than that of $P$.
2. **Rank Constraint**: Among all nodes on $P$, either $u$ or $v$ has the highest rank.

Intuitively, the dominance constraint in Definition 5 requires that a canonical path to be both the earliest arrival path and the latest departure path from $u$ to $v$ at a certain timestamp. Meanwhile, the rank constraint is crucial in proving the correctness of TTL’s query algorithm, which will be discussed in Section 4. We illustrate Definition 5 with an example.

**Example 2.** Consider the timetable graph in Figure 1. Assume that we use a node order $o(v_1) < o(v_2) < \ldots < o(v_{13})$. The path $P$ from $v_2$ to $v_{13}$ taking vehicle $b_3$ at time $t = 6$ is a canonical path, since (i) no path from $v_2$ to $v_{13}$ after time $t = 6$ can arrive at $v_{13}$ earlier than $P$ does or arrive $v_2$ before time $t = 8$ with departure time later than $P$'s, and (ii) the rank of $v_{13}$ is the highest among all nodes on $P$. In contrast, the path from $v_6$ to $v_{13}$ taking $b_3$ via $v_2$ at time $t = 7$ is not a canonical path, since the rank of $v_{13}$ is higher than those of $v_3$ and $v_4$.

In relation to canonical paths, we also introduce the concepts of intermediate nodes and pivots.

**Definition 6 (Pivots).** An **intermediate node** of a path $P$ refers to any node on $P$, except for $P$’s starting and ending nodes. The **pivot** of a canonical path is null if the path contains only two nodes; otherwise, it is the intermediate node of the path with the highest rank.

Given Definition 5 and 6, a TTL index on $G$ is uniquely determined by a node order $o$ as follows:

**Definition 7 (Timetable Labelling).** Given a node order $o$, let $L$ be the set of all canonical paths on $G$. For each $P \in L$, let $u, v, t_d, t_a, b, p$ be $P$’s starting node, ending node, departure time, arrival time, vehicle, and pivot, respectively. A **Timetable Labelling (TTL)** consists of label sets where $P$ is mapped to a label $l = (x, t_d, t_a, b, p)$ in either $L_{\text{out}}(u)$ or $L_{\text{in}}(v)$ as follows:

1. If $o(u) > o(v)$ (i.e., $u$ ranks lower than $v$), then $x = v$ and $l \in L_{\text{out}}(u)$.
2. If $o(u) < o(v)$, then $x = u$ and $l \in L_{\text{in}}(v)$.

In short, in a TTL index, the label sets $L_{\text{in}}(v)$ and $L_{\text{out}}(v)$ for a node $v$ only contain the information about the canonical paths between $v$ and the nodes that rank higher than $v$. For convenience, we refer to $L_{\text{in}}(v)$ (resp. $L_{\text{out}}(v)$) as the in-label set (resp. out-label set) of $v$, and each label in $L_{\text{in}}(v)$ (resp. $L_{\text{out}}(v)$) as an in-label (resp. out-label) of $v$. As an example, Table 2 shows the label sets of $v_2$ and $v_{13}$ in Figure 1, assuming a node order $o(v_i) = i$ for $i \in [1, 6]$.

The label sets of a TTL index have an important property: for any path query from a node $u$ to another node $v$, we can retrieve essential information about the query result from the label sets of $u$ and $v$, as shown in the following lemma.

**Lemma 1.** Let $q$ be a path query from a node $u$ to another node $v$, and $P$ be the result of $q$. Then, $L_{\text{out}}(u)$ contains a label $(w, t_d, t_a, b, p)$, and $L_{\text{in}}(v)$ contains a label $(w', t_d', t_a', b', p')$, such that one of the following conditions holds:

1. $w = v$, and $t_d, t_a, b, p$ equal $P$’s departure time, arrival time, vehicle, and pivot, respectively. That is, $L_{\text{out}}(u)$ contains a label pertinent to $P$. 

<table>
<thead>
<tr>
<th>$v$</th>
<th>$L_{\text{in}}(v)$</th>
<th>$L_{\text{out}}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_2$</td>
<td>$(v_1, 9, 10, b_2, \text{null})$</td>
<td>$(v_1, 8, 10, b_1, \text{null})$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>$(v_3, 5, 9, b_2, v_3)$</td>
<td>$(v_1, 10, 13, \text{null}, v_2)$</td>
</tr>
</tbody>
</table>

Table 2: The label sets of $v_2$ and $v_{13}$ in a TTL index for Figure 1, with a node order $o(v_i) = i$ for $i \in [1, 6]$. 

2. \( w' = u \) and \( t_d, t_a, t_v, t'_v \) equal \( P \)'s departure time, arrival time, vehicle, and pivot, respectively. That is, \( L_{in}(v) \) contains a label pertinent to \( P \).

3. \( w = w', t_a \leq t'_v \) and \( t_d (\text{resp. } t'_a) \) equals \( P \)'s departure (resp. arrival) time. That is, \( P \) contains a node \( w \), such that \( L_{out}(u) \) (resp. \( L_{in}(v) \)) contains a label pertinent to the sub-path of \( P \) from \( u \) to \( w \) (resp. \( w \) to \( v \)).

We demonstrate Lemma 1 with the following example.

**Example 3.** Consider the label sets shown in Table 2 for the timetable graph in Figure 1. For an EAP query that from \( v_4 \) to \( v_2 \) with a starting timestamp \( t = 4 \), the result is a path \( P \) via vehicle \( b_2 \) that starts from \( v_4 \) at time 5, passes through \( v_5 \) and \( v_1 \), and ends at \( v_2 \) at time 10. Observe that the out-label set \( L_{out}(v_4) \) of \( v_4 \) contains a label \( \{v_1, 5, 9, b_2, v_3\} \), which captures the sub-path of \( P \) from \( v_4 \) to \( v_1 \). Meanwhile, the in-label set \( L_{in}(v_2) \) of \( v_2 \) contains a label \( \{v_1, 9, 10, b_2, null\} \), which captures the sub-path of \( P \) from \( v_1 \) to \( v_2 \).

\[ \square \]

Lemma 1 is the basis of the query algorithms for \( TTL \). We clarify in Section 4. Meanwhile, the following lemma shows that the label sets of a \( TTL \) annotation is minimal, in the sense that if we remove any label from the label sets, we can no longer answer queries based on Lemma 2.

**Lemma 2.** For any node \( v \), if we remove any label from \( L_{in}(v) \) or \( L_{out}(v) \), then Lemma 1 no longer holds.

\[ \square \]

In subsequent sections, we first introduce \( TTL \)'s query processing procedure (in Section 4), followed by its preprocessing algorithm (in Section 5) and several optimizations in terms of query time and index space (in Section 6 and 7). This is for the ease of exposition, since the preprocessing method of \( TTL \) is considerably more sophisticated than its query algorithm.

**Remark.** \( TTL \) can be regarded as a non-trivial extension of Hierarchical Hub Labelling (HHIL) [2], which is the state-of-the-art indexing approach for answering shortest path queries on simple graphs. In particular, HHIL also constructs label sets based on a total order of nodes, and it utilizes the label sets for query processing; nevertheless, the label sets of HHIL do not contain any temporal information. Compared with HHIL, \( TTL \)'s preprocessing and query algorithms are much more sophisticated, due to the complexity induced by the spatiotemporal constraints in timetable graphs. Furthermore, \( TTL \) features several novel optimization methods (for reducing space consumption and improving query efficiency) that are unique to timetable graphs, as we clarify in Sections 7 and 8.

4. QUERY PROCESSING

This section presents the query algorithm for \( TTL \). In a nutshell, \( TTL \)'s query algorithm consists of three phases:

1. **Candidate Generation:** This phase inspects the label sets of \( TTL \) to retrieve a set of path sketches, each of which corresponds to a candidate path that could be the answer for the given query.

2. **Refinement:** This phase inspects the path sketches produced by the candidate generation phase, and then pinpoints the sketch \( S_{ana} \) corresponding to the query answer.

3. **Path Reconstruction:** This phase converts the sketch \( S_{ana} \) back to a path on \( G \), and returns the path as the query result.

In what follows, we elaborate each phase of \( TTL \)'s query algorithm, assuming that we are given an SDP query from a node \( u \) to another node \( v \), with a starting timestamp \( t \) and an ending timestamp \( t' \). We will clarify how the algorithm can be modified to handle EAP and LDP queries in Section 4.3.

4.1 Candidate Generation and Refinement

Consider the out-label set \( L_{out}(u) \) of \( u \) and the in-label set \( L_{in}(v) \) of \( v \). By Definition 7, the labels in \( L_{out}(u) \) and \( L_{in}(v) \) capture the information about a set \( \mathcal{P}_{uv} \) of paths from \( u \) to \( v \) as follows. First, if \( L_{out}(u) \) contains a label \( l_1 = \langle \{u, t'_d, t'_a, \ldots\} \rangle \) then there exists a path \( P_1 \) from \( u \) to \( v \) with departure time \( t_d \) and arrival time \( t_a \). In this case, we define the ordered pair \( \langle l_1, null \rangle \) as a sketch of \( P_1 \). Second, if \( L_{in}(v) \) contains a label \( l_2 = \{\langle u, t'_d, t'_a, \ldots\} \rangle \) then there is a path \( P_2 \) from \( u \) to \( v \) with departure time \( t'_d \) and arrival time \( t'_a \). Accordingly, we refer to \( \langle null, l_2 \rangle \) as a sketch of \( P_2 \).

Third, if \( L_{out}(u) \) and \( L_{in}(v) \) contains labels \( l_2 = \{u, t'_d, t'_a, \ldots\} \) and \( l_3 = \langle \{u, t'_d, t'_a, \ldots\} \rangle \) respectively, such that \( t'_d \leq t'_a \), then there exists a path \( P_3 \) from \( u \) to \( v \) with departure time \( t'_d \) and \( t_a \). In that case, we say that \( \langle l_3, l_4 \rangle \) is a sketch of \( P_3 \). In summary, based on \( L_{out}(u) \) and \( L_{in}(v) \), we can construct the sketches for a set \( \mathcal{P}_{uv} \) of paths from \( u \) to \( v \), and each sketch records the departure and arrival time of the corresponding path.

