

# Finding Time-Dependent Shortest Paths over Large Graphs

Bolin Ding  
The Chinese University of  
Hong Kong  
blding@se.cuhk.edu.hk

Jeffrey Xu Yu  
The Chinese University of  
Hong Kong  
yu@se.cuhk.edu.hk

Lu Qin  
The Chinese University of  
Hong Kong  
lqin@se.cuhk.edu.hk

## ABSTRACT

The spatial and temporal databases have been studied widely and intensively over years. In this paper, we study how to answer queries of finding the best departure time that minimizes the total travel time from a place to another, over a road network, where the traffic conditions dynamically change from time to time. We study a generalized form of this problem, called the time-dependent shortest-path problem. A time-dependent graph  $G_T$  is a graph that has an edge-delay function,  $w_{i,j}(t)$ , associated with each edge  $(v_i, v_j)$ , to be stored in a database. The edge-delay function  $w_{i,j}(t)$  specifies how much time it takes to travel from node  $v_i$  to node  $v_j$ , if it departs from  $v_i$  at time  $t$ . A user-specified query is to ask the minimum-travel-time path, from a source node,  $v_s$ , to a destination node,  $v_e$ , over the time-dependent graph,  $G_T$ , with the best departure time to be selected from a time interval  $T$ . We denote this user query as  $LTT(v_s, v_e, T)$  over  $G_T$ . The challenge of this problem is the added complexity due to the time dependency in the time-dependent graph. That is, edge delays are not constants, and can vary from time to time. In this paper, we propose a novel algorithm to find the minimum-travel-time path with the best departure time for a  $LTT(v_s, v_e, T)$  query over a large graph  $G_T$ . Our approach outperforms existing algorithms in terms of both time complexity in theory and efficiency in practice. We will discuss the design of our algorithm, together with its correctness and complexity. We conducted extensive experimental studies over large graphs and will report our findings.

## 1. INTRODUCTION

Due to the increasing interest in the dynamic management of transportation systems, there are needs to find shortest paths over a large graph (e.g., a road network), where the weights (or delays) associated with edges dynamically change over time (*time-dependency*). Transportation systems, which can provide real-time traffic information (used to calculate edge delays) to users, include the Vehicle Information and Communication System<sup>1</sup> (VICS) and

the European Traffic Message Channel<sup>2</sup> (TMC). The former operates in Japan and the latter operates in most European countries, North America, and Australia. Together with road networks available as large graphs<sup>3</sup>, when such traffic information is available and the (periodical) traffic patterns are known over a long time period, it becomes possible to provide users with services, such as “how to travel from a place in a city to another place in another city as fast as possible”, by taking “rush hour” into consideration.

Consider tourism as an application. Suppose a group of people wants to visit several places in several cities. When such road traffic information is available, the group wants to know whether they can travel to the next place *faster* (*spending less travel time on the way*), if they depart from a place later to avoid rush hour. In a similar fashion, consider a logistic company that delivers products for their customers using trucks. A truck may travel to a place with less travel time, if it stays somewhere for some time, say 3 hours. In this case, the company can utilize the 3 hours to deliver products to nearby customers where possible with this truck.

Assume a road network is stored as a large graph with the traffic information in a database. Such a query can be specified as follows. Given a source  $v_s$  and a destination  $v_e$ , over the graph, and a time window  $T$  for consideration of departure from  $v_s$ , find the best time within  $T$  to depart from  $v_s$ , and identify the path along which one can arrive at  $v_e$  with the minimum travel time.

In this paper, we study the generalized form of this query, called *time-dependent shortest-path (TDSP) problem*: to find the optimal path (with the minimum travel time) from a source to a destination, over a *time-dependent graph*, when the starting time (departure time from the source) is selected from a user-given starting-time interval. The time-dependent graph is a graph that has an edge-delay (travel time from  $v_i$  to  $v_j$ ) function  $w_{i,j}(t)$ , w.r.t. departure time  $t$  from  $v_i$ , for each edge  $(v_i, v_j)$ . TDSP problem was studied to either find approximate answers with discrete-time approaches [1, 2] or find optimal answers with continuous-time approaches [20, 15].

