Efficient Route Planning on Public Transportation Networks: A Labelling Approach

Sibo Wang[1] Wenqing Lin[2] Yi Yang[3] Xiaokui Xiao[1] Shuigeng Zhou[3]

[1]Nanyang Technological University [2] Institute for Infocomm Research, Singapore
{wang0759, xkxiao}@ntu.edu.sg linw@i2r.a-star.edu.sg

[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 $\langle u, v\rangle$ 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.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749456 .

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* [32], where (i) each node represents a station; and (ii) each directed edge $\langle u, v\rangle$ 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_d, 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_a, 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 [10], each of the above three types of queries can be solved with a modified version of the Dijkstra's algorithm [17] 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., [7, 12, 16, 21]) 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 [16] or augment G with information about certain important routes [21], 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 [16, 21] still incur considerable query overheads on sizable timetable graphs G, since they require traversing a substantial portion of the temporal

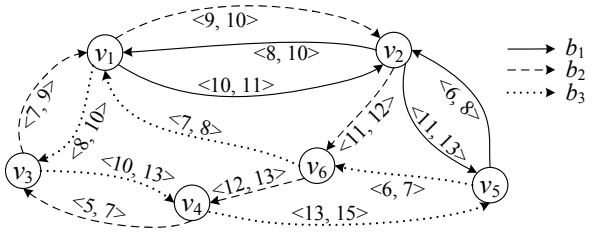


Figure 1: A timetable graph G.

edges in G (or its augmented version). This motivates us to develop a more efficient route planning schema on transportation networks.

Contributions. This paper presents *Timetable Labelling (TTL)*, an efficient indexing technique for route planning on timetable networks. 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. The labels are constructed in such a way that, for any EAP, LDP, or SDP query from a node u to another node v, we can efficient retrieve a tiny summary of the query result by inspecting the label sets of u and v (without inspecting G), and we can easily convert the summary into the corresponding path in $O(k)$ time, where k is the number of nodes in the path. Compared with the existing techniques [10, 16, 21], *TTL* incurs lower query overheads as it completely avoids traversing G during query processing. In particular, in our experiments on 11 real transportation networks, *TTL* is shown to outperform the state of the art by up to 3 orders of magnitude in terms of query efficiency.

TTL is similar in spirit to *hierarchical hub labelling (HHL)* [3] (i.e., the state-of-the-art approach for shortest path queries on simple graphs), as both of them rely on pre-computed label sets for query processing. Nevertheless, *TTL* is not a straightforward extension of *HHL* that simply augments each label set with timetable information. Instead, *TTL* incorporates a set of non-trivial optimization techniques in its preprocessing and query algorithms, which exploit the characteristics of timetable graphs to achieve significant improvements in terms of both query performance and space consumptions. More specifically, we make the following contributions.

- We propose *TTL*, an indexing approach for timetable graphs that assigns two label sets for each node. We devise an advanced query processing algorithm for *TTL*, such that each EAP, LDP, or SDP query from a node u to another node v can be answered with a linear scan of u and v's label sets, followed by an $O(k)$-time process to retrieve the query result P, where k is the number of nodes in P. In contrast, a straightforward query algorithm would require quadratic (instead of linear) time in inspecting the label sets of u and v. (Sections 3 and 4)
- We study the preprocessing algorithm for *TTL* from both theoretical and practical perspectives. On the theoretical side, we show that there is a polynomial time algorithm that returns a *TTL* whose space consumption is $O(\sqrt{m})$ 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 $\langle u, v, t_d, t_a, b\rangle$. We refer to b, t_d, and t_a as the vehicle, departure time, and arrival time of e, respectively, and 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 (reps. 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 $null$.

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 latter 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 $\langle t_d, t_a\rangle$ 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_5 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

Notation	Description
G	a timetable graph
n	the number of nodes in G
$\langle v, u, t_d, t_a, b\rangle$	an edge in E
$P = \langle e_1, \ldots, e_k\rangle$	a path in G (see Section 2)
$o(v)$	the rank of v in the node order o (see Section 3)
$\mathcal{L}$	the TTL on G (see Definition 7)
$\langle v, t_d, t_a, b, p\rangle$	a label in $\mathcal{L}$ (see Definition 7)
$L_{in}(v)$	the in-label set of v (see Definition 7)
$L_{out}(v)$	the out-label set of v (see Definition 7)
$\cdot$	a placeholder that denotes irrelevant information in the context

Table 1: Table of notations.

$t' = 8$, for which we need to take vehicle b_3 from v_5 at time 6, and stay on the vehicle until it arrives at v_1. In other words, if we have an EAP query from v_5 to v_1 with a starting timestamp $t = 5$, then the answer to the query is a path via vehicle b_3 that starts from v_5 at time 6 and ends at v_1 at time 8. On the other hand, for an LDP query from v_1 to v_4 with an ending timestamp $t' = 13$, then the answer is a path that (i) starts from v_1 at time 10 via vehicle b_1, (ii) transfers to vehicle b_2 at v_2, and then (iii) arrives at v_4 at time 13 on b_2. Finally, for an SDP query from v_5 to v_1 with starting time $t = 6$ and ending time $t' = 10$, the query result is identical to the EAP query mentioned above. □

Given G, our objective is to construct an index structure on G that can efficiently answer path queries. Table 1 lists the notations that we will frequently use in the paper. In particular, we will use "$\cdot$" as a placeholder to denote some information that is irrelevant to the discussions in the context.

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*, $L_{in}(v)$ and $L_{out}(v)$, for each node v in G, such that each label in $L_{in}(v)$ (resp. $L_{out}(v)$) 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 o 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.

v	$L_{in}(v)$	$L_{out}(v)$
v_2	$\langle v_1, 9, 10, b_2, null\rangle$ $\langle v_1, 10, 11, b_1, null\rangle$	$\langle v_1, 8, 10, b_1, null\rangle$
v_4	$\langle v_1, 10, 13, null, v_2\rangle$ $\langle v_2, 11, 13, b_2, v_6\rangle$ $\langle v_3, 10, 13, b_3, null\rangle$	$\langle v_1, 5, 9, b_2, v_3\rangle$ $\langle v_3, 5, 7, b_2, null\rangle$

Table 2: The label sets of v_2 and v_4 in a *TTL* index for Figure 1, with a node order $o(v_i) = i$ for $i \in [1, 6]$.