By Lemma 1, \( \mathcal{P}_{uv} \) must contain the answer \( P_{ana} \) to an SDP query from \( u \) to \( v \) with a starting timestamp \( t \) and an ending timestamp \( t' \). If we are given the sketches of all paths in \( \mathcal{P}_{uv} \), then we can easily identify one corresponding to \( P_{ana} \), by searching for the sketch with the shortest duration among those with departure time no sooner than \( t \) and arrival time no later than \( t' \). However, constructing sketches for all paths in \( \mathcal{P}_{uv} \) is costly, since \( |\mathcal{P}_{uv}| = |L_{out}(u)| \cdot |L_{in}(v)| \) in the worst case. (This worst case occurs when each label in \( L_{out}(u) \) can be combined with any label in \( L_{in}(v) \) to form a path sketch.) Nevertheless, many of the sketches could have been omitted without affecting the correctness of the query result. For example, consider the label sets \( L_{out}(u) \) and \( L_{in}(v) \). Observe that \( L_{out}(u) \) contains a label \( l_1 = \{v_1, 5, 9, b_2, v_3\} \), which can form two path sketches when combined with the labels \( l_2 = \{v_1, 9, 10, b_2, null\} \) and \( l_3 = \langle v_1, 10, 11, b_2, null \rangle \) in \( L_{in}(v) \). However, the sketch \( \langle l_1, l_2 \rangle \) is dominated by the sketch \( \langle l_1, l_3 \rangle \), in the sense that the \( l_1, l_2 \) corresponds to a path that departs from \( u \) at time 5 and arrives at \( v \) at time 10, which is strictly better than the path corresponding to \( \langle l_1, l_3 \rangle \), as the latter has departure time 5 and arrival time 11. In other words, the path pertinent to \( \langle l_1, l_3 \rangle \) can never be the result for any path query, and hence, we can omit \( \langle l_1, l_3 \rangle \) during query processing.

In general, when \( TTL \) inspects label sets in its candidate generation phase, it only constructs path sketches that are not dominated by others, so as to improve query efficiency. To facilitate this, the preprocessing algorithm of \( TTL \) sorts the labels \( l \) in each label set in ascending order of \( f(l) \), where \( f(l) \) is a total order of labels as follows: for any two labels \( l_1 = \langle v, t_d, t_a, \ldots \rangle \) and \( l_2 = \langle v', t'_d, t'_a, \ldots \rangle \) in the same label set, we have \( f(l_1) < f(l_2) \) if and only if

1. \( o(v) < o(v') \) i.e., \( v \) ranks higher than \( v' \), or
2. \( o(v) = o(v') \) and \( t_d < t'_d \), or
3. \( o(v) = o(v') \), \( t_d = t'_d \), and \( t_a < t'_a \).

With the above ordering of labels, \( TTL \) generates path sketches in \( O(|L_{out}(u)| + |L_{in}(v)|) \) time using the SketchGen method in Algorithm 1. The input to the algorithm includes \( L_{out}(u) \) and \( L_{in}(v) \), as well as the SDP query’s starting timestamp \( t \) and ending timestamp \( t' \); its output is a set \( S_{ana} \) of path sketches corresponding to the candidate answers for the query. The basic idea of the algorithm is to generate path sketches with two concurrent linear scans on \( L_{out}(u) \) and \( L_{in}(v) \), respectively.
Algorithm 1: SketchGen

input: $L_{\text{out}}(u), L_{\text{in}}(v), t, t'$
output: a set $S_{uv}$ of path sketches
1. initialize $i = 1$, $j = 1$, and $C = 0$;
2. while $i \leq L_{\text{out}}(u)$ or $j \leq L_{\text{in}}(v)$ do
3. let $l_u = (w, t_u, l_1, \ldots, i)$ be the $i$-th label of $L_{\text{out}}(u)$;
4. if $[t_u, t_u'] \subseteq [t, t']$ then
5. increase $i$ by one, and then goto Line 4;
6. else if $w = v$ then
7. insert $(l_u, \text{null})$ into $S_{uv}$;
8. increase $i$ by one, and then goto Line 4;
9. let $l_v = (w', t_v, l_1', \ldots, j)$ be the $j$-th tuple of $L_{\text{in}}(v)$;
10. if $[t_v, t_v'] \subseteq [t, t']$ then
11. increase $j$ by one, and then goto Line 4;
12. else if $w' = w$ then
13. insert $(\text{null}, l_v)$ into $S_{uv}$;
14. increase $j$ by one, and then goto Line 4;
15. else if $w' = w$ then
16. insert $(l_u, l_v)$ into $S_{uv}$;
17. increase $i$ by one;
18. else
19. increase $j$ by one;
20. if $o(w) < o(w')$ then
21. increase $i$ by one;
22. else
23. increase $j$ by one;
24. return $S_{uv}$;

Specifically, SketchGen first scans the labels in $L_{\text{out}}(u)$, until it finds a label $l_u = (w; t_u, l_1, \ldots, i)$ with $t_u \geq t$ and $l_u \leq t'$ (Lines 1-5). If $w = v$, then SketchGen inserts $l_u$ into $S_{uv}$ as a path sketch, since $l_u$ captures a candidate path from $u$ to $v$ with departure time $t_u$ and arrival time $t_u'$; after that, SketchGen moves on to the next label in $L_{\text{out}}(u)$ (Lines 6-8).

On the other hand, if $w \neq v$, then SketchGen turns its attention to $L_{\text{in}}(v)$, and scans the labels in $L_{\text{in}}(v)$ until it encounters a label $l_v = (w', t_v, l_1', \ldots, j)$ with $t_v \geq t$ and $l_v \leq t'$ (Lines 9-11). If $w' = u$, then SketchGen adds $l_v$ to $S_{uv}$ as a path sketch, and moves on to the next label in $L_{\text{in}}(v)$ (Lines 12-14).

Meanwhile, if $w' \neq u$, then SketchGen considers whether $l_u$ and $l_v$ can be combined to form a path sketch $(l_u, l_v)$. In particular, if $w = w'$ and $l_u \leq l_v$, then SketchGen inserts $(l_u, l_v)$ into $S_{uv}$, and move on to the next label in $L_{\text{out}}(u)$ (Lines 15-18). However, if $w \neq w'$ or $w > t_u$, then SketchGen would move on to the next label in either $L_{\text{out}}(u)$ or $L_{\text{in}}(v)$ to continue the construction of path sketches. Here, SketchGen differentiates two cases.

Case 1: $w = w'$ but $t_u > t_v$. In this case, SketchGen proceeds to the next label in $L_{\text{in}}(v)$ (Lines 19-20). The rationale is that, due to the way that we sort the labels in $L_{\text{in}}(v)$, the next label in $L_{\text{in}}(v)$ could be $l_v = (w', t_v, l_1', \ldots, j)$, in which case $l_u$ and $l_v$ may form a path sketch.

Case 2: $w \neq w'$. In this case, SketchGen checks whether $o(w) < o(w')$. If $o(w) < o(w')$ holds, then SketchGen moves on to the next label in $L_{\text{out}}(u)$ (Lines 21-23), since the label could be $l_u = (w, t_u, l_1, \ldots, i)$, in which case $l_u$ and $l_v$ may form a path sketch. On the other hand, if $o(w) > o(w')$, then the next label in $L_{\text{out}}(u)$ cannot be combined with $l_v$ as a path sketch, and hence, SketchGen would proceed to the next label in $L_{\text{in}}(v)$ instead.

In general, our discussions above are applicable for any pair of labels $l_u \in L_{\text{out}}(u)$ and $l_v \in L_{\text{in}}(v)$ encountered by SketchGen during its linear scans of $L_{\text{out}}(u)$ and $L_{\text{in}}(v)$. In particular, any labels with starting time before $t$ or ending time after $t'$ are omitted (Lines 4-5 and 10-11), while any remaining $l_u \in L_{\text{out}}(u)$ that concerns $v$ and any remaining $l_v \in L_{\text{in}}(v)$ that concerns $u$ are inserted into $S_{uv}$, as path sketches (Lines 6-8 and 12-14). For the other labels, SketchGen carefully decides (i) whether a pair of labels can form a path sketch, and (ii) whether it should proceed to the next label in $L_{\text{out}}(u)$ or $L_{\text{in}}(v)$ (Lines 15-25).

It can be verified that SketchGen runs in $O(|L_{\text{out}}(u)| + |L_{\text{in}}(v)|)$ time and generates at most $O(|L_{\text{out}}(u)| + |L_{\text{in}}(v)|)$ path sketches in $S_{uv}$, since it linearly scans $L_{\text{out}}(u)$ and $L_{\text{in}}(v)$. In addition, $S_{uv}$ always contains a sketch that corresponds to the answer of the given SDP query, as shown in Lemma 3. The rationale is that, due to the order in which TTL sorts the labels in each label set, the linear scans employed by SketchGen only omit the label pairs that either (i) cannot form path sketches or (ii) correspond to paths that are dominated by others.

Lemma 3. Let $S_{uv}$ be the set of sketches that SketchGen returns for an SDP query, and $P_{ans}$ be the answer of the query. Then, one of the sketches in $S_{uv}$ corresponds to $P_{ans}$. □

Once $S_{uv}$ is obtained, it is straightforward to identify the sketch in $S_{uv}$ that is pertinent to the SDP query $P_{ans}$. In particular, for each sketch in $S_{uv}$, we can calculate the duration of the corresponding path based on the departure and arrival time recorded in the sketch; based on that, we can pinpoint the sketch with the shortest duration. In Section 4.2, we will explain how we can convert this sketch back into $P_{ans}$.

4.2 Path Reconstruction

Let $s_{ans}$ be the sketch in $S_{uv}$ that corresponds to the result $P_{ans}$ of the given SDP query. Then, $s_{ans} = (l_u, l_v)$, where $l_u$ (resp. $l_v$) is either null or a label in $L_{\text{out}}(u)$ (resp. $L_{\text{in}}(v)$). To transform $s_{ans}$ back to $P_{ans}$, it suffices to convert $l_u \neq \text{null}$ (resp. $l_v \neq \text{null}$) to the canonical path $P_u$ (resp. $P_v$) in $G$ that correspond to $l_u$ (resp. $l_v$). Such conversions can be performed by exploiting the pivot of $P_u$ (resp. $P_v$), which is recorded in $l_u$ (resp. $l_v$). To explain, we first present a lemma about pivots.

Lemma 4. Let $P$ be a canonical path from $u$ to $v$, and $p$ be the pivot of $P$. Then, the sub-path of $P$ from $u$ to $p$ (resp. from $p$ to $v$) is also a canonical path. □

Suppose that the pivot of $P_u$ is a node $p$, and $l_u = (w, t_u, l_1, \ldots, i)$. Let $P_1$ (resp. $P_2$) be the sub-path of $P$ from $u$ to $p$ (resp. from $p$ to $v$). By Lemma 4, both $P_1$ and $P_2$ are canonical paths. Then, by Definition 7, there should exist two labels $l_1$ and $l_2$ (in TTL) that correspond to $P_1$ and $P_2$, respectively. In other words, we can "unfold" $l_u$ into a sequence of labels $S = (l_1, l_2)$, such that the canonical path pertinent to $l_u$ equals a concatenation of the canonical paths corresponding to $l_1$ and $l_2$. We refer to $l_1$ (resp. $l_2$) as the left (resp. right) child of $l_u$.