We focus on finding optimal answers for the TDSP problem using a continuous-time approach with less time/space complexity. We consider a specific class of graphs, called *FIFO* time-dependent graphs (refer to Section 5.1), as well as general time-dependent graphs. Our approach can handle arbitrary edge-delay functions, and allows waiting on nodes in order to minimize the travel time.

**Contributions of this paper:** (1) We propose a novel algorithm to find optimal answers for the TDSP problem. Our algorithm can handle both undirected and directed time-dependent graphs, and both *FIFO* and non-*FIFO* time-dependent graphs. (2) We show that the time complexity of our algorithm is  $O((n \log n + m)\alpha(T))$  and

<sup>1</sup><http://www.vics.or.jp/english/index.html/>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

EDBT'08, March 25–30, 2008, Nantes, France.

Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...\$5.00.

<sup>2</sup><http://www.tmcforum.com/>

<sup>3</sup><http://maps.google.com/>

the space complexity is  $O((n + m)\alpha(T))$ , where  $n$  is the number of nodes,  $m$  is the number of edges, and  $\alpha(T)$  is the cost required for each function (defined in interval  $T$ ) operation. Our algorithm can be used to handle large time-dependent graphs. (3) We discuss storage model and implementation, and show that our approach can be easily implemented in a database system. (4) We conducted extensive performance studies, and we show that our algorithm outperforms existing solutions in terms of efficiency and effectiveness.

**Organization:** Section 2 gives the problem statement. Section 3 introduces existing solutions to the time-dependent shortest-path problem. We give an overview of our algorithm in Section 4, and give the details in Section 5, including a running example, discussions on the correctness and time/space complexity of our algorithm, implementation details, and how to handle non-FIFO graphs. We give the experimental results in Section 6. Section 7 discusses the related work. Finally, we conclude our paper in Section 8.

## 2. PROBLEM DEFINITION

**Definition 2.1: (Time-Dependent Graph)** A time-dependent graph is defined as  $G_T(V, E, W)$  (or  $G_T$  for short):  $V = \{v_i\}$  is a set of nodes;  $E \subseteq V \times V$  is a set of edges;  $W$  is a set of positive-valued functions. For every edge  $(v_i, v_j) \in E$ , there is a function  $w_{i,j}(t) \in W$ , where  $t$  is a time variable in a time domain  $\mathcal{T}$ . An edge-delay function  $w_{i,j}(t)$  specifies how much time it takes to travel from  $v_i$  to  $v_j$ , if departing  $v_i$  at time  $t$ .  $\square$

In this paper, we concentrate on finding the *least total travel time* (LTT) from source node  $v_s$  to destination node  $v_e$  when the *starting time*  $t$  (departure time from the source), can be selected in a user-given *starting-time interval*  $T = [t_s, t_e] \subseteq \mathcal{T}$ . Such a query is called an *LTT query*, denoted as  $\text{LTT}(v_s, v_e, T)$ .

Note the *travel time* is the *arrival time* minus the *starting time*. In order to find LTT, we allow *waiting time*, denoted as  $\varpi(v_i)$ , at each node  $v_i$ . That is, when arriving at node  $v_i$ , we can wait for a time period  $\varpi(v_i)$  if LTT can be minimized. Below, let  $\text{arrive}(v_i)$  and  $\text{depart}(v_i)$  denote the *arrival time* at node  $v_i$  and the *departure time* from node  $v_i$ , respectively. For each node  $v_i$ , we have

$$\text{depart}(v_i) = \text{arrive}(v_i) + \varpi(v_i). \quad (1)$$

Let  $p = (v_1, v_2)(v_2, v_3) \cdots (v_{k-1}, v_k)$  be a fixed path with waiting time  $\varpi(v_i)$  at node  $v_i$ . For a fixed starting time  $t$ ,

$$\text{arrive}(v_1) = t \quad (2)$$

$$\text{arrive}(v_2) = \text{depart}(v_1) + w_{1,2}(\text{depart}(v_1))$$

...

$$\text{arrive}(v_k) = \text{depart}(v_{k-1}) + w_{k-1,k}(\text{depart}(v_{k-1})) \quad (3)$$

$$g_p(t) = \text{arrive}(v_k). \quad (4)$$

$g_p(t)$  above is the *arrival-time function*, representing the arrival time from  $v_1$  to  $v_k$  along path  $p$ , possibly waiting at some nodes on this path, if departing from  $v_1$  at starting time  $t$ . The *travel-time function* along path  $p$  is thus  $g_p(t) - t$ . We formally define the time-dependent shortest-path (TDSP) problem as follows.