EXAMPLE 2. Consider the timetable graph in Figure 1. Assume that we use a node order $o(v_1) < o(v_2) < \ldots < o(v_6)$. The path P from v_5 to v_1 taking vehicle b_3 at time $t = 6$ is a canonical path, since (i) no path from v_5 to v_1 after time $t = 6$ can arrive at v_1 earlier than P does or arrive v_1 before time $t = 8$ with departure time later than P's, and (ii) the rank of v_1 is the highest among all nodes on P. In contrast, the path from v_6 to v_3 taking b_3 via v_1 at time $t = 7$ is not a canonical path, since the rank of v_1 is higher than those of v_3 and v_6. □

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 $\mathcal{C}$ be the set of all canonical paths on G. For each $P \in \mathcal{C}$, let u, v, t_d, t_a, b, and 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 $l = \langle x, t_d, t_a, b, p\rangle$ in either $L_{out}(u)$ or $L_{in}(v)$ as follows:*

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

In short, in a *TTL* index, the label sets $L_{in}(v)$ and $L_{out}(v)$ 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 $L_{in}(v)$ (resp. $L_{out}(v)$) as the *in-label set* (resp. *out-label set*) of v, and each label in $L_{in}(v)$ (resp. $L_{out}(v)$) as an *in-label* (resp. *out-label*) of v. As an example, Table 2 shows the label sets of v_2 and v_4 in Figure 1, assuming a node order $o(v_1) < o(v_2) < \ldots < o(v_6)$.

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 v, and P be the result of q. Then, $L_{out}(u)$ contains a label $\langle w, t_d, t_a, b, p\rangle$, and $L_{in}(v)$ contains a label $\langle w', t'_d, t'_a, b', p'\rangle$, 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, $L_{out}(u)$ contains a label pertinent to P.*

2. *$w' = u$, and t'_d, t'_a, b', p' equal P's departure time, arrival time, vehicle, and pivot, respectively. That is, $L_{in}(v)$ contains a label pertinent to P.*
3. *$w = w'$, $t_a \leq t'_d$, and t_d (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. from 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_3 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 $\langle v_1, 5, 9, b_2, v_3\rangle$, 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 $\langle v_1, 9, 10, b_2, null\rangle$, which captures the sub-path of P from v_1 to v_2. □

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* index 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.* □

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 (HHL)* [3], which is the state-of-the-art indexing approach for answering shortest path queries on simple graphs. In particular, *HHL* 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 *HHL* do not contain any temporal information. Compared with *HHL*, *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_{ans} corresponds to the query answer.
3. *Path Reconstruction:* This phase converts the sketch s_{ans} 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 $l_1 = \langle v, t_d, t_a, \cdot, \cdot\rangle$, 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 $l_2 = \langle u, t'_d, t'_a, \cdot, \cdot\rangle$, 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 $l_3 = \langle w, t^*_d, t^*_a, \cdot, \cdot\rangle$ and $l_4 = \langle w, t^\circ_d, t^\circ_a, \cdot, \cdot\rangle$, respectively, such that $t^*_a \leq t^\circ_d$, 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_{ans} 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 S_{uv}, then we can easily identify one corresponding to P_{ans}, 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 $L_{out}(v_4)$ and $L_{in}(v_2)$. Observe that $L_{out}(v_4)$ contains a label $l_1 = \langle v_1, 5, 9, b_2, v_3\rangle$, which can form two path sketches when combined with the labels $l_2 = \langle v_1, 9, 10, b_2, null\rangle$ and $l_3 = \langle v_1, 10, 11, b_1, null\rangle$ in $L_{in}(v_2)$. However, the sketch $\langle l_1, l_3\rangle$ is *dominated* by the sketch $\langle l_1, l_2\rangle$, in the sense that $\langle l_1, l_2\rangle$ 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 $l_1 = \langle v, t_d, t_a, \cdot, \cdot\rangle$ and $l_2 = \langle v', t'_d, t'_a, \cdot, \cdot\rangle$ 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 timestmp t'; its output is a set S_{uv} 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 : $L_{out}(u)$, $L_{in}(v)$, t, t'
output: a set S_{uv} of path sketches

```
1  initialize i = 1, j = 1, and C = ∅;
2  while i ≤ |L_out(u)| or j ≤ |L_in(v)| do
3      let l_u = ⟨w, t_d, t_a, ·, ·⟩ be the i-th label of L_out(v);
4      if [t_d, t_a] ⊈ [t, t'] then
5          increase i by one, and then goto Line 4;
6      else if w = v then
7          insert ⟨l_u, null⟩ into S_uv;
8          increase i by one, and then goto Line 4;
9      let l_v = ⟨w', t'_d, t'_a, ·, ·⟩ be the j-th tuple of L_in(u);
10     if [t'_d, t'_a] ⊈ [t, t'] then
11         increase j by one, and then goto Line 4;
12     else if w' = u then
13         insert ⟨null, l_v⟩ into S_uv;
14         increase j by one, and then goto Line 4;
15     if w = w' then
16         if t_a ≤ t'_d then
17             add ⟨l_u, l_v⟩ into S_uv;
18             increase i by one;
19         else
20             increase j by one;
21     else
22         if o(w) < o(w') then
23             increase i by one;
24         else
25             increase j by one;
26 return S_uv;
```

Specifically, *SketchGen* first scans the labels in $L_{out}(u)$, until it finds a label $l_u = \langle w, t_d, t_a, \cdot, \cdot\rangle$ with $t_d \geq t$ and $t_a \leq t'$ (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_d and arrival time t_a; after that, *SketchGen* moves on to the next label in $L_{out}(u)$ (Lines 6-8).

On the other hand, if $w \neq v$, then *SketchGen* turns its attention to $L_{in}(v)$, and scans the labels in $L_{in}(v)$ until it counters a label $l_v = \langle w', t'_d, t'_a, \cdot, \cdot\rangle$ with $t'_d \geq t$ and $t'_a \leq t'$ (Lines 9-11). If $w' = u$, then *SketchGen* adds l_v into S_{uv} as a path sketch, and moves on to the next label in $L_{in}(v)$ (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 $\langle l_u, l_v\rangle$. In particular, if $w = w'$ and $t_a \leq t'_d$, then *SketchGen* inserts $\langle l_u, l_v\rangle$ into S_{uv}, and move on to the next label in $L_{out}(u)$ (Lines 15-18). However, if $w \neq w'$ or $t_a > t'_d$, then *SketchGen* would move on to the next label in either $L_{out}(u)$ or $L_{in}(v)$ to continue the construction of path sketches. Here, *SketchGen* differentiates two cases.

Case 1: $w = w'$ but $t_a > t'_d$. In this case, *SketchGen* proceeds to the next label in $L_{in}(v)$ (Lines 19-20). The rationale is that, due to the way that we sort the labels in $L_{in}(v)$, the next label in $L_{in}(v)$ could be $l^*_v = \langle w, t^*_d, \cdot, \cdot, \cdot\rangle$ with $t_a \leq t^*_d$, 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 $L_{out}(u)$ (Lines 21-23), since the label could be $l^*_u = \langle w', \cdot, t^*_a, \cdot, \cdot\rangle$, 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 $L_{out}(u)$ cannot be combined with l_v as a path sketch, and hence, *SketchGen* would proceed to the next label in $L_{in}(v)$ instead.