In general, the above unfolding method can be applied on any label $(\cdot, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot)$ where $p \neq \text{null}$ (i.e., the label corresponds to a canonical path that contains at least two edges). Therefore, given the label sequence $S = (l_1, l_2)$ mentioned above, we can recursively modify $S$ by replacing each label with its children, until every label in $S$ has a null pivot. In that case, each label in $S$ represents an edge in $G$. Then, by replacing each label in $S$ with its corresponding edge, we can transform $S$ into the canonical path $P_u$ pertinent to $l_u$. Using the same method, we can unfold $l_v$ into its corresponding canonical path $P_v$, after which we can concatenate $P_u$ and $P_v$ to obtain the answer $P_{ans}$ of the given SDP query.
Algorithm 2: PathUnfold

\[ \text{input: a label } l = \langle v_1, \ldots, v_p \rangle \text{ in } \mathcal{TTL} \]
\[ \text{output: the canonical path corresponding to } l \]

1. if \( p = \text{null} \) then
2. \quad return a path that only contains the edge in \( G \) corresponding to \( l \)
3. else
4. \quad \( l_1 = \text{the left child of } l \)
5. \quad \( l_2 = \text{the right child of } l \)
6. \quad \( P_1 = \text{PathUnfold}(l_1) \)
7. \quad \( P_2 = \text{PathUnfold}(l_2) \)
8. \quad set \( P = \text{a concatenation of } P_1 \text{ and } P_2 \), with \( P_1 \text{ proceeding } P_2 \)
9. return \( P \)

Algorithm 2 shows the pseudo-code of the above unfolding approach, which takes as input a label \( l \) and returns the canonical path \( P \) corresponding to \( l \). The most crucial step of the algorithm is the identification of \( l \)'s children. To ensure the efficiency of this step, \( \mathcal{TTL} \) pre-computes two pointers from each label to its left and right children, respectively, so that its takes \( O(1) \) time to pinpoint the children of any label. As such, the running time of Algorithm 2 is linear to the number of nodes on \( P \).

**Example 4.** Consider an SDP query from \( v_3 \) to \( v_4 \) in Figure 1, with starting time \( t = 8 \) and ending time \( t' = 13 \). By a linear scan of the labels in \( L_{\text{out}}(v_2) \) and \( L_{\text{in}}(v_4) \) (in Table 2), we obtain two path sketches: (i) \( \langle l_1, l_2 \rangle \), with \( l_1 = \langle v_1, 8, 10, b_1, \text{null} \rangle \) in \( L_{\text{out}}(v_2) \) and the label \( l_2 = \langle v_1, 10, 13, \text{null}, v_2 \rangle \) in \( L_{\text{in}}(v_4) \), (ii) \( \langle \text{null}, l_3 \rangle \), with \( l_3 = \langle v_2, 11, 13, b_2, v_6 \rangle \) in \( L_{\text{in}}(v_4) \). Then, we set \( \text{sa}_{\text{ans}} = \langle \text{null}, l_3 \rangle \), since \( \langle \text{null}, l_3 \rangle \) has a shorter duration than \( \langle l_1, l_2 \rangle \). After that, we unfold \( l_3 \) by recursively reconstructing the SDPs from \( v_2 \) to \( v_6 \) and \( v_6 \) to \( v_4 \). In the end, we obtain the path \( P_{\text{ans}} = \langle (v_2, v_6, 11, 13, b_2), (v_6, v_4, 12, 13, b_2) \rangle \).

The following corollary shows the correctness of Algorithm 2.

**Corollary 1.** Given a label \( l \) in \( \mathcal{TTL} \), Algorithm 2 returns the canonical path corresponding to \( l \).

### 4.3 Summary and Query Extensions

Summing up the discussions in Sections 4.1 and 4.2, \( \mathcal{TTL} \) answers any SDP query from \( u \) to \( v \) in \( O(\mid L_{\text{out}}(u) \mid + \mid L_{\text{in}}(v) \mid + k) \) time, where \( k \) is the number of nodes on the query result. Furthermore, \( \mathcal{TTL} \)'s query algorithm for SDP queries can be easily extended to support EAP and LDP queries. In particular, the extension is based on the following lemma about sketchGen.

**Lemma 5.** Let \( L_{\text{out}}(u) \), \( L_{\text{in}}(v) \), \( t \), and \( t' \) be the input to sketchGen (as in Algorithm 1), and \( S_{uv} \) be its output. When \( t' = +\infty \), one of the sketches in \( S_{uv} \) corresponds to the answer of an EAP query from \( u \) to \( v \) with starting time \( t \). On the other hand, if \( t = -\infty \), then \( S_{uv} \) contains a sketch corresponding to the answer of an LDP query from \( u \) to \( v \) with ending time \( t' \).

By Lemma 5, if we are to answer an EAP query from \( u \) to \( v \) with starting time \( t \), we can first invoke sketchGen, setting \( t' = +\infty \). Then, we examine the path sketches returned by sketchGen to identify the one \( S_{uv} \) with the earliest arrival time. After that, we unfold \( S_{uv} \) into a path using Algorithm 2, and return the path as the query result.

On the other hand, if we are given an LDP query from \( u \) to \( v \) with ending time \( t \), we can apply sketchGen with \( t = -\infty \), and obtain a set of sketches \( S_{uv} \). After that, we inspect \( S_{uv} \) to pinpoint the sketch \( S_{uv} \) with the latest departure time. Finally, we use Algorithm 2 to convert \( S_{uv} \) into the query answer. It can be verified that the above query algorithms for EAP and LDP queries have the same time complexity as the one for SDP queries.

### 5. INDEX CONSTRUCTION

This section presents the algorithms for constructing \( \mathcal{TTL} \) indices, assuming that the node order \( o \) is given. (We will discuss the choice of node order in Section 6.) By Definition 7, each label in \( \mathcal{TTL} \) corresponds to a canonical path in \( G \), and vice versa. Therefore, if we are to build a \( \mathcal{TTL} \) index \( \mathcal{L} \) based on a node order \( o \), then conceptually, we should identify all canonical paths in \( G \), and convert each of them into a label in \( \mathcal{L} \). But how can we derive all canonical paths in \( G \)? A straightforward approach is to run the temporal Dijkstra’s algorithm to retrieve all temporal shortest paths in \( G \), and then examine the paths obtained to identify the canonical paths. This approach, however, incurs prohibitive preprocessing costs due to the large number of temporal shortest paths that it retrieves. To address this issue, we will first present an algorithm for computing paths in \( G \) that satisfy the Dominance Constraint (referred to as the non-dominated paths) in Section 5.1. Then, we explain how the algorithm can be used to avoid computing the paths that violate Rank Constraint, which allows us to construct a \( \mathcal{TTL} \) index \( \mathcal{L} \) in a more efficient manner. Note that, even though our index construction algorithm has the same worst-case complexity as the straightforward approach, the former’s practical efficiency is significantly higher than that of the latter, as we show in Appendix D.2.

#### 5.1 Computing Non-Dominated Paths

Let \( u \) be a node in \( G \), and \( T_{\mathcal{U}} \) be a set that contains the departure timestamps of all outgoing edges of \( u \). Suppose that we are to compute all non-dominated paths that start from a node \( u \) in \( G \). We first make an interesting observation as follows.

**Observation 1.** Let \( t_u \) be the largest timestamp in \( T_{\mathcal{U}} \). Then, any earliest arrival path (EAP) \( P \) from \( u \) to a node \( v \) with departure time \( t_v \) is a non-dominated path.

Based on Observation 1, we can compute all non-dominated paths that start from \( u \) at time \( t_u \), by deriving all EAPs from \( u \) at time \( t_u \). This derivation can be performed with a modified version of Dijkstra’s algorithm [9], referred to as temporal Dijkstra’s algorithm. First, we create a hash table \( A \) that maps each node \( v \) to the earliest arrival time (EAT) \( t_v \) from \( u \) to \( v \). We initialize \( t_v = +\infty \) for any \( v \neq u \), and \( t_v = 0 \) otherwise. Then, we collect all outgoing edges of \( u \) with departure time \( t_u \), and insert them into a min-heap \( H \) that sorts edges in ascending order of their arrival timestamps. After that, we iteratively remove the top edge \( v = (x, y, t_x, t_y, t_x) \) in \( H \), and process it as follows. We first retrieve the current EAT of \( x \) and \( y \) (denoted as \( t_x \) and \( t_y \), respectively) from the hash table \( A \). If \( t_y \geq t_u \) and \( t_x < t_u \), we set \( t_y = t_u \), and insert into \( H \) all outgoing edges of \( y \) with departure time no sooner than \( t_u \). Finally, we proceed to pop the next edge in \( H \). When \( H \) becomes empty, we terminate the algorithm. Then, by Observation 1, each final EAT \( t_v \) recorded in \( H \) corresponds to a non-dominated path (from \( u \) to \( v \) with starting time \( t_u \)).

Now suppose that we apply the temporal Dijkstra’s algorithm to compute all EAPs that start from \( u \) at time \( t_u \), where \( t_u \) denotes the second largest timestamp in \( T_{\mathcal{U}} \). We have another observation as follows.

**Observation 2.** Any EAP \( P \) from \( u \) with departure time \( t_u \) is a non-dominated path, if it is not dominated by any EAP from \( u \) with departure time \( t_v \).

Given any EAP from \( u \) to \( v \) with departure time \( t_u \), we can decide whether it is a non-dominated path, by comparing its arrival time with that of the EAP from \( u \) to \( v \) with departure time \( t_v \). In general, an EAP from \( u \) with departure time \( t \) is a non-dominated path, if it is not dominated by any EAP from \( u \) with departure time \( t'' > t \). Therefore, if we are to compute all non-dominated paths
from \( u \), we can examine the timestamps in \( T_d \) in descending order, and apply the modified Dijkstra’s algorithm once for each timestamp \( t \in T_d \), to identify the non-dominated EAPs from \( u \) with departure \( t \). The following lemma shows that our approach does not miss any non-dominated path that starts from \( u \).

**Lemma 6.** For any non-dominated path \( P \) starting from \( u \), there exists a timestamp \( t \in T_d \), such that \( P \) is an EAP starting from \( u \) at time \( t \).

Finally, let us consider a “backward” version of the temporal Dijkstra’s algorithm, such that (i) the min-heap \( H \) sorts edges based on arrival time, (ii) the hash table \( A \) records the latest departure time of each node, and (iii) we start by inserting into \( H \) all incoming edges of \( u \) with a given arrival time \( t' \). It can be verified that such an algorithm computes the latest departure paths (LDP) that arrive at \( u \) at time \( t' \). In addition, we can observe that an LDP to \( v \) with arrival time \( t' \) is non-dominated, if it is not dominated by any LDP to \( v \) with arrival time \( t' < t' \). Furthermore, we have the following lemma.

**Lemma 7.** Let \( T_a \) be a set that contains the arrival timestamps of all incoming edges of \( u \). For any non-dominated path \( P \) ending at \( u \), there exists a timestamp \( t \in T_a \), such that \( P \) is an LDP ending at \( u \) at time \( t \).

Therefore, we can compute all non-dominated paths ending at \( u \), by enumerating the timestamps in \( T_a \) in ascending order, and by applying the reverse version of our Dijkstra’s algorithm once for each timestamp. In Section 5.2, we will utilize our modified version of the Dijkstra’s algorithm to identify canonical paths and construct label sets.