**Definition 2.2: (TDSP Problem)** Given a time-dependent graph  $G_T(V, E, W)$  and an LTT Query  $\text{LTT}(v_s, v_e, T)$ , where  $v_s, v_e \in V$ , and  $T \subseteq \mathcal{T}$  is a starting-time interval, the *Time-Dependent Shortest-Path (TDSP) problem* is to minimize LTT:

$$g_{p^*}(t^*) - t^* = \min_{p, \varpi(\cdot), t} \{g_p(t) - t\} \quad (5)$$

finding a  $v_s$ - $v_e$  path  $p^*$  with waiting time  $\varpi^*(v_i)$  at  $v_i$ , along which the best starting time  $t^*$  results in the minimum travel time  $g_p(t) - t$  among all starting times  $t \in T$  and over all  $v_s$ - $v_e$  paths  $p$ 's.  $\square$

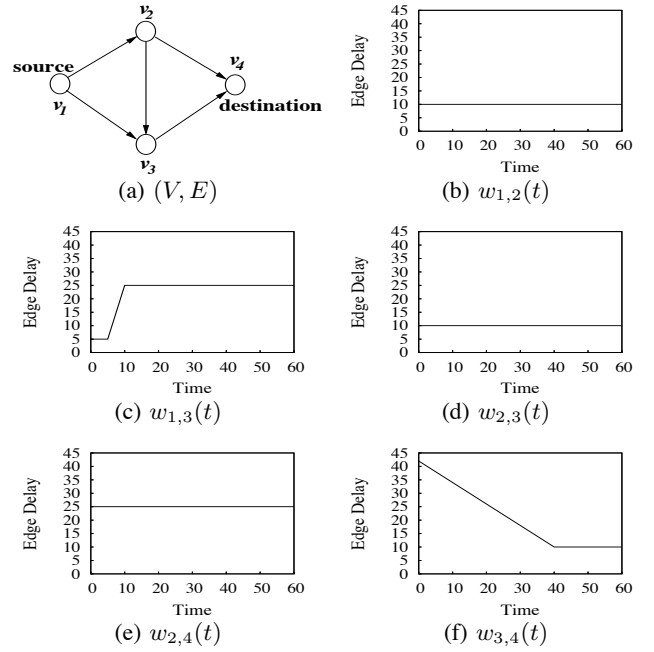


Figure 1: A Time-Dependent Graph  $G_T(V, E, W)$

**Example 2.1:** A road network can be modelled as a time-dependent graph  $G_T(V, E, W)$  in Fig. 1. Fig. 1 (a) shows its graph structure  $(V, E)$ , with four nodes and five edges. The edge-delay functions for the edges,  $(v_1, v_2)$ ,  $(v_1, v_3)$ ,  $(v_2, v_3)$ ,  $(v_2, v_4)$ , and  $(v_3, v_4)$ , are shown in Fig. 1 (b), (c), (d), (e), and (f), respectively.

For query  $\text{LTT}(v_1, v_4, [0, 60])$ ,  $p^* = (v_1, v_2)(v_2, v_3)(v_3, v_4)$  is the optimal  $v_1$ - $v_4$  path. Along  $p^*$  with no waiting time required at any node, the best starting time  $t^* = 20$  results in the minimum travel time  $g_{p^*}(t^*) - t^* = 30$  (which will be further explained in Section 5.3 as a running example of our solution).  $\square$

In the following part, we focus on the TDSP problem, i.e., for query  $\text{LTT}(v_s, v_e, T)$  over a time-dependent graph  $G_T(V, E, W)$ , finding the optimal  $v_s$ - $v_e$  path  $p^*$ , waiting times  $\varpi^*(\cdot)$ , and starting time  $t^*$ , to minimize LTT. Note  $T$  is a continuous time interval, and  $t^*$  can be any time point within this interval.