In general, our discussions above are applicable for any pair of labels $l_u \in L_{out}(u)$ and $l_v \in L_{in}(v)$ encountered by *SketchGen* during its linear scans of $L_{out}(u)$ and $L_{in}(v)$. 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 $l_u \in L_{out}(u)$ that concerns v and any remaining $l_v \in L_{in}(u)$ 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 $L_{out}(v)$ or $L_{in}(v)$ (Lines 15-25).

It can be verified that *SketchGen* runs in $O(|L_{out}(u)| + |L_{out}(v)|)$ time and generates at most $O(|L_{out}(u)| + |L_{out}(v)|)$ path sketches in S_{uv}, since it linearly scans $L_{out}(u)$ and $L_{out}(v)$. 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 result 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, $s_{ans} = \langle l_u, l_v\rangle$, where l_u (resp. l_v) is either $null$ or a label in $L_{out}(u)$ (resp. $L_{in}(v)$). To transform s_{ans} back to P_{ans}, it suffices to convert $l_u \neq null$ (resp. $l_v \neq null$) to the canonical path P_u (resp. P_v) in G that correspond to l_u (res. 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 $l_u = \langle w, \cdot, \cdot, \cdot, p\rangle$. Let P_1 (resp. P_2) be the sub-path of P from u to p (resp. from p to w). By Lemma 4, both P_1 and P_2 are canonical paths. Then, by Definition 7, there should exists 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 $S = \langle l_1, l_2\rangle$, 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 $\langle \cdot, \cdot, \cdot, \cdot, p\rangle$ where $p \neq null$ (i.e., the label corresponds to a canonical path that contains at least two edges). Therefore, given the label sequence $S = \langle l_1, l_2\rangle$ 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*