### 5.2 Construction of Label Sets

Algorithm 3 shows the preprocessing method of TTL, referred to as *IndexBuild*. Given \( G \) and a node order \( o \), *IndexBuild* first identifies the node \( u_1 \) with the highest rank, as well as the latest departure time \( t_d \) among the outgoing edges of \( u_1 \) (Lines 1-4). After that, it invokes the temporal Dijkstra’s algorithm to compute the EAPs from \( u_1 \) with departure \( t_d \) (Lines 5-28), with some simple bookkeeping to keep track of the vehicle \( b \) of each EAP \( P \) (Lines 17-20 and 23-24), as well as the intermediate node \( p \) on \( P \) with the highest rank (Lines 25-28). For each EAP \( P \) ending at a node \( v \) at time \( t_a \), *IndexBuild* inserts a label \((u_1, t_d, t_a, b, p)\) into \( L(v) \) (Lines 30-34). The reason is as follows. First, based on our discussions in Section 5.1, \( P \) satisfies the Dominance Constraint in Definition 5. Second, given that \( u_1 \) has a higher rank than all other nodes, \( P \) also satisfies the Rank Constraint in Definition 5. Therefore, \( P \) is a canonical path, and hence, we add into \( L(v) \) a label corresponding to \( P \) (see Definition 7).

After that, *IndexBuild* examines the other departure time of \( u_1 \)’s outgoing edges in descending order (Line 5). For each departure time \( t_d \), *IndexBuild* derives the EAPs starting from \( u_1 \) at time \( t_d \), using the temporal Dijkstra’s algorithm (6-8). For each of the EAP \( P \) derived, if \( P \) ends at a node \( v \) at time \( t_a \), *IndexBuild* examines whether the current \( L(v) \) contains a label \( \ell = (u_1, \ldots, t_d, \ldots, \cdot) \) with \( t_a \leq t_a \) (Line 31). If such a label exists, then \( P \) is dominated by an EAP that *IndexBuild* previously derived, in which case \( P \) will be omitted; otherwise, \( P \) must be a canonical path, and hence, *IndexBuild* inserts into \( L(v) \) a label corresponding to \( P \).

Once all departure timestamps from \( u_1 \) are enumerated, *IndexBuild* employs the reverse version of the temporal Dijkstra’s algorithm to derive all canonical paths ending at \( u_1 \) (Line 34). In particular, *IndexBuild* examines the arrival timestamps at \( v \) in ascending order and, for each timestamp \( t_a \), uses the backward Dijkstra’s algorithm to compute the LDPs that end at \( u_1 \) at time \( t_a \). Based on the LDPs computed, *IndexBuild* identifies all canonical paths ending at \( u_1 \), and constructs labels in other nodes’ label sets, in a manner similar to the processing of canonical paths starting from \( u \). After that, *IndexBuild* removes \( u_1 \) from \( G \) (Line 35), and proceeds to process the remaining nodes.

Let \( u_i \) denote the node with \( o(u_i) = i \) (\( i \in [2, n] \)). The subsequent execution of *IndexBuild* inspects \( u_i \) in ascending order of \( i \). For each \( u_i \), *IndexBuild* examines a modified version of \( G \) (denoted as \( G_i \)) where all nodes with higher rank than \( u_i \) are removed. In particular, *IndexBuild* computes the all non-dominated paths in \( G_i \) that start from \( u_i \), using the same approach as in the case of \( u_1 \) (Lines 5-28). The set \( S \) of non-dominated paths thus derived would satisfy the Rank Constraint in Definition 5, since they do not contain any node that ranks higher than \( u_i \). However, as they...
are derived on $G$, instead of $G$, they may not all be non-dominated paths in $G$. To identify the paths in $\hat{S}$ that are non-dominated in $G$, IndexBuild processes each path $P$ in $\hat{S}$ as follows. Assume that $P$ starts from $u_i$ at time $t_a$ and ends at a node $v$ at time $t_b$. If $L_{out}(u_i)$ contains a label $l_1 = (\tau_j, t_j, u_i, \ldots)$ and $L_{in}(v)$ has a label $l_2 = (\tau_k, t_k, v, \ldots)$ such that $l_2 < l_1$ and $\{t_2 \ldots \tau_k\} \subseteq [t_a, t_b]$, then $P$ is dominated by a path in $G$ that goes from $u_i$ to $v$ via $u_j$.

(Notes that $j < i$ always holds, due to the order in which IndexBuild processes nodes.) In that case, IndexBuild omits $P$ (Line 32). On the other hand, if such labels $l_1$ and $l_2$ do not exist, then $P$ is not dominated by any path in $G$ that passes through a node $u_k$ with $k < i$. In that case, $P$ must be a non-dominated path in $G$. Then, given that $P$ satisfies the Rank Constraint in Definition 5, it is a canonical path on $G$; as such, IndexBuild would insert into $L_{in}(v)$ a label pertinent to $P$ (Line 33). In a similar fashion, IndexBuild would also apply the backward Dijkstra’s algorithm to compute all canonical paths that ends at $u_i$, and create labels in the out-label sets accordingly. Interested readers are referred to Appendix C for an example that demonstrates the execution of IndexBuild.

Once all $u_i$ are processed, IndexBuild returns the collection of all label sets constructed as a TTL index $L$. The following Lemma 8 shows the correctness and time complexity of IndexBuild.

**Lemma 8.** Given $G$ and a node order $\pi$, IndexBuild returns a TTL index $L$. The following Lemma 8 shows the correctness and time complexity of IndexBuild.

**Lemmas 9.** $|I(u)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.

**Lemma 9.** $|L(u)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.

**Lemma 9.** $|I(u_i)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.

**Lemma 9.** $|I(u_i)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.

**Lemma 9.** $|I(u_i)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.

**Lemma 9.** $|I(u_i)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.

**Lemma 9.** $|I(u_i)|$ is a lower-bound of $|N(v)|$, which is the number of nodes that are selected before $u_i$, and aim to maximize $u_i$’s residual coverage with respect to the previously selected nodes.

To analyze the theoretical guarantee of our method, we first define the influence set of $u_i$ (denoted as $I(u_i)$) as the set of non-dominated paths in $R_i(u_i)$ that either start from $u_i$ or end at $u_i$. Observe that each path $P$ in $I(u_i)$ is a canonical path with respect to our node order. Furthermore, if we construct a TTL based on our node order, then $P$ would be mapped into a label $\langle u_i, \ldots, \pi \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to $u_i$, and we have $|L| = \sum_{i=1}^{m} |I(u_i)|$.
6.2 Heuristic Approach

As shown in Theorem 1, the approximation algorithm in Section 6.1 incurs tremendous overheads, which renders it inapplicable in sizable graphs. To remedy this issue, we devise a heuristic approach that has the same spirit of the approximation algorithm, but entails much smaller computation costs. First, we randomly sample a number of edges from $G$. After that, for each edge $e = (u, v)$, we employ the temporal Dijkstra’s algorithm to identify all EAPs starting from $u$ at time $t_u$. Notice that all these EAPs form a tree $T$ rooted at $u$, such that the directed path from the root to any leaf is an EAP from $u$. Furthermore, for any node $v$ in $T$, the number of nodes in the subtree under $v$ (including $v$ itself) equals the number of EAPs from $u$ that are covered by $v$. We refer to this number as the coverage of $v$ in $T$, and refer to $T$ as the EAP tree from $u$ at time $t_u$. Note that the coverage of all nodes in $T$ can be computed in $O(n)$ time using a bottom-up scan of $T$.

Let $T$ denote the set of all EAP trees that we have sampled, and $u_1, u_2, \ldots, u_n$ denote the node order that we chose. To decide our node order, we maintain an array that records, for each node $v$, the sum of the coverage of $v$ in all trees in $T$. We refer to this sum as the coverage sum $u \in T$. As the approximation algorithm in Section 7.1, followed by a pivot-based compression method in Section 7.2.

7. LABEL COMPRESSION

Even with a good node order, if a TTL index $L$ may still have a sizable number of labels and incur considerable space overheads. To reduce the space consumption of $L$, we propose to compress the labels in $L$ by exploiting their similarities. In particular, we present a route-based compression approach in Section 7.1, followed by a pivot-based compression method in Section 7.2.

7.1 Route-Based Compression

Consider the example in Figure 2a, where we have three vehicles $b_1, b_2,$ and $b_3$ serving the same route $r_1 = (v_1, v_2, v_3)$, i.e., they all start from node $v_1$, pass by $v_2$, and then arrive at $v_3$. Figure 2b shows the label sets of $v_1$, assuming that $\alpha(v_1) < \alpha(v_2) < \alpha(v_3)$.

Observe that the labels in $L_{in}(v_3)$ are rather redundant, as they repeatedly record the arrival time of $b_1, b_2$, and $b_3$. To eliminate such redundancy, we propose to replace the labels in $L_{in}(v_3)$ with two labels $(v_1, null, null, r_1, v_2)$ and $(v_2, null, null, r_1, null)$, as shown in Figure 2c. This is, we combine the labels in $L_{in}(v_3)$ that share the same starting node, ending node, and pivot. To ensure lossless compression, we create a time table that records, for $v_1$, $v_2$, and $v_3$, the departure and arrival time of each bus serving route $r_1$, as shown in Figure 2d. During query processing, if we encounter the label $(v_1, null, null, r_1, v_2)$ in $L_{in}(v_3)$, then we inspect the timetable entries associated with $v_1$ and $v_3$ to reconstruct three original labels concerning $b_1, b_2$, and $b_3$. The label $(v_2, null, null, r_1, null)$ can be decompressed in the same way.

In general, in the in-label set $L_{in}(v)$ of a node $v$, if all labels from a node $u$ are associated with vehicles serving the same route $r$ (hence the same pivot $p$), then we compress the labels into a single label $(u, null, null, r, p)$, and create the timetables of $r$ associated with $u$ and $v$. The out-label sets in $L$ can be compressed in the same manner. We observe that this route-based compression approach is particularly effective on a path that is only served by buses from one route, as in the case of Figure 2a.

7.2 Pivot-Based Compression

The route-based compression approach is applicable on a label $l$, only if the vehicle of $l$ is not null, i.e., the canonical path corresponding to $l$ is carried out by one vehicle without transfer. Nonetheless, we may still compress labels pertinent to paths with vehicle transfers, by exploiting the pivots recorded in the labels. For example, consider an in-label set $L_{in}(v)$ of a node $v$, where there are three labels from a node $u$: $l_1 = (u, t_a, t_b, null, p)$, $l_2 = (u, t_a, t_b, null, p)$, and $l_3 = (u, t_a, t_b, null, p)$. Although the vehicles of $l_1$, $l_2$, and $l_3$ equal $null$, they all have the same pivot $p$, i.e., the canonical paths corresponding to $l_1$, $l_2$, and $l_3$ all pass through $p$. In that case, we compress $l_1$, $l_2$, and $l_3$ into a single label $l = (u, null, null, p)$. After that, whenever we encounter $l$, during query processing, we decompress it by reconstructing $l_1$, $l_2$, and $l_3$ using their children (see Section 4.2). For instance, suppose that $\alpha(u) < \alpha(p) < \alpha(o)$. Then, $L_{in}(p)$ should contain three labels $(u, t_a, t_b, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot), (u, t_a, t_b, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot), (u, t_a, t_b, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot)$, which are the left children of $l_1$, $l_2$, and $l_3$, respectively. In addition, $L_{in}(v)$ should have three labels $(p, t_a, t_b, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot), (p, t_a, t_b, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot), (p, t_a, t_b, \cdot, \cdot, \cdot, \cdot, \cdot, \cdot)$ that are the right children of $l_1$, $l_2$, and $l_3$, respectively. Therefore, we can easily reconstruct $l_1$, $l_2$, and $l_3$, by inspecting $L_{in}(o)$ and $L_{in}(v)$, based on the pivot information in $l$.