## 3. EXISTING SOLUTIONS

In this section, we discuss three types of algorithms for the TDSP problem, to answer query  $\text{LTT}(v_s, v_e, T)$  over a time-dependent graph  $G_T(V, E, W)$ . They are discrete-time algorithms [17, 2, 1], BELLMAN-FORD based algorithm [20], and extended A\* algorithm [15]. The discrete-time algorithms find an approximate LTT solution, and both BELLMAN-FORD and A\* algorithms find the optimal (minimized) LTT. The main challenge to find the optimal LTT over  $G_T(V, E, W)$  is, because edge delays are different functions of departure times, the  $v_s$ - $v_e$  path with the least total travel time changes in a complicated manner as the starting time changes.

**Discrete-Time Algorithms:** The discrete-time approaches have been well-studied. To the best of our knowledge, the most efficient one, denoted as DOT, was presented in [2]. They find approximate LTT by globally discretizing time interval into time points. In brief, given a graph  $G_T(V, E, W)$ , a discrete-time approach discretizes the starting-time interval  $T = [t_s, t_e]$  into  $k$  time points evenly, and constructs a static graph  $G'_T(V', E', W')$  by making  $k$  copies of each node and each edge, respectively. Thus,  $|V'| =$

$k|V|$ ,  $|E'| = k|E|$ , and edge delay  $W'$  is static. For each edge  $(v'_i, v'_j) \in E'$ , edge delay  $w'_{i,j}$  is equal to the value of  $w_{i,j}(t)$  on a time point. The TDSP problem on  $G_T(V, E, W)$  can be solved as a static single-source shortest path problem on  $G'_T(V', E', W')$ , whose size is enlarged  $k$  times. Its solution can be used to approximate LTT over  $G_T(V, E, W)$ .

There are two fundamental drawbacks inherent in discrete-time approaches. First, the difference between the LTT obtained using a discrete-time approach and the optimal LTT, called *LTT error*, is very sensitive to parameter  $k$ , and is unbounded. This is because the optimal starting time  $t^*$  for LTT( $v_s, v_e, T$ ) can be always between any two of the  $k$  time points, and the LTT error is generated in an accumulative way along  $v_s$ - $v_e$  paths. Second, increasing  $k$  deteriorates the efficiency of discrete-time approaches, since  $G'_T$  is  $k$  times larger than  $G_T$ .

**Bellman-Ford Based Algorithm:** Orda and Rom [20] proposed a continuous-time algorithm to solve the TDSP problem. We call it OR algorithm by taking the initials of Orda and Rom. Algorithm OR takes time-dependent graph  $G_T(V, E, W)$  and query LTT( $v_s, v_e, T$ ) as the input. It is outlined below.

- 1: **for all**  $v_l \in V$  **do**  $g_l(t) \leftarrow \infty$  for  $t \in T$ ;
- 2: **for all**  $(v_k, v_l) \in E$  **do**  $h_{k,l}(t) \leftarrow \infty$  for  $t \in T$ ;
- 3:  $g_s(t) \leftarrow t$  for  $t \in T$ ;
- 4: **repeat**
- 5:   **for all**  $(v_k, v_l) \in E$  **do**  $h_{k,l}(t) \leftarrow g_k(t) + w_{k,l}(g_k(t))$ ;
- 6:   **for all**  $v_l \in V$  **do**  $g_l(t) \leftarrow \min_{v_k \in N(v_l)} \{h_{k,l}(t)\}$ ;
- 7: **until** all functions  $g_l(t)$  are unchanged
- 8: **return** ( $t^* \leftarrow \arg\min_{t \in T} \{g_e(t) - t\}, \mathbf{p}^*$ );

OR generalizes the BELLMAN-FORD shortest-path algorithm. Let function  $g_l(t)$  be the *earliest arrival time* at node  $v_l$ , from source  $v_s$ , for starting time  $t$ , and let function  $h_{k,l}(t)$  be the *earliest arrival time* at  $v_l$ , from source  $v_s$  via edge  $(v_k, v_l)$ , for starting time  $t$ . It first initializes  $g_l(t)$  and  $h_{k,l}(t)$  functions (line 1-3), and then repeatedly updates  $g_l(t)$  and  $h_{k,l}(t)$  until they converge to the correct values (line 4-7). Finally (line 8), it returns the best starting time  $t^*$ , and the optimal  $v_s$ - $v_e$  path  $\mathbf{p}^*$ , as the answer to LTT( $v_s, v_e, T$ ).  $\mathbf{p}^*$  is constructed based on  $g_l(t)$  and  $h_{k,l}(t)$  functions (refer to [20]).