input : a label $l = \langle \cdot, \cdot, \cdot, \cdot, p \rangle$ in *TTL*
output: the canonical path corresponding to l

```
if p = null then
    return a path that only contains the edge in G corresponding to l;
else
    l1 = the left child of l;
    l2 = the right child of l;
    P1 = PathUnfold( l1 );
    P2 = PathUnfold( l2 );
    set P = a concatenation of P1 and P2, with P1 proceeding P2;
    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, *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_2 to v_4 in Figure 1, with starting time $t = 8$ and ending time $t' = 13$. By a linear scan of the labels in $L_{out}(v_2)$ and $L_{in}(v_4)$ (in Table 2), we obtain two path sketches: (i) $\langle l_1, l_2 \rangle$, with $l_1 = \langle v_1, 8, 10, b_1, null \rangle$ in $L_{out}(v_2)$ and the label $l_2 = \langle v_1, 10, 13, null, v_2 \rangle$ in $L_{in}(v_4)$, (ii) $\langle null, l_3 \rangle$, with $l_3 = \langle v_2, 11, 13, b_2, v_6 \rangle$ in $L_{in}(v_4)$. Then, we set $s_{ans} = \langle null, l_3 \rangle$, since $\langle null, l_3 \rangle$ 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 $P_{ans} = \langle \langle v_2, v_6, 11, 12, b_2 \rangle, \langle v_6, v_4, 12, 13, b_2 \rangle \rangle$. □

The following corollary shows the correctness of Algorithm 2.

COROLLARY 1. *Given a label l in 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, *TTL* answers any SDP query from u to v in $O(|L_{out}(u)| + |L_{in}(v)| + k)$ time, where k is the number of nodes on the query result. Furthermore, *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 $L_{out}(u)$, $L_{in}(v)$, 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_{ans} with the earliest arrival time. After that, we unfold s_{ans} 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_{ans} with the latest departure time. Finally, we use Algorithm 2 to convert s_{ans} 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 *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 *TTL* corresponds to a canonical path in G, and vice versa. Therefore, if we are to build a *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 *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_d 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 $t_\dashv$ be the largest timestamp in T_d. Then, any earliest arrival path (EAP) P from u to a node v with departure time $t_\dashv$ is a non-dominated path.* □

Based on Observation 1, we can compute all non-dominated paths that start from u at time $t_\dashv$, by deriving all EAPs from u at time $t_\dashv$. This derivation can be performed with a modified version of Dijkstra's algorithm [10], 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 $t_\dashv$, 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 $e = \langle x, y, t_d, t_a, \cdot \rangle$ 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 $t_d \geq t_x$ and $t_a < t_y$, we set $t_y = t_a$, and insert into H all outgoing edges of y with departure time no sooner than t_a. 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 $t_\dashv$).

Now suppose that we apply the temporal Dijkstra's algorithm to compute all EAPs that start from u at time $t^*_\dashv$, where $t^*_\dashv$ denotes the second largest timestamp in T_d. We have another observation as follows.

OBSERVATION 2. *Then, any EAP P from u with departure time $t^*_\dashv$ is a non-dominated path, if it is not dominated by any EAP from u with departure time $t_\dashv$.* □

Given any EAP from u to v with departure time $t^*_\dashv$, 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 $t_\dashv$. 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 starting at 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 $\langle u_1, t_d, t_a, b, p\rangle$ into $L_{in}(v)$ (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 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 $L_{in}(v)$ 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_v, then *IndexBuild* examines whether the current $L_{in}(v)$ contains a label $l = \langle u_1, \cdot, t_a, \cdot, \cdot\rangle$ with $t_a \leq t_v$ (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 $L_{in}(v)$ 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.

Algorithm 3: *IndexBuild*

input : G and a node order o
output: a *TTL* index $\mathcal{L}$

1 let $G_1 = G$;
2 **for** $i = 1, 2, \ldots, n$ **do**
3 let u_i be the node with $o(u_i) = i$;
4 let T_d be a set containing the departure timestamps of all outgoing edges of u in G_i;
5 **for each** $t_d \in T_d$ *in descending order* **do**
6 create a hash table A that maps each node v to a tuple $\langle t_v, b_v, p_v\rangle$, where t_v is the earliest arrival time at v from u (initialized as $+\infty$), b_v is a vehicle (initialized as a placeholder $\star$), and p_v is a pivot node (initialized as $null$);
7 create a min-heap H that (i) accepts entries of the form $\langle e, p\rangle$, where e is an edge, and p is a node, and (ii) sorts entries in ascending order of the arrival time of their edges;
8 **for each** *outgoing edge e' of u with departure time t_d* **do**
9 insert an entry $\langle e', null\rangle$ into H;
10 **while** H *is not empty* **do**
11 remove the top entry $\langle e, p\rangle$ from H;
12 let $\langle x, y, t_d, t_a, b\rangle$ denote e;
13 retrieve the tuple $\langle t_x, b_x, p_x\rangle$ associated with x in A;
14 retrieve the tuple $\langle t_y, b_y, p_y\rangle$ associated with y in A;
15 **if** $t_x = +\infty$ *or* $t_d \geq t_x$ *and* $t_a < t_y$ **then**
16 set $t_y = t_a$ and $p_y = p$;
17 **if** $b_y = \star$ **then**
18 set $b_y = b$;
19 **else if** $b_y \neq b$ **then**
20 set $b_y = null$;
21 **for each** *outgoing edge e^* of y* **do**
22 let $\langle \cdot, \cdot, \cdot, \cdot, b^*\rangle$ denote e^*;
23 **if** $b^* \neq b$ **then**
24 set $b^* = null$;
25 **if** $p = null$ *or* $o(p) < o(y)$ **then**
26 insert $\langle e^*, p\rangle$ into H;
27 **else**
28 insert $\langle e^*, y\rangle$ into H;
29 **for each** *node v in G_i* **do**
30 let $\langle t_v, b_v, p_v\rangle$ be the tuple in A corresponding to v;
31 **if** *there is no* $\langle u, \cdot, t_a, \cdot, \cdot\rangle \in L_{in}(v)$ *with* $t_a \leq t_v$ **then**
32 **if** *there does not exist* $\langle u_j, t'_d, t'_a, \cdot, \cdot\rangle \in L_{out}(u_i)$ *and* $\langle u_j, t^*_d, t^*_a, \cdot, \cdot\rangle \in L_{in}(v)$, *such that* $t'_a < t^*_d$ *and* $[t'_d, t^*_a] \subset [t_d, t_v]$ **then**
33 insert into $L_{in}(v)$ a label $\langle u_i, t_d, t_v, b_v, p_v\rangle$;
34 repeat Lines 4-33 to construct labels in the out-label set of each node, using the reverse version of temporal Dijkstra's algorithm;
35 remove v from G_i, and let G_{i+1} denote the resulting graph;
36 return the collection of all label sets as a *TTL* index $\mathcal{L}$;

Based on the LDPs computed, *IndexBuild* identifies all canonical paths ending at u_1, and constructs labels in other nodes' out-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 $\mathcal{S}$ 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_i instead of G, they may not all be non-dominated paths in G. To identify the paths in $\mathcal{S}$ that are non-dominated in G, *IndexBuild* processes each path $P \in \mathcal{S}$ as follows. Assume that P starts from u_i at time t_d and ends at a node v at time t_a. If $L_{out}(u_i)$ contains a label $l_1 = \langle u_j, t'_d, t'_a, \cdot, \cdot \rangle$ and $L_{in}(v)$ has a label $l_2 = \langle u_j, t^*_d, t^*_a, \cdot, \cdot \rangle$, such that $t'_a < t^*_d$ and $[t'_d, t^*_a] \subset [t_d, t_a]$, then P is dominated by a path in G that goes from u_i to v via u_j. (Note 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 $\mathcal{L}$. The following Lemma 8 shows the correctness and time complexity of *IndexBuild*.

LEMMA 8. *Given G and a node order o, IndexBuild returns a TTL index in $O(n \cdot m + n^2 \cdot \log n)$ time, where m is the number of edges in G.* □

6. NODE ORDERING