In general, in an in-label (resp. out-label) set, if all in-labels from (resp. out-labels to) a node $u$ have the same pivot $p$, then we may replace them with a compressed label $(u, null, null, null, p)$. There is one caveat, though: if a label $l$ and its left (resp. right) child are both compressed, then the reconstruction of $l$ would also require reconstruction of its left (resp. right) child. Such recursive reconstruction incurs considerable overheads, which degrades query efficiency on compressed label sets. To avoid such recursions, we impose a constraint in label compression: if a label is compressed, then neither of its children should be compressed. An
immediate question is, how should we choose the labels to compress, so as to minimize space consumption without violating the compression constraint?

To answer the above question, we construct a dependency graph as follows. First, in the in-label set $L_{in}(v)$ (resp. out-label set $L_{out}(v)$) of a node $v$, if we may generate a compressed label $(u, null, null, null, p)$, then we create a tuple $(u, v, p)$ (resp. $(v, u, p)$). After that, we map each tuple into a node, and draw an undirected edge between any two nodes with a parent-child relationship, e.g., $(u, v, p)$ and $(u, p, \cdot)$. In addition, we assign to each node $(u, v, p)$ a weight that equals $c - 1$, where $c$ equals the number of labels compressed into $(u, v, p)$. In other words, $c$ captures the reduction of space due to $(u, v, p)$. We refer to the resulting graph as a dependency graph $D$.

Observe that, if we are to create a set $S$ of compressed labels under our compression constraints, then the labels in $S$ correspond to an independent set on $D$. Accordingly, we can maximize the effect of compression by identifying the maximum-weight independent set on $D$, which is NP-hard. We address the problem by adopting an approximation algorithm [28] for maximum-weight independent set, which runs in $O(n^2 \log n + m^\alpha)$ time, where $n$ and $m$ are the numbers of nodes and edges in $D$, respectively.

Finally, we note that the above pivot-based compression method can be combined with the route-based compression approach in Section 7.1. Specifically, we first apply route-based compression on the label sets, and then, employ pivot-based compression but exclude those labels that have been compressed using the route-based approach. As such, we benefit from both types of compression. We refer readers to Appendix B for a more detailed description of TTL's query algorithm under a compressed index structure.

8. EXTENSIONS

Concise Representation of Query Results. Recall that our query algorithms always return a path $P$ represented as a sequence of edges $(e_1, e_2, ..., e_k)$ in $G$. Although such an edge sequence provides very detailed information about $P$, users in practice may prefer a more concise representation of $P$ as the query result. For example, suppose that $e_1, e_2, ..., e_j$ $(j < k)$ have the same vehicle $b_1$, while $e_{j+1}, ..., e_k$ all concern the same vehicle $b_2$. In other words, the path $P$ requires the user to (i) take vehicle $b_1$ from node $u$ at time $t_u$ (where $u$ and $t_u$ are the starting node and departure time of $e_1$, respectively), and then (ii) get off $b_1$ at node $w$ and transfer to vehicle $b_2$ at time $t'_u$ (where $t'_u$ is the departure time of $e_{j+1}$), and stay on $b_2$ until she reaches her destination. In that case, instead of presenting $P$ to the user, we could provide her with a concise description of $P$, which is a sequence of two tuples: $(u, b_1, t_u)$, $(w, b_2, t'_u)$. Such a concise path is more user-friendly as it is easier to comprehend and remember than $P$. Moreover, the concise representation of query results is favorable to the compression techniques in Section 7, since it reduces the amount of label decompression required during query processing.

To generate a concise path $P_c$ for a query, a straightforward approach is to first derive the corresponding path $P$, and then convert $P$ into $P_c$ by identifying the edges on $P$ where there are changes of vehicles. Interestingly, our query algorithm can be easily modified to return $P_c$ even without deriving $P$. In particular, given a path query $q$, we first invoke our candidate generation and refinement algorithms (in Section 4.1) to identify the path sketch $sk_{ans} = \langle l_u, l_e \rangle$ of the query result. After that, we invoke the PathUnfold algorithm to unfold the labels in $sk_{ans}$, with one slight modification: during the unfolding process, whenever we encounter a label $l$ whose vehicle does not equal null (i.e., the path corresponding to $l$ does not involve a transfer), then we do not unfold $l$. The output of modified algorithm would then be a partially unfolded path, which is a sequence where each element is either an edge in $G$ or a label whose corresponding path is carried out by one vehicle. After that, it is straightforward to construct $P_c$ from the partially unfolded paths. This partial unfolding approach leads to higher query efficiency since it not only reduces the number of paths to unfold, but also accelerates the construction of $P_c$.

Extended Timetables. So far we have assumed that we are given a timetable graph $G$ with a finite set $E$ of temporal edges. For example, $E$ corresponds to the predefined vehicles routes on a weekday. However, if we construct a TTL index on such a timetable graph $G$, then the index can only return paths whose starting and ending time are within the same day, e.g., we will miss paths that start on late Monday night and ends on early Tuesday morning. To address this issue, we can extend $E$ to include the timetables of two consecutive weekdays, and construct a TTL index accordingly. Then, the index could return any path whose total duration does not exceed 24 hours. In case that the weekend timetable is different from that of a weekday, we can create three additional TTL indices, each of which incorporates the timetables of two consecutive days from Friday to Monday. We note that such indexing partitioning approach is widely adopted in spatial-temporal indexing (e.g., [24]).

9. RELATED WORK

A plethora of techniques [6, 7, 9, 11, 15, 20–22, 26, 27, 31, 34, 36] have been proposed for processing EAP, LAP, and SDP queries on timetable graphs. Among them, the states of the art include the Connection Scan Algorithm (CSA) [15, 37], the Contraction Hierarchies for Timetables (CHT) [20], and T.Patterns [4]. In particular, CSA represents the timetable graph $G$ as two sequences of edges, such that the first (resp. second) sequence sorts edges in ascending order of their departure timestamps (resp. in descending order of their arrival timestamps). For any path query $Q$, CSA derives the query answer using one linear scan of one of the edge sequences. This technique incurs very small preprocessing overheads, and is shown to outperform the temporal Dijkstra’s algorithm [9] in terms of query time.

Meanwhile, CHT [20], preprocesses $G$ by constructing shortcuts (i.e., artificial edges) among the nodes in $G$, such that each shortcut captures a fastest route between the two nodes that it connects. During query processing, CHT employs a bidirectional Dijkstra-like search from the source and destination nodes simultaneously, and it utilizes the pre-computed shortcuts to reduce the number of nodes that need to be traversed. T.Patterns [4] exploits a similar idea to CHT, but instead of maintaining shortcuts, it pre-computes a set of fastest paths in $G$ and record them in a set $S$. Then, when processing queries, T.Patterns utilizes the pre-computed paths to construct the major parts of the query results, which improves query performance. Nevertheless, T.Patterns does not guarantee exact query results, e.g., its answer for an EAP query might not be an actual EAP. In other words, T.Pattern trades query accuracy for efficiency. In our experiments, we compare TTL with CSA and CHT instead of T.Patterns, since we aim to provide exact answers for path queries.

Apart from CSA, CHT, and T.Patterns, other techniques for timetable graphs can be divided into two categories. Techniques in the first category [21, 22, 26, 27, 36] convert $G$ into a time-expanded graph $G'$, such that each spatiotemporal event in $G$ (e.g., arrival of a vehicle $b$ at a station $u$) is mapped to a node in $G'$, and two consecutive events (e.g., arrival of $b$ at $u$, followed by its departure) is mapped to a directed edge in $G'$. Based on $G'$, various techniques [21, 22, 26, 27, 36] have been proposed to accelerate path queries. However, as pointed out in [5, 7], such techniques are generally not incomparable to the state-of-the-art methods that process
queries on $G$ instead of $G'$. On the other hand, techniques in the second category [10, 12, 14, 17, 29, 30, 32, 35] considers query types that are more sophisticated than EAP, LDP, SDP, in that they take into account advanced query constraints, such as the cost of making a transfer, the fare of a ride, etc.

In addition, there is a line of research [9, 11, 13, 18, 19, 23, 25– 27, 33] on route planning on time-dependent graphs, i.e., simple graphs where the length of each edge is a function of time. Such graphs are typically used to model road networks where the traveling time of an edge is a certain pre-defined function of time. Techniques developed for time-dependent graphs can also be applied on timetable graphs (via a sophisticated mapping from timetables to time-dependent functions), but as shown in [6, 11, 20], they are inferior to CHT in terms of query performance on timetable graphs.

Finally, there exists a large body of literature on processing shortest path and distance queries on simple graphs without temporal information (see [5] for a recent survey). The classic approach for this problem is Cohen et al.’s 2-hop labelling (2HL) [8] for distance queries. Cohen et al. present a preprocessing algorithm for 2HL that achieves an $O(\log n)$ approximation in terms of index size, but the algorithm has $O(n^2)$ complexity and does not scale to large graphs. To address this issue, the state-of-the-art approach is to construct 2HL based on a total order on the nodes. The resulting indexing method is referred to as Hierarchical Hub Labelling (HHHL) [2, 3]. Compared with Cohen et al.’s method, HHHL provides higher query efficiency and smaller pre-computation cost. However, approximation algorithms for HHHL (in terms of index size) remain an open problem.

10. EXPERIMENTS

This section experimentally evaluates TTL on a machine with an Intel E5-2650v2 2.60GHz CPU and 64GB RAM, running Ubuntu 14.04.1. All methods tested are implemented in C++ and compiled with GCC 4.8.2. In each experiment, we repeat each method 5 times, and report the average measurement.

Datasets. We use 11 publicly available datasets from [1] as our datasets. Each dataset records the timetable of the public transportation network of a major city or country on a weekday. Table 3 shows the characteristics of the data.

Query Sets. For each dataset, we generate 10° queries for EAP, LDP, and SDP, respectively, and we report the average query time of each method for each query set. Each query is generated by selecting source and destination node uniformly at random. In addition, the starting and ending timestamps of each query are randomly generated, except that the starting (resp. ending) timestamp has a uniform distribution from $0$ to $\tau$.

Methods. We evaluate two versions of TTL: one with the compression methods in Section 7 applied (referred to as C-TTL), and one without (referred to as TTL). For each version, we test its query performance under two scenarios: when it returns a complete path in answering a query, and when it returns a concise representation of the path. We compare all versions of TTL with the state-of-the-art indexing methods on timetable graphs, namely, CHT [20] and CSA [15]. Unless otherwise specified, we use heuristic algorithm in Section 6.2 to decide the node ordering used in TTL and C-TTL.

10.1 Performance of Query Processing

In the first set of experiments, we evaluate the query efficiency of all methods. Figure 3 shows the average query time of each method for SDP queries in log-scale. Observe that, TTL and C-TTL significantly outperform CSA and CHT in all cases. In particular, the average query time of TTL is always below 30µs, and is at least three (resp. two) orders lower than that of CSA (resp. CHT) in almost all cases. C-TTL is slightly slower than TTL (as it needs to decompress labels during query processing), but is still orders of magnitude faster than CSA and CHT. Meanwhile, CHT is superior to CSA, as it adopts a more advanced index structure than the latter.

Interestingly, the query time of TTL and C-TTL does not always increase with the size of the input dataset. To explain, observe that the average query cost of TTL and C-TTL is decided by two factors: (i) the average number of labels in each label set, denoted as $l_{avg}$, and (ii) the average number of path unfolding operations required during path reconstructions, which is roughly comparable to the average number of nodes on the query result path (denoted as $n_{avg}$). We observe that neither $l_{avg}$ nor $n_{avg}$ necessarily increases with the dataset size. For example, on the Austin dataset, $l_{avg}$ is close to 1600, and $n_{avg}$ is close to 30 for SPD queries; in contrast, on Sweden, we have $l_{avg} \approx 775$ and $n_{avg} \approx 19$ for SPD queries, even though Sweden is more than ten times larger than Austin. In addition, we note that a similar phenomenon is also demonstrated in previous work [2, 3] on labelling indices for conventional graphs, i.e., the query cost of a labelling approach depends more on the input data’s topology than its size.

The above results concern the case when we return a detailed path for each SPD query. In addition to that, Figure 3 also plots the performance of TTL and C-TTL when they return concise paths as query results. Observe that the query efficiency of both TTL and C-TTL improves noticeably when concise paths are adopted. This is because when TTL and C-TTL return concise paths instead of complete paths, their costs of path reconstruction during query processing are significantly reduced, due to the decreased amount of path unfolding operations, as mentioned in Section 8. In addition, the performance gap between TTL and C-TTL is reduced in the case of concise paths, because C-TTL performs a smaller number of label decompressions when constructing concise paths.

Apart from SDP queries, we also evaluate the average query time of each method for EAP and LDP queries, respectively, and we observe that TTL and C-TTL consistently outperform CSA and CHT by large margins. Interested readers are referred to Appendix D for these experimental results.

10.2 Preprocessing and Space Overheads

Our second set of experiments evaluates the space overheads of each method. Figure 4 shows the results in log-scale. The index size of CSA is rather small, since it only requires storing two copies of $G$ (in the form of sorted edge sequences). The space consumption of CHT is comparable to that of CSA, which indicates that the shortcuts that it creates take roughly the same space as the input graph $G$. On the other hand, the space costs of TTL and C-TTL are considerably larger than those of CSA and CHT. However, the relatively large space overhead of TTL and C-TTL is justified by their superior query performance against CSA and CHT. Furthermore,
even on the country-scale transportation network Sweden, the index size of TTL (resp. C-TTL) is only a few hundred MB, which can easily fit in the main memory of a commodity machine. In other words, the space overheads of TTL and C-TTL are still acceptable for practical applications.

In the next set of experiments, we evaluate the effectiveness of label compression schemes. In particular, on each dataset, we construct four TTL indices: (i) one without compression, denoted as \( \mathcal{L} \); (ii) one with route-based compression, denoted as \( \mathcal{L}_1 \); (iii) one with pivot-based compression, denoted as \( \mathcal{L}_2 \), and (iv) one with both route-based and pivot-based compressions, denoted as \( \mathcal{L}_3 \). We define \( \Delta_i = |\mathcal{L}_i| - |\mathcal{L}| \) (\( i = 1, 2, 3 \)). Table 4 shows the values of \( \Delta_1/|\mathcal{L}|, \Delta_2/|\mathcal{L}|, \) and \( \Delta_3/|\mathcal{L}| \) on each dataset. Observe that, by using route-based compression and pivot-based compression separately, our index size can be reduced by up to 38.00% and 38.78%, respectively. When both compression techniques are applied, the size of our TTL indices can be reduced by up to 61.43%.

This reduction in space consumption comes at a moderate cost of query efficiency, though. For instance, on the Budapest dataset, our compression techniques reduce the size of C-TTL by 41.71%, but increase the query time (when returning concise paths) from 10\( \mu \)s to 12\( \mu \)s. Nonetheless, even with compression applied, the query performance of C-TTL is still significantly better than that of CSA and CHT, as shown in Figure 3. As such, C-TTL is a competitive method compared to TTL, especially when the index size is a concern (e.g., when the index is to be loaded in a mobile device).

Finally, Figure 5 shows the pre-processing time of all methods. CSA incurs the smallest pre-computation overheads as it only requires sorting the edges in \( G \) twice. CHT has a much larger pre-processing cost, since it needs to construct a sizable number of shortcuts on \( G \). The pre-computation time of TTL and C-TTL are larger than that of CSA and CHT, but is still below 17 minutes even on the largest dataset Sweden. Furthermore, TTL and C-TTL have similar pre-processing overheads, which indicates that the compression algorithms adopted by C-TTL entail negligible additional costs. We also refer interested readers to Appendix D for additional experiments on the effectiveness of TTL’s pre-processing algorithm.

11. CONCLUSIONS

This paper presents Timetable Labelling (TTL), an efficient indexing technique for route planning on public transportation networks. We investigate various aspects in the design of TTL indices, and propose advanced algorithms for query processing, index construction, and label compression. With extensive experiments on real datasets, we show that TTL significantly outperforms the state-of-the-art in terms of query efficiency.

12. ACKNOWLEDGMENTS

This work was supported by AcRF Tier 2 Grant ARC19/14 from the Ministry of Education, Singapore, an SUG Grant from Nanyang Technological University, Grant No. 14JC140300 from Shanghai Municipal Commission of Science and Technology, and a gift from Microsoft Research Asia.
13. REFERENCES


APPENDIX

A. PROOFS

Proof of Lemma 1. We prove the lemma by contradiction: Assume that there does not exist a label \((w, t_w, t_a, b, p)\) in \(L_u(u)\) or a label \((w', t_{w'}, t_{a'}, b', p')\) in \(L_v(v)\) such that either of them satisfies any one of the conditions in the lemma. In other words, there are two cases, stated as follows.

Case 1: There does not exist a canonical path from \(u\) to \(v\).

Case 2: There does not exist a node \(u^*\) such that both the paths from \(u\) to \(u^*\) and from \(u^*\) to \(v\) contain a canonical path respectively.
Given that $P$ is the result of the path query $q$ from $u$ to $v$, we have that $P$ is not dominated by the other paths that answer $q$ by Definitions 2, 3, and 4. Besides, any sub-path of $P$ is also not dominated by the other paths. If not, we can replace the sub-path with the dominated one.

Let $w^o$ be the node with highest rank among nodes in $P$. If $w^o$ equals $u$ or $v$, then $P$ is a canonical path (by Definition 5), which contradicts Case 1. On the other hand, if $w^o$ does not equal $u$ or $v$, let $P_1$ be the sub-path from $u$ to $w^o$, and $P_2$ be the sub-path from $w^o$ to $v$. As such, we have $P_1$ and $P_2$ as canonical paths (by Definition 5), which contradicts Case 2. Thus, the lemma is proved.

\[\square\]

**Proof of Lemma 2.** Let $l = (u, t_u, t_a, \ldots, b, p)$ be an in-label in $L_{in}(v)$, and $P = (u, t_u, t_a, \ldots, b, p)$ be an out-label in $L_{out}(v)$. The lemma says that if we remove $l$ or $l'$, there exists a path query $q$ whose result $P$ does not satisfy the constraints in Lemma 1.

Consider $l$. Given that $(u, t_u, t_a, b, p) \in L_{in}(v)$, we have $o(u) < o(v)$ and $l$ is pertinent to a canonical path $P$ from $u$ to $v$. Let $q$ be a path query from $u$ to $v$ with starting timestamp $t_d$ and ending timestamp $t_a$ and $P$ be the result of $q$. By contradiction, assume that after removing $l$ from $L_{in}(v)$, $P$ still satisfies Lemma 1. Since $o(u) < o(v)$, by Definition 7, $v$ cannot appear in any label of $L_{out}(v)(u)$. Thus, there must exist a node $w$, such that $(w, t_w, t_a, \ldots, b, p) \in L_{out}(u)$, $(w, t_w, t_a, \ldots, b, p) \in L_{out}(v)$, and $t_u < t_d$. That is, the path $P$ from $u$ to $v$ passes $w$, and $w$ has a higher rank than both $u$ and $v$, which contradicts the Rank Constraint in Definition 5. The case for $l'$ can be proved in a similar manner.

\[\square\]

**Proof of Lemma 3.** Consider a path query $q$ from $u$ to $v$ with a starting timestamp $t$ and an ending timestamp $t'$. Let $P_{ans}$ be the result of $q$, $l = (u, t_u, t_a, \ldots, b, p)$ be an out-label of $u$, and $l' = (u', t'_u, t'_a, \ldots, b, p')$ be an in-label of $v$. If $l$ is pertinent to $P_{ans}$ or the sub-path of $P_{ans}$, we have $t < t_d$ and $t_u < t'$, which is checked by Line 4 of Algorithm 1. Similarly, the timestamps of $l'$ are checked by Line 10 of Algorithm 1. By Lemma 1, there are three cases:

**Case 1.** If $w = v$, we have $P_{ans}$, which is pertinent to $l$.

**Case 2.** If $w' = u$, we have $P_{ans}$, which is pertinent to $l'$.

**Case 3.** If $w \neq v, w' \neq u$, $t_u \leq t'_d$ and $w = w'$, we have $l$ and $l'$ each of which is pertinent to a sub-path of $P_{ans}$.

Obviously, Case 1 and Case 2 can be correctly handled by Lines 6-8 and Lines 12-14 respectively. Now, we focus on the discussion of Case 3. By contradiction, assume that Lines 15-20 cannot find a sketch corresponding to $P_{ans}$. Since labels in $L_{out}(u)$ and $L_{in}(v)$ are sorted, we have $t_u > t'_d$ for all pairs of $l$ and $l'$ where $w = w'$. This contradicts Case 3, which completes the proof.

\[\square\]

**Proof of Lemma 4.** Without loss of generality, let $o(u) < o(v)$, i.e., the rank of $u$ is higher than that of $v$. Let $P_1$ denote the sub-path of $P$ from $u$ to $p$, and $P_2$ the sub-path of $P$ from $p$ to $v$. By Definition 6, the rank of $p$ is higher than all nodes in $P$, except $u$ and $v$. As such, both $P_1$ and $P_2$ satisfy the Rank Constraint in Definition 5.