As shown in Sections 2 and 5, a *TTL* index $\mathcal{L}$ on G is uniquely decided by the given node ordering o. As such, the choice of o has a profound effect on the *size* of $\mathcal{L}$, defined as

$$|\mathcal{L}| = \sum_{v \in V} \Big(|L_{in}(v)| + |L_{out}(v)| \Big). \tag{1}$$

The following example illustrates how o may affect $|\mathcal{L}|$:

EXAMPLE 5. Consider a graph G that contains n nodes $v_1, v_2, \ldots, v_n$, and contains a path P representing a vehicle route that traverses v_i in ascending order of i. For simplicity, assume that G contains no other edges except for those in P, and that n is a power of 2. Suppose that $o(v_i) = i$ for $1 \leq i \leq n$. Then, the size of the corresponding $\mathcal{L}$ is $O(n^2)$, since each v_i would be added into the label set $L_{in}(v_j)$ for all v_j with $j > i$.

Now consider that we construct the index with a different node order o, in n iterations as follows. In the first iteration, we set $o(v_{n/2}) = 1$, and then remove $v_{n/2}$ from P, obtaining two sub-paths of P. After that, in the j-th iteration ($j > 1$), we select the longest sub-path of P in our collection, and identify the node v' exactly in the middle of the sub-path; then, we set $o(v') = j$ and remove v' from the sub-path, which breaks the sub-path into two smaller sub-paths. With such an ordering, each node v' is added only into the label sets of the nodes on the sub-path from which v' is removed, and hence, the resulting $\mathcal{L}$ has a size of $O(n \log n)$. □

In what follows, we investigate the choice of o, aiming to reduce $|\mathcal{L}|$. In particular, we first present an approximation algorithm for the minimization of $\mathcal{L}$ (in Section 6.1), and then introduce a heuristic method that runs well in practice (in Section 6.2).

6.1 Approximation Algorithm

Let $\mathcal{N}$ be the set of all non-dominated paths on G. Intuitively, if a node v appears on a large number of non-dominated paths, then v is likely to be an *important* node, in which case it should be given a high rank in a node order. Our approximation algorithm exploits this intuition in selecting the node order o. To explain, we first introduce a few notations. We say that a node v *covers* a non-dominated path $P \in \mathcal{N}$, if v is on P. We use $\mathcal{N}(v)$ to denote the set of non-dominated paths in $\mathcal{N}$ that are covered by v, and we refer to $|\mathcal{N}(v)|$ as the *coverage* of v.

Let u_i ($i \in [1, n]$) denote the node with the i-th highest rank in our node order o. We first choose u_1 to be the node with the maximum coverage. After that, for $i \in [2, n]$, we decide u_i in ascending order of i, and we set u_i to the node that covers the largest number of non-dominated paths that have not been covered by $u_1, u_2, \ldots, u_{i-1}$, i.e.,

$$u_i = \arg\max_v |R_i(v)|,$$

$$\text{where } R_i(v) = \Big| \mathcal{N}(v) \setminus \cup_{j=1}^{i-1} \mathcal{N}(u_j) \Big|.$$

For example, in Figure 2a, $R_1(v_1) = 6, R_1(v_2) = 9$, and $R_1(v_3) = 6$; accordingly, $u_1 = v_2$. That is, in our choice of u_i, we take into account the nodes that are selected before u_i, and aim to maximize u_i's *residual* coverage with respect to the previously selected nodes. Note that this ordering method runs in polynomial time, since (i) $\mathcal{N}$ can be constructed in polynomial time using our algorithm in Section 5.1, and (ii) the total number of nodes in the paths in $\mathcal{N}$ is polynomial to the number of edges in G.

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 $\langle u_i, \cdot, \cdot, \cdot, \cdot \rangle$. In other words, $|I(u_i)|$ equals the number of labels that are due to u_i, and we have $|\mathcal{L}| = \sum_{i=1}^{n} |I(u_i)|$. Intuitively, $|I(u_i)|$ captures the space overhead incurred by u_i in the *TTL* index.

For any node v in G, let $T_d(v)$ (resp. $T_a(v)$) be the set that contains the departure (resp. arrival) timestamps of all outgoing (resp. incoming) edges of v, and

$$\alpha = \sum_{v \text{ in } G} (|T_d(v)| + |T_a(v)|). \tag{2}$$

The following lemma shows an upper-bound of $|I(u_i)|$ based on $|R_i(u_i)|$ and α.

LEMMA 9. $|I(u_i)| \leq \sqrt{\alpha} \cdot \sqrt{|R_i(u_i)|}$.

Let $v_1, v_2, \ldots, v_n$ be the node ordering minimizing $|\mathcal{L}|$. We define

$$R'_i(v) = \Big| \mathcal{N}(v) \setminus \cup_{j=1}^{i-1} \mathcal{N}(v_j) \Big|.$$

The influence set $I(v_i)$ of v_i can be similarly defined as the set of non-dominated paths in $R'_i(v_i)$ that either start from v_i or end at v_i. The following lemma shows a lower-bound of $|I(v_i)|$ based on $|R'_i(v_i)|$.

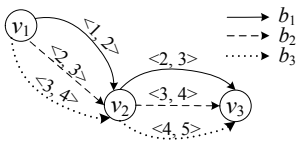
LEMMA 10. $\sqrt{|R'_i(v_i)|} \leq |I(v_i)|$.

The following lemma shows the connection between $|R_i(u_i)|$ and $|R'_i(v_i)|$:

LEMMA 11. $\sum_{i=1}^{n} \sqrt{|R_i(u_i)|} \leq 2 \cdot \sum_{i=1}^{n} \sqrt{|R'_i(v_i)|}$.

Based on Lemmas 9, 10 and 11, we show in Theorem 1 the approximation ratio of our ordering method.

THEOREM 1. *A TTL index based on a node order $u_1, u_2, \ldots, u_n$ has a size that is at most $2\sqrt{\alpha}$ times of that with the optimal node ordering. To derive such a node order, it takes $O(n^2 \cdot m)$ time and $O(n \cdot m)$ space, where m is the number of edges in the timetable graph.* □



(a) Timetable graph G'.

$\langle v_1, 1, 3, b_1, v_2\rangle$
$\langle v_1, 2, 4, b_2, v_2\rangle$
$\langle v_1, 3, 5, b_3, v_2\rangle$
$\langle v_2, 2, 3, b_1, null\rangle$
$\langle v_2, 3, 4, b_2, null\rangle$
$\langle v_2, 4, 5, b_3, null\rangle$

(b) $L_{in}(v_3)$.

$\langle v_1, null, null, r_1, v_2\rangle$
$\langle v_2, null, null, r_1, null\rangle$

(c) $L_{in}(v_3)$ compressed.

route	vehicle	v_1		v_2		v_3	
		t_a	t_d	t_a	t_d	t_a	t_d
r_1	b_1	-	1	2	2	3	-
	b_2	-	2	3	3	4	-
	b_3	-	3	4	4	5	-

(d) Timetable for r_1.

Figure 2: An example of route-based compression.

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 $e = \langle u, \cdot, t_d, \cdot, \cdot\rangle$, we employ the temporal Dijkstra's algorithm to identify all EAPs starting from u at time t_d. Notice that all these EAPs form a tree T rooted at u, such that the directed path from the root to any leave 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_d. Note that the coverage of all nodes in T can be computed in $O(n)$ time using a bottom-up scan of T.

Let $\mathcal{T}$ 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 $\mathcal{T}$. We refer to this sum as the *coverage sum* of v. We first identify u_1 as the node with the maximum coverage sum, i.e., u_1 covers the largest number of EAPs that we constructed. Then, we remove, from each EAP tree $T \in \mathcal{T}$, the subtree under u_1 (including u_1), and we update the coverage sum of each node accordingly. Subsequently, for each $i \in [2, n]$ in ascending order, we select u_i to be the node with the maximum coverage sum in the modified $\mathcal{T}$ with the subtrees under $u_1, \ldots, u_{i-1}$ removed.

The above heuristic approach runs in $O(n^2 \cdot |\mathcal{T}|)$ time, where $|\mathcal{T}|$ denotes the number of EAP trees in $\mathcal{T}$. To explain, observe that for any $i \in [1, n]$, we take $O(n \cdot |\mathcal{T}|)$ time to identify u_i, remove it and its subtrees from all EAP trees, and update the remaining nodes' coverage sums. Therefore, the total running time of the heuristic approach equals $O(n^2|\mathcal{T}|)$. Apparently, a larger $|\mathcal{T}|$ leads to a better node ordering but a larger computation overheads. In our implementation of *TTL*, we start from $\mathcal{T} = \emptyset$, then iteratively sample EAP trees from G, until the total number of edges in the EAP trees reaches $x \cdot m$, where x is a constant and m is the number of edges in G. That is, we restrict the total size of the EAP trees. Our experiments show that $x = 8$ strikes a good tradeoff between index size and preprocessing time.

7. LABEL COMPRESSION

Even with a good node order o, a *TTL* index $\mathcal{L}$ may still have a sizable number of labels and incur considerable space overheads. To reduce the space consumption of $\mathcal{L}$, we propose to *compress* the labels in $\mathcal{L}$ 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* $r_1 = \langle v_1, v_2, v_3\rangle$, 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_3, assuming that $o(v_1) < o(v_2) < o(v_3)$. 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 $\langle v_1, null, null, r_1, v_2\rangle$ and $\langle v_2, null, null, r_1, null\rangle$, as shown in Figure 2c. That is, we *combine* the labels in $L_{in}(v_3)$ that share the same starting node, ending node, and pivot. To ensure lossless decompression, 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 $\langle v_1, null, null, r_1, v_2\rangle$ 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 $\langle v_2, null, null, r_1, null\rangle$ 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 $\langle u, null, null, r, p\rangle$, and create the timetables of r associated with u and v. The out-label sets in $\mathcal{L}$ 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: $l_1 = \langle u, t_d, t_a, null, p\rangle$, $l_2 = \langle u, t'_d, t'_a, null, p\rangle$, and $l_3 = \langle u, t^*_d, t^*_a, null, p\rangle$. 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 $l_c = \langle u, null, null, null, p\rangle$.

After that, whenever we encounter l_c 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 $o(u) < o(p) < o(v)$. Then, $L_{in}(p)$ must contain three labels $\langle u, t_d, \cdot, \cdot, \cdot\rangle$, $\langle u, t'_d, \cdot, \cdot, \cdot\rangle$, $\langle u, t^*_d, \cdot, \cdot, \cdot\rangle$, which are the left children of l_1, l_2, and l_3, respectively. In addition, $L_{in}(v)$ should have three labels $\langle p, \cdot, t_a, \cdot, \cdot\rangle$, $\langle p, \cdot, t'_a, \cdot, \cdot\rangle$, $\langle p, \cdot, t^*_a, \cdot, \cdot\rangle$ 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 $L_{in}(p)$ and $L_{in}(v)$, based on the pivot information in l_c.

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 $\langle u', null, null, null, p'\rangle$. 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 $\langle u, null, null, null, p\rangle$, then we create a tuple $\langle u, v, p\rangle$ (resp. $\langle v, u, p\rangle$). 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., $\langle u, v, p\rangle$ and $\langle u, p, \cdot\rangle$. In addition, we assign to each node $\langle u, v, p\rangle$ a weight that equals $c-1$, where c equals the number of labels compressed into $\langle u, v, p\rangle$. In other words, c captures the reduction of space due to $\langle u, v, p\rangle$. 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 [29] for maximum-weight independent set, which runs in $O(n' \log n' + m')$ 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 $\langle e_1, e_2, ..., e_k\rangle$ 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 $e_1, e_2, \ldots, e_j$ ($j < k$) have the same vehicle b_1, while $e_{j+1}, \ldots, e_k$ 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_d (where u and t_d 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'_d (where t'_d 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_c, which is a sequence of two tuples: $\langle u, b_1, t_d\rangle$, $\langle w, b_2, t'_d\rangle$. 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 without even 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 $s_{ans} = \langle l_u, l_v\rangle$ of the query result. After that, we invoke the *PathUnfold* algorithm to unfold the labels in s_{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., [25]).

9. RELATED WORK

A plethora of techniques [7, 8, 10, 12, 16, 21–23, 27, 28, 32, 35, 37] 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)* [16, 38], the *Contraction Hierarchies for Timetables (CHT)* [21], and *T.Patterns* [5]. 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, *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 [10] in terms of query time.

Meanwhile, *CHT* [21], 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* [5] 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 [22, 23, 27, 28, 37] 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 [22, 23, 27, 28, 37] have been proposed to accelerate path queries. However, as pointed out in [6, 8], 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 [11, 13, 15, 18, 30, 31, 33, 36] 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 [10, 12, 14, 19, 20, 24, 26–28, 34] 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 [7, 12, 21], 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 [6] for a recent survey). The classic approach for this problem is Cohen et al.'s *2-hop labelling (2HL)* [9] 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 $O(n^5)$ 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 (HHL)* [3, 4]. Compared with Cohen et al's method, *HHL* provides higher query efficiency and smaller pre-computation cost. However, approximation algorithms for *HHL* (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 10^5 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 also randomly generated, except that the starting (resp. ending) timestamp of an LDP (resp. EAP) query is always set to $-\infty$ (resp. $+\infty$).

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* [21] and *CSA* [16]. 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 $30\mu s$, 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.

dataset	$\|V\|$	$\|E\|$	avg degree
Austin	2.7K	317.5K	118.7
Madrid	4.6K	1912.2K	412.5
Budapest	5.5K	1375.2K	251.5
Salt Lake City	6.3K	329.0K	52.6
Rome	8.8K	2267.7K	258.3
Denver	9.5K	708.7K	74.5
Houston	9.8K	1111.8K	112.9
Toronto	10.8K	3295.1K	305.4
Berlin	12.8K	1965.8K	153.0
Los Angeles	15.0K	1903.2K	126.6
Sweden	51.4K	3926.5K	76.4

Table 3: Public transportation networks (K=10^3).

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, $l_{avg} \approx 1600$, and $n_{avg} \approx 39$ for SDP queries; in contrast, on *Sweden*, we have $l_{avg} \approx 775$ and $n_{avg} \approx 19$ for SDP 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 [3, 4] 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 SDP 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 in-

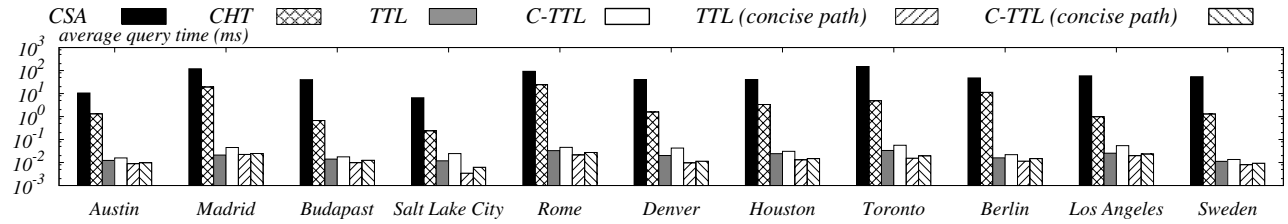


Figure 3: Average query time for SDP queries.

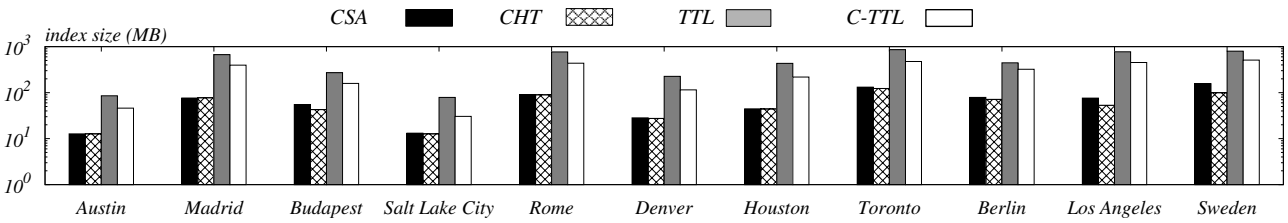


Figure 4: Index size.

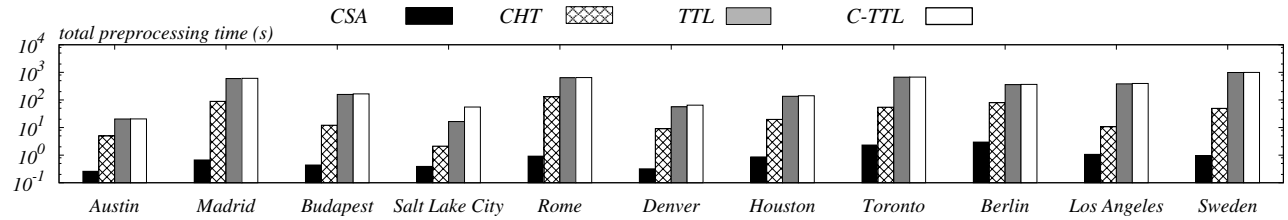


Figure 5: Total preprocessing time.

dex 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 $\Delta_i = |\mathcal{L}| - |\mathcal{L}_i|$ $(i = 1, 2, 3)$. 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 devise).

Finally, Figure 5 shows the preprocessing 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 preprocessing 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 preprocessing algorithm.

dataset	$\Delta_1/\|\mathcal{L}\|$	$\Delta_2/\|\mathcal{L}\|$	$\Delta_3/\|\mathcal{L}\|$
Austin	20.25%	33.66%	46.20%
Madrid	16.52%	30.23%	45.61%
Budapest	10.41%	35.45%	41.71%
Salt Lake City	38.00%	38.78%	61.43%
Rome	14.42%	31.85%	43.87%
Denver	23.84%	34.92%	49.23%
Houston	28.77%	32.62%	49.54%
Toronto	19.95%	32.77%	44.78%
Berlin	6.70%	23.51%	27.58%
Los Angeles	16.78%	31.68%	41.66%
Sweden	12.74%	28.52%	35.92%

Table 4: Compression ratio.

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-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.14JC1400300 from Shanghai Municipal Commission of Science and Technology, and a gift from Microsoft Research Asia.

13. REFERENCES

[1] https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds.

[2] https://sites.google.com/site/timetablelabelling/.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.

[4] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD Conference*, pages 349–360, 2013.

[5] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger. Fast routing in very large public transportation networks using transfer patterns. In *ESA*, pages 290–301, 2010.