Assume on the contrary that either $P_1$ or $P_2$ is not a canonical path. Consider that $P_1$ is not a canonical path. Since $P_1$ satisfies the Rank Constraint, there must exist another path $P_\alpha$ from $u$ to $p$ dominating $P_1$. This contradicts that $P$ is a canonical path. The case for $P_2$ can be similarly established.

\[\square\]

**Proof of Lemma 5.** By Lemma 1 and Lemma 3, given a starting timestamp $t$ and an ending timestamp $t'$, we can find the sketches pertinent to all paths from $u$ to $v$, each of which is an SDP regarding its departure time $t_d \geq t$ and arrival time $t_a \leq t'$. Denote these paths by $SDP(u, t, v, t')$. As such, when $t' = +\infty$, we can obtain the sketches pertinent to paths in $SDP(u, v, t, +\infty)$. Hence, we are able to find the EAP whose arrival time is the minimum in $SDP(u, v, t, +\infty)$. The similar analysis can be applied to the extension of LDP queries.

\[\square\]

**Proof of Lemmas 6 and 7.** Consider Lemma 6. Given the non-dominated path $P$ starting from $u$ at time $t \in T_d$ and ending at $v$ at time $t'$, assume that $P$ is not an EAP. In other words, there exists another path $P'$ such that $P'$ also starts from $u$ at time $t$ but ends at $v$ at time $t''$, which is earlier than $t'$, i.e., $t'' < t'$. However, this violates that $P$ is a non-dominated path, which completes the proof of Lemma 6. The proof of Lemma 7 can be established in the similar manner.

\[\square\]

**Proof of Lemma 8.** We first show that each label in $L$ constructed by Algorithm 3 is pertinent to a canonical path of $G$. Consider a label $l = (u, t_u, t_a, \ldots, b, p)$ in the in-label set $L_{in}(v)$ of $v$, which is pertinent to a path $P'$ from $u$ to $v$ with departure time $t_d$ and arrival time $t_a$. As all nodes whose ranks are higher than $u$ are removed before the computation on $u$, we have that $P'$ fulfills Rank Constraint in Definition 5. Assume that $P'$ does not satisfy Dominance Constraint, i.e., there exists another canonical path $P''$ from $u$ to $v$ with departure time $t''_d > t_d$ and arrival time $t''_a < t_a$. Since we compute the outgoing edges of $u$ in descending order of their departure timestamps, $P'$ is computed before $P''$. As such, $P'$ should not be added to $L_{in}(v)$ due to that $[t'_d, t'_a] \subseteq [t_d, t_a]$, which leads to a contradiction. In sum, $P$ is a canonical path.

Now we prove that each canonical path of $G$ is pertinent to a label in $L$. Assume by contradiction that there exists a canonical path $P^*$ from $u$ to $v$ such that $P^*$ is not pertinent to any label in $L$. By Lemmas 6 and 7, given a starting timestamp $t$, we can find all EAPs from $u$ to $v$ by the modified version of Dijkstra’s algorithm; and given an ending timestamp $t'$, we can find all LDPs to $v$ by the reverse version of Dijkstra’s algorithm. As such, $P^*$ must be found by the modified version or the reverse version of Dijkstra’s algorithm. This indicates that there must already exists a label $l'$ in $L$ such that $l'$ is pertinent to another path $P'$ and $P'$ dominates $P^*$. This contradicts to the assumption that $P^*$ is a canonical path, as a consequence. IndexBuild correctly produces a TTL index.

To analyze the complexity, we consider the time cost in each iteration. In each iteration, a vertex is chosen and a temporal Dijkstra is started from this vertex. Observe that each edge will be traversed at most once, and during the traversal, a vertex is popped at most once from the heap. By maintaining a Fibonacci Heap, the amortized cost hence is $O(n \cdot \log n)$. The label insertion process as shown in Lines 29-33 requires only $O(1)$ time for each insertion by forcing the vertices sorted in the out-label (resp. in-label) by the vertex level order and then in decreasing order of arrival timestamps (resp. in increasing order of departure timestamps). So in each iteration, the complexity is $O(n + m \cdot \log n)$. There are $n$ iterations, and the time-complexity can be bounded by $O(n \cdot m + n^2 \cdot \log n)$, which completes the proof.

\[\square\]

**Proof of Lemma 9.** By the definition of $I(u_i)$, we have the following two inequalities:

$$|I(u_i)| \leq \sum_{v \in V} |T_d(v)| + \sum_{v \in V} |T_a(v)| = \alpha, \quad (3)$$

$$|I(u_i)| \leq |R_t(u_i)|. \quad (4)$$

Based on the above inequalities, we have

$$|I(u_i)| = \sqrt{|I(u_i)|} \cdot \sqrt{|I(u_i)|} \leq \sqrt{\alpha} \cdot \sqrt{|R_t(u_i)|}. \quad (5)$$
Proof of Lemma 10. We define the marginal out-coverage of \( v \) as 
\[ I_{\text{out}}(v_i) = \{ (v_i, \cdot, \cdot, \cdot, \cdot) \in I(v_i) \} \]
and marginal in-coverage of \( v \) as 
\[ I_{\text{in}}(v_i) = \{ (\cdot, v_i, \cdot, \cdot, \cdot) \in I(v_i) \} . \]
That is, \( |I(v_i)| = |I_{\text{out}}(v_i)| + |I_{\text{in}}(v_i)| \). Therefore, we have

\[
\sqrt{|R'_i(v_i)|} \leq |I_{\text{out}}(v_i)| + |I_{\text{in}}(v_i)| = \sqrt{(|I_{\text{out}}(v_i)| + |I_{\text{in}}(v_i)|)^2} = |I(v_i)|, \]

which completes the proof. \( \square \)

Proof of Lemma 11. Given the ordering \( o \) outputted by the approximation algorithm, we have \( u_1, u_2, \ldots, u_n \) with \( o(u_i) = i \). Let \( r \) be a tuple in \( R_i(u_i) \). We define \( c(r) = \frac{1}{\sqrt{|R'_i(v_i)|}} \) as the contribution of \( r \) to the amount of \( \sqrt{|R_i(v_i)|} \). Then, we have

\[
\sqrt{|R_i(v_i)|} = \sum_{r \in R_i(u_i)} c(r). \tag{6}
\]

As such, we have \( \sum_{i=1}^n \sqrt{|R_i(v_i)|} = \sum_{i \in R} c(r) \).

Now, consider the optimal ordering \( o' \), which is \( v_1, v_2, \ldots, v_n \) with \( o'(v_i) = i \). Let \( r_{i,j} \) be the \( j \)-th tuple in \( R'_i(v_i) \) that will be covered if we choose the nodes with the order given by the approximation algorithm. Then, we have \( c(r_{i,j}) \leq \frac{1}{\sqrt{|R'_i(v_i)|}} \).

This is because (i) the approximation algorithm always chooses the node with the maximum residual coverage, and (ii) \( r_{i,j} \) has been covered but not by node \( v_i \).

On the other hand, we have

\[
\frac{1}{\sqrt{1}} + \frac{1}{\sqrt{2}} + \ldots + \frac{1}{\sqrt{|R'_i(v_i)|}} \leq \sqrt{|R'_i(v_i)|}. \tag{7}
\]

Based on the aforementioned observations, we have

\[
\sum_{i=1}^n \sqrt{|R_i(v_i)|} = \sum_{r \in R} c(r) = \sum_{i=1}^n \frac{|R'_i(v_i)|}{\sqrt{|R'_i(v_i)|}} \cdot \sum_{j=1}^{|R'_i(v_i)|} c(r_{i,j}) \leq \sum_{i=1}^n \sum_{j=1}^{|R'_i(v_i)|} \frac{1}{\sqrt{|R'_i(v_i)|}} - j + 1 \leq 2 \cdot \sum_{i=1}^n \sqrt{|R'_i(v_i)|}, \quad \text{(By Inequality 7)}
\]

which completes the proof. \( \square \)

Proof of Theorem 1. Let \( |L_{\text{apx}}| \) be the index size with the node ordering given by the approximation algorithm, and \( |L_{\text{opt}}| \) be the index size with the optimal node ordering, then we have

\[
|L_{\text{apx}}| = \sum_{i=1}^n |I(u_i)| \leq \sqrt{\alpha} \cdot \sum_{i=1}^n \sqrt{|R_i(v_i)|} \leq 2 \sqrt{\alpha} \cdot \sum_{i=1}^n \sqrt{|R_i(v_i)|} \leq 2 \sqrt{\alpha} \cdot \sum_{i=1}^n |I(v_i)| = 2 \sqrt{\alpha} \cdot |L_{\text{opt}}|,
\]

this derives the approximation ratio.

To obtain the node order with approximation guarantee, the main challenges are to (i) compute the number of the non-dominated paths covered by each node, and (ii) update the number when another node is ordered. We address these challenges by maintaining three hash tables, \( T_1, T_2, \) and \( T_3 \). In particular, for each item in \( T_1 \) (resp. \( T_2 \)), its key is a triple \( (u, v, t) \) (resp. \( (u, v, t_d) \)), and its value is the arrival time \( t_a \) (resp. departure time \( t_d \)) of the canonical path that is from \( u \) to \( v \) with departure time \( t_d \) and arrival time \( t_a \). As such, \( t_a \) (resp. \( t_d \)) of a non-dominated path can be retrieved from \( T_1 \) (resp. \( T_2 \)) in constant time given \( (u, v, t) \) (resp. \( (u, v, t_d) \)). Moreover, a counter \( r(v) \) for a node \( v \) is maintained to record the number of non-dominated paths covered by \( v \) but not covered by any node \( u \) with \( o(u) < o(v) \). Since the number of non-dominated paths starting from a node \( v \) is at most \( O(n \cdot d_{\text{out}}(v)) \), the space complexity of three hash tables is \( O(n \cdot m) \).

Specifically, we first compute \( r(v) \) for each node \( v \) and construct the hash tables using the temporal Dijkstra’s algorithm, which can be finished in \( O(n^2 \cdot m) \) time. Then, in each iteration, we choose the node \( v' \) with the highest \( r(v') \) score, after which we identify the non-dominated paths covered by \( v' \), and update \( r(v) \) of the other nodes \( v \) accordingly. Consider the total cost of retrieving the non-dominated paths in all \( n \) iterations. To retrieve the non-dominated paths covered by node \( v' \), we need to examine all possible non-dominated paths that start from a node \( u \) and pass through \( v' \). To compute such paths, we first examine all incoming edges \( e \) of \( v' \), and identify the arrival timestamp \( t_a \) of \( e \); after that, we search \( T_2 \) for an entry with a key \( (u, v', t) \), and obtain the corresponding departure time \( t_1 \). Then, we inspect all outgoing edges \( e' \) of \( v' \), and obtain the departure timestamp \( t_d \) of \( e' \); after that, we search \( T_1 \) for an entry with the key \( (v', w, t_d) \), and obtain the corresponding arrival time \( t_2 \). Finally, we search \( T_3 \) for an entry with a key \( (u, v', t) \), to see if the path is a non-dominated path that has not been covered. Note that the checking for all paths from \( u \) to \( w \) can be done in \( d_{\text{out}}(v') + d_{\text{out}}(v') \). As there are \( O(n^2) \) combinations of \( u \) and \( w \), the cost to compute the non-dominated paths covered by \( v' \) is \( O(n^2 \cdot \text{d}_{\text{out}}(v')) \). Summing up the costs in \( n \) iterations, the total cost of computing all non-dominated paths is \( O(n^2 \cdot m) \).