[6] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck. Route planning in transportation networks. *MSR Technical Report*, 2014.

[7] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *ALENEX*, volume 9, 2009.

[8] R. Bauer, D. Delling, and D. Wagner. Experimental study of speed up techniques for timetable information systems. *Networks*, 57(1):38–52, 2011.

[9] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[10] K. L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of mathematical analysis and applications*, 14(3):493–498, 1966.

[11] B. C. Dean. *Continuous-time dynamic shortest path algorithms*. PhD thesis, Massachusetts Institute of Technology, 1999.

[12] D. Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.

[13] D. Delling, B. Katz, and T. Pajor. Parallel computation of best connections in public transportation networks. *Journal of Experimental Algorithmics (JEA)*, 17:4–4, 2012.

[14] D. Delling and G. Nannicini. Bidirectional core-based routing in dynamic time-dependent road networks. In *Algorithms and Computation*, pages 812–823, 2008.

[15] D. Delling, T. Pajor, and R. F. F. Werneck. Round-based public transit routing. In *ALENEX*, pages 130–140, 2012.

[16] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner. Intriguingly simple and fast transit routing. *SEA*, 7933:43–54, 2013.

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

[18] Y. Disser, M. Müller-Hannemann, and M. Schnee. Multi-criteria shortest paths in time-dependent train networks. In *SEA*, pages 347–361, 2008.

[19] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969.

[20] L. Foschini, J. Hershberger, and S. Suri. On the complexity of time-dependent shortest paths. *Algorithmica*, 68(4):1075–1097, 2014.

[21] R. Geisberger. Contraction of timetable networks with realistic transfers. In *SEA*, pages 71–82, 2010.

[22] A. V. Goldberg and C. Harrelson. Computing the shortest path: *A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[23] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *ALENEX*, pages 129–143, 2006.

[24] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41–72, 2009.

[25] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *PVLDB*, pages 768–779, 2004.

[26] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, page 10, 2006.

[27] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation*, 22:219–230, 2004.

[28] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In *WEA*, pages 189–202, 2005.

[29] J. Monnot. A simple approximation algorithm for WIS based on the approximability in k. *European Journal of Operational Research*, 171(1):346–348, 2006.

[30] M. Müller-Hannemann and M. Schnee. Paying less for train connections with MOTIS. In *ATMOS*, 2005.

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

[32] M. Müller-Hannemann, F. Schulz, D. Wagner, and C. Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90, 2007.

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

[34] G. Nannicini, D. Delling, L. Liberti, and D. Schultes. Bidirectional a^* search for time-dependent fast paths. In *SEA*, pages 334–346, 2008.

[35] G. Nannicini, D. Delling, D. Schultes, and L. Liberti. Bidirectional a* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012.

[36] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics*, 12:2–4, 2008.

[37] P. Sanders and D. Schultes. Engineering highway hierarchies. In *ESA*, pages 804–816, 2006.

[38] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.

APPENDIX

A. PROOFS

We omit the proofs of Lemma 8-10 due to the space constraint. A technical report that includes all proofs is available from [2].

Proof of Lemma 1. We prove the lemma by contradiction: Assume that there does not exist a label $\langle w, t_d, t_a, b, p\rangle$ in $L_{out}(u)$ or a label $\langle w', t'_d, t'_a, b', p'\rangle$ in $L_{in}(v)$ 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 w^* such that both the paths from u to w^* and from w^* 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. □

Proof of Lemma 2. Let $l = \langle u, t_d, t_a, b, p\rangle$ be an in-label in $L_{in}(v)$, and $l' = \langle u', t'_d, t'_a, b', p'\rangle$ 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 $\langle u, t_d, t_a, b, p\rangle \in L_{in}(v)$, 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 $L_{out}(u)$. Then, there must exist a node w, such that $\langle w, t_d, t_a^*, \cdot, \cdot\rangle \in L_{out}(u)$, $\langle w, t_d^*, t_a, \cdot, \cdot\rangle \in L_{in}(v)$, and $t_a^* \leq t_d^*$. 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. □

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, $l = \langle w, t_d, t_a, \cdot, \cdot\rangle$ be an out-label of u, and $l' = \langle w', t'_d, t'_a, \cdot, \cdot\rangle$ be an in-label of v. If l is pertinent to P_{ans} or the sub-path of P_{ans}, we have $t \leq t_d$ and $t_a \leq t'$, 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 $w \neq v$, $w' \neq u$, $t_a \leq t'_d$ 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 $t_a > t'_d$ for all pairs of l and l' where $w = w'$. This contradicts *Case 3*, which completes the proof. □

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 u. 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^o from u to p dominating P_1. This contradicts that P is a canonical path. The case for P_2 can be similarly established. □

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 $SDP(u, v, t, t')$. 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. □

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. □

Proof of Theorem 1. Let $|\mathcal{L}_{apx}|$ be the index size with the node ordering given by the approximation algorithm, and $|\mathcal{L}_{opt}|$ be the index size with the optimal node ordering. Then, we have

$$
\begin{aligned}
|\mathcal{L}_{apx}| &= \textstyle\sum_{i=1}^{n} |I(u_i)| \\
&\leq \sqrt{\alpha} \cdot \textstyle\sum_{i=1}^{n} \sqrt{|R_i(u_i)|} && \text{(By Lemma 9)} \\
&\leq 2\sqrt{\alpha} \cdot \textstyle\sum_{i=1}^{n} \sqrt{|R'_i(v_i)|} && \text{(By Lemma 11)} \\
&\leq 2\sqrt{\alpha} \cdot \textstyle\sum_{i=1}^{n} |I(v_i)| && \text{(By Lemma 10)} \\
&= 2\sqrt{\alpha} \cdot |\mathcal{L}_{opt}|.
\end{aligned}
$$

In other words, the approximation algorithm yields a $2\sqrt{\alpha}$ approximation in terms of index size.

To obtain the node order with the above 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 $\langle u, v, t_d\rangle$ (resp. $\langle u, v, t_a\rangle$), 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 (resp. 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 $\langle u, v, t_d\rangle$ (resp. $\langle u, v, t_a\rangle$). Besides, for each item in T_3, its key is a quadruple $\langle u, v, t_d, t_a\rangle$, and its value is $True$ if the non-dominated path from u to v with departure time t_d and arrival time t_a has been covered, otherwise $False$. Furthermore, 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 $O(n \cdot d_{out}(v))$, 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 node 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, end at a node w, 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 $\langle u, v', t_a\rangle$, 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 $\langle v', w, t_d\rangle$, and obtain the corresponding arrival time t_2. Finally, we search T_3 for an entry with a key $\langle u, w, t_1, t_2\rangle$, 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 $d_{in}(v') + d_{out}(v')$ time. As there are $O(n^2)$ combinations of u and w, the cost to compute the non-dominated paths covered by v' is $O(n^2 \cdot (d_{in}(v') + d_{out}(v')))$. 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 $\langle u, v, t_d\rangle$, and obtain the corresponding arrival time t_1; then, we search T_2 for an entry with a key $\langle v, w, t_a\rangle$, 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

Algorithm 4: *CPathUnfold*

input : a label $l = \langle \cdot, \cdot, \cdot, \cdot, p \rangle$ in *TTL*
output: the canonical path corresponding to l

```
1  if p = null then
2      return a path that only contains the edge in G corresponding to l;
3  else
4      l1 = the left child of l;
5      l2 = the right child of l;
6      let l1' be the original label of l1 to be computed;
7      if l1 is a label with route-based compression then
8          obtain l1' by inspecting the timetable of the route associated
           with l1;
9      else if l1 is a label with pivot-based compression then
10         obtain l1' by inspecting the left and right children of l1;
11     P1 = CPathUnfold( l1' );
12     repeat Lines 7-14 by replacing l1 with l2, l1' with l2', and P1 with
       P2;
13     set P = a concatenation of P1 and P2, with P1 proceeding P2;
14     return P;
```

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. □

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 $\langle \cdot, null, null, r, \cdot \rangle$, i.e., the route information r of l is available. Then, as explained in Section 7.1, the original 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 $\langle \cdot, null, null, null, p \rangle$. Let $l_1, l_2, \ldots, l_j$ ($j \geq 1$) be the original labels associated with l. 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. $L_{in}(v)$). As such, we can recover $l_1, l_2, \ldots, l_j$ by inspecting the labels in $L_{in}(p)$ and $L_{in}(v)$ based on the pivot information of l. After that, we can replace l with $l_1, l_2, \cdots, l_j$ 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.

v	$L_{in}(v)$	$L_{out}(v)$
v_2	$\emptyset$	$\emptyset$
v_1	$\emptyset$	$\langle v_2, 1, 2, b_1, null \rangle$ $\langle v_2, 2, 3, b_2, null \rangle$ $\langle v_2, 3, 4, b_3, null \rangle$
v_3	$\langle v_2, 2, 3, b_1, null \rangle$ $\langle v_2, 3, 4, b_2, null \rangle$ $\langle v_2, 4, 5, b_3, null \rangle$	$\emptyset$

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 $o(v_3) = 3$.

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 $L_{in}(v_3)$. 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. This 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.

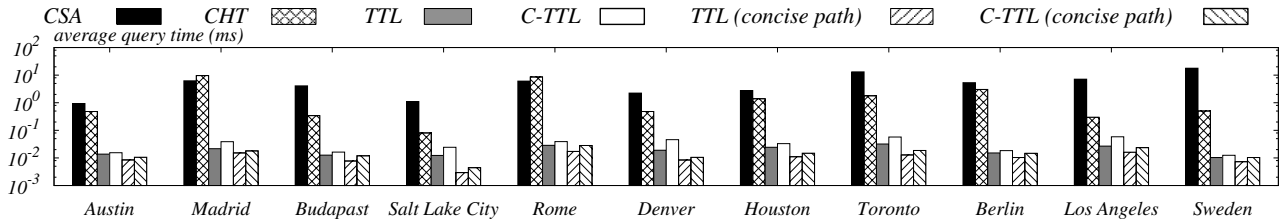


Figure 6: Average query time for EAP queries.

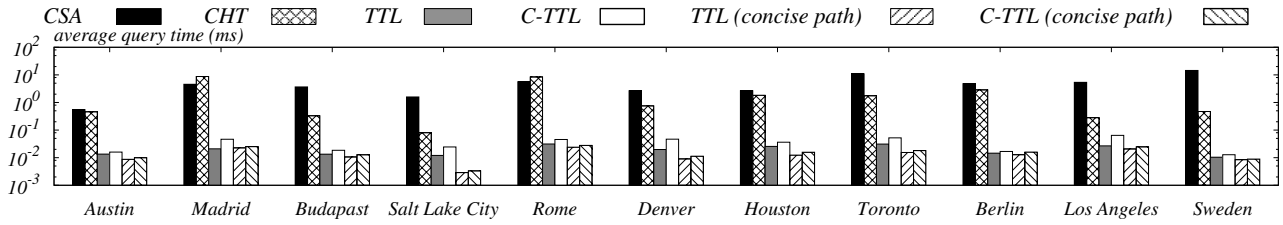


Figure 7: Average query time for LDP queries.

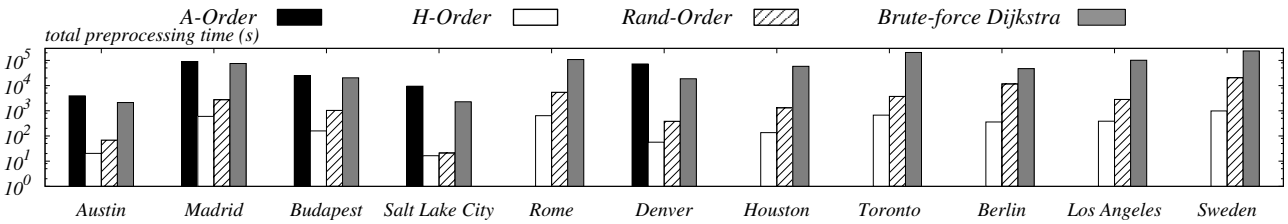


Figure 8: Total preprocessing time.

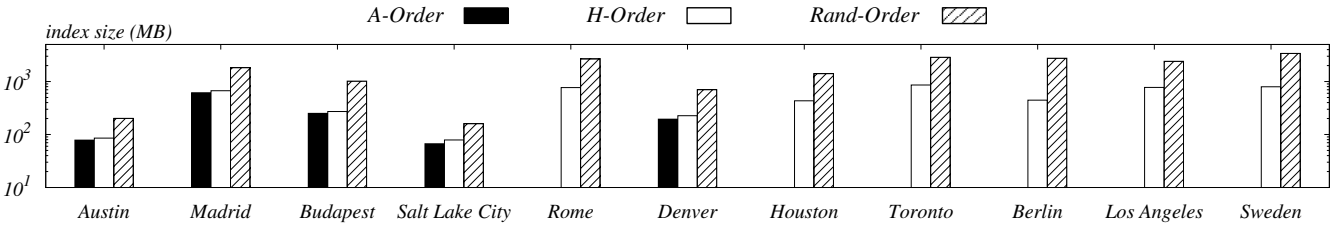


Figure 9: Index size.

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.