Next, consider the cost of updating \( r(v) \) of an unordered node \( v \), after we obtain all non-dominated paths covered by \( v' \). Let \( P \) be a non-dominated path covered by \( v' \). If \( v \) also covers \( P \), we decrease \( r(v) \) by one. Let \( u \) (resp. \( w \) ) be the starting (resp. ending) node of \( P \), and \( t_d \) (resp. \( t_a \) ) be the departure (resp. arrival) time of \( P \). To check whether \( P \) is covered by \( v \), we first search \( T_1 \) for an entry with a key \( (u, v, t_d) \), and obtain the corresponding arrival time \( t_1 \); then, we search \( T_2 \) for an entry with a key \( (v, w, t_a) \), and obtain the corresponding departure time \( t_2 \). If \( t_1 \) is no larger than \( t_2 \), then \( P \) is covered by \( v \), and we decrease \( r(v) \) by one. Such a check takes constant time. As we needs to perform such check for each node on each of the \( O(n \cdot m) \) non-dominated path covered by \( v' \), the total cost of the checks is \( O(n^2 \cdot m) \). In summary, the approximation algorithm runs in \( O(n^2 \cdot m) \) time and \( O(n \cdot m) \) space. \( \square \)

Proof of Corollary 1. Consider a label \( l = (\cdot, \ldots, \cdot, p) \). If \( p = \text{null} \), \( l \) is pertinent to an edge in \( G \). Otherwise, by Definition 7, \( l \) is pertinent to a canonical path consisting of at least two edges. If \( p \neq \text{null} \), we recursively find the edges that form the sub-paths \( P_i \) and
B. QUERY WITH COMPRESSED LABELS

In Section 7, we have explained how we can reconstruct original labels from the ones generated by our compression methods. In what follows, we clarify how we can process queries on the compressed index. In particular, we first explain the candidate generation and refinement process on compressed labels, and then introduce the algorithm for path reconstruction. For ease of exposition, we focus on SDP queries, but our solution can be easily extended to EAP and LDP queries.

Candidate Generation and Refinement. Given a query asking for the SDP from \( u \) to \( v \) with a starting timestamp \( t \) and an ending timestamp \( t' \), we generate the set \( S_{uv} \) of path sketches by linearly scanning \( L_{out}(u) \) and \( L_{in}(v) \) using Algorithm 1, with one modification: whenever we encounter a compressed label \( l \), we decompress it based on the compression technique used. In particular, if \( l \) is produced by the route-based compression, then \( l \) should be in the form of \( \langle r, null, null, r, \ldots \rangle \), i.e., the route information of all labels associated with \( l \) can be easily retrieved from the timetable associated with \( r \). Next, consider that \( l \) is generated by the pivot-based compression. In that case, \( l \) should be in the form of \( \langle r, null, null, \ldots, r \rangle \), i.e., the labels with pivot-based compression. Without loss of generality, assume that \( o(u) < o(p) < o(v) \). Then, as explained in Section 7.2, the left (resp. right) children of \( l_1, l_2, \ldots, l_j \) are stored in \( L_{in}(p) \) (resp. \( L_{out}(p) \)) as such, we can recover \( l_1, l_2, \ldots, l_j \) by inspecting the labels in \( L_{in}(p) \) and \( L_{out}(p) \) based on the pivot information of \( v \). After that, we can replace \( l \) with \( l_1, l_2, \ldots, l_j \) in the linear scan. After the linear scan, the set \( S_{uv} \) of all the path sketches with respect to the SDP query can be obtained. Then, we identify the sketch \( s_{ans} \) in \( S_{uv} \) that is pertinent to the SDP query result, using the same approach as described in Section 4.1. Note that all labels in \( s_{ans} \) are not compressed, due to the decompression operations performed during the linear scan.

Path reconstruction. Once \( s_{ans} \) is identified, we proceed to reconstruct the path corresponding to \( s_{ans} \). Algorithm 4 presents the pseudo-code of \( CPathUnfold \), namely, the method for path reconstruction with the compressed index. \( CPathUnfold \) takes as input a label \( l \) (in \( s_{ans} \)), which, as mentioned, is guaranteed to be uncompressed due to the way \( s_{ans} \) is generated. Without loss of generality, assume that the left child \( l' \) of \( l \) is compressed. Then, the departure timestamp of \( l' \) is the same as that of \( l \) and the starting node and ending node of \( l' \) can be learnt from \( l \). Therefore, to decompress \( l' \), it suffices to identify the arrival timestamp and vehicle of \( l' \). If \( l' \) is produced from the route-based compression, then we can pinpoint the arrival timestamp and vehicle of \( l' \) by inspecting the timetable of the route associated with \( l' \). Meanwhile, if \( l' \) is generated by the pivot-based compression, then \( l' \) can be reconstructed by inspecting the children of \( l' \) and the pivot information in \( l' \) (as explained in Section 7.2). The case that the right child of \( l \) is compressed can be addressed in a similar manner.

C. ADDITIONAL EXAMPLE

Example 6. Consider the timetable graph \( G' \) in Figure 2a with a node order \( o(v_2) = 1, o(v_1) = 2, \) and \( o(v_3) = 3 \). The index construction algorithm first selects \( v_2 \), and performs a temporal Dijkstra traversal from \( v_2 \) with the largest departure timestamp, i.e., 4. Then it obtains one EAP, based on which \( IndexBuild \) inserts a label \( \langle v_2, 4, 5, b_3, null \rangle \) into \( L_{in}(v_2) \). Next, it performs a temporal Dijkstra traversal from \( v_2 \) at timestamp 3 and timestamp 2 in sequence. These two traversals lead to two additional labels. After that, a backward Dijkstra is performed from \( v_2 \) with the smallest arrival timestamp, i.e., 2. Then it obtains one LDP, and \( IndexBuild \) inserts a label \( \langle v_2, 1, 2, b_1, null \rangle \) into \( L_{out}(v_1) \) corresponding to this LDP. Afterwards, two more backward Dijkstra traversals are performed from \( v_2 \) at timestamp 3 and 4 in a row, resulting in two more labels. These finishes the traversal from \( v_2 \), and \( v_2 \) is removed from the graph. Then \( v_1 \) and \( v_3 \) have no incoming and outgoing edges, and both incur no label cost during the traversal. This finishes the index construction, ending up with 6 labels as shown in Table 5.

D. ADDITIONAL EXPERIMENTS

D.1 EAP and LDP queries

Figure 6 and Figure 7 report the average query time for EAP and LDP queries, respectively. Observe that the performance of each method for EAP and LDP queries on each dataset is generally similar. However, CSA and CHT incur several times less query time for EAP and LDP queries than for SDP queries. This is because, to answer EAP (resp. LDP) queries, they only maintain the earliest arrival time to (resp. latest departure time from) each node that is traversed. However, to answer an SDP query, they both need to maintain a list to record the departure time and arrival time of each non-dominated path that ends at a certain traversed node, which can be excessively large. Overall, \( TTL \) and \( C-TTL \) still outperform CSA and CHT by a large margin.
Table 5: The label sets of $v_1$, $v_2$, and $v_3$ in a TTL index of $G'$ in Figure 2a, with a node order $o(v_2) = 1$, $o(v_1) = 2$, and $o(v_3) = 3$.

<table>
<thead>
<tr>
<th>$v$</th>
<th>$L_{in}(v)$</th>
<th>$L_{out}(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_2$</td>
<td>$\emptyset$</td>
<td>$(v_2, 1, 2, b_1, null)$, $(v_2, 2, 3, b_2, null)$, $(v_2, 3, 4, b_3, null)$</td>
</tr>
<tr>
<td>$v_1$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>$(v_2, 2, 3, b_1, null)$, $(v_2, 3, 4, b_2, null)$, $(v_2, 4, 5, b_3, null)$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

D.2 Index Construction and Node Orders

In this set of experiments, we evaluate the efficiency of our index construction algorithm and the effectiveness of our ordering algorithms. In particular, we first compare our index construction algorithm with a baseline approach that (i) runs the temporal Dijkstra’s algorithm $O(n)$ times to compute all temporal shortest paths in $G$, and then (ii) examines the paths obtained to identify the canonical paths. We denote the baseline approach as brute-force Dijkstra.

Figure 8 reports the index construction time of Algorithm 3 (referred to as H-Order) and brute-force Dijkstra, both of which adopt H-Order as the ordering methods. Observe that Algorithm 3 is at least two orders of magnitude faster than brute-force Dijkstra, even though both algorithms have the same worst-case time complexity. The main reason is that Algorithm 3 performs a more advanced version of the Dijkstra’s search from $v_i$, with various heuristics adopted to prune the search space (see Section 5). For instance, whenever it encounters a node $v_j$ with $o(v_j) < o(v_i)$ during the search, it would omit all edges pertinent to $v_j$, since any path that contains those edges would violate the Rank Constraint in Definition 5 (and hence, cannot be canonical). This demonstrates that our index construction algorithm is highly superior to the brute-force Dijkstra approach in terms of practical efficiency.

Next, we compare our heuristic ordering algorithm (referred to as H-Order) and our approximate ordering algorithm (referred to as A-Order), with a baseline approach that orders nodes randomly (denoted by Rand-Order). Figure 9 reports the index size of TTL (in log-scale) for each of the three node ordering methods. The results of A-Order are omitted on some datasets due to its prohibitive memory consumption (> 64GB). Observe that, both A-Order and
H-Order outperform the baseline approach by large margins, which demonstrates the importance of node ordering in index construction and the superiority of our approaches. In particular, the index size of H-Order is more than one order of magnitude smaller than that of Rand-Order on Berlin dataset. Besides, the index size of H-Order is comparable to that of A-Order, which shows the effectiveness of our heuristic ordering approach.

Figure 8 presents the total processing time of TTL for each ordering methods. The total processing time of A-Order is at least 2 orders of magnitude larger than that of H-Order, since A-Order incurs significant overheads in computing all non-dominated paths and updating the marginal coverage of each node. In addition, Rand-Order has a total processing time several times larger than that of H-Order, since the former generates an enormous number of labels (see Figure 9). In summary, our heuristic ordering method yields similar index size as the approximation algorithm, but incurs much less preprocessing cost.