The time complexity of Algorithm OR is  $O(nm\alpha(T))$ , where  $\alpha(T)$  is the time required in a function operation in interval  $T$ ,  $n = |V|$ , and  $m = |E|$ . The high time complexity makes it infeasible for OR to work on large or dense time-dependent graphs. We outline the reasons for its high time complexity below.

OR takes a strategy of determining paths toward destination  $v_e$  while refining the arrival-time functions,  $g_i(t)$ , in the whole interval  $T$ . We call such an algorithm a *path-selection and time-refinement* approach. The *path-selection* is accomplished implicitly in line 5, attempting to arrive at  $v_l$  earlier via edge  $(v_k, v_l)$ . The *time-refinement* is done in line 6, updating arrival-time function  $g_l(t)$  using  $h_{k,l}(t)$ . The interweavement of path-selection and time-refinement in the whole interval  $T$  makes functions,  $g_l(t)$  and  $h_{k,l}(t)$ , converge slowly, possibly in  $n$  iterations of line 4-7. Actually, after some iterations,  $g_l(t)$  might have converged in a subinterval of  $T$ , but Algorithm OR cannot recognize this, and still needs to recalculate  $g_l(t)$  and  $h_{k,l}(t)$  in the whole interval  $T$ .

**A\* Algorithm:** Kanoulas et al. in [15] gave an extension to A\* algorithm for the TDSP problem. We denote it as KDXZ by taking the initials from the authors in [15]. The main idea is to maintain a priority queue  $\mathcal{Q}$  of all paths to be expanded. Let  $\mathbf{p}_k$  be a path from source  $v_s$  to a node  $v_k$ . Note: there are possibly multiple paths from  $v_s$  to  $v_k$  in  $G_T$ , and all of them may be maintained in  $\mathcal{Q}$  at the same time. Each distinct  $v_s$ - $v_k$  path  $\mathbf{p}_k$  is associated with

a function,  $f_{\mathbf{p}_k}(t) = g_{\mathbf{p}_k}(t) + d_{k,e} - t$ . Here,  $g_{\mathbf{p}_k}(t)$  is the arrival time from source  $v_s$  to  $v_k$  along path  $\mathbf{p}_k$  for starting time  $t$ ;  $d_{k,e}$  is a lower bound estimation of the travel time from  $v_k$  to destination  $v_e$ ;  $f_{\mathbf{p}_k}(t)$  is the estimated travel time from source  $v_s$  to destination  $v_e$  along path  $\mathbf{p}_k$  for starting time  $t$ . In each iteration, it picks the path  $\mathbf{p}_i$  from the priority queue  $\mathcal{Q}$  to expand, such that  $\min_t \{f_{\mathbf{p}_i}(t)\}$  is the minimum among all paths  $\mathbf{p}_k$ 's in  $\mathcal{Q}$ . Each path  $\mathbf{p}_j$ , extended from  $\mathbf{p}_i$  with one more edge  $(v_i, v_j)$ , will be added into the priority queue  $\mathcal{Q}$  for further expansion, and path  $\mathbf{p}_i$  will be deleted from  $\mathcal{Q}$ . This process will terminate when the first  $v_s$ - $v_e$  path  $\mathbf{p}_e$  is picked from  $\mathcal{Q}$ . Note KDXZ assumes no waiting is allowed.

KDXZ is also a *path-selection and time-refinement* approach. The *path-selection* is done explicitly in the path extension from  $\mathbf{p}_k$  to  $\mathbf{p}_l$ , followed by the *time-refinement* done in the computation of  $g_{\mathbf{p}_l}(t)$  and  $f_{\mathbf{p}_l}(t)$ . The *path-selection* and the *time-refinement* here are coupled even more closely than those in OR. Resultingly, in the worst case, all  $v_s$ - $v_e$  paths are enumerated, and the time/space complexity of KDXZ is exponential w.r.t. the size of  $G_T$ .

Algorithm KDXZ is efficient only when estimation can assist pruning the search space effectively, and  $v_s$  and  $v_e$  are closed to each other in graph  $G_T$ . It is difficult to find such estimation  $d_{k,e}$  in general graphs, and it is infeasible to use KDXZ to handle large time-dependent graphs, where  $v_e$  may be far away from  $v_s$ .

**Remark 3.1: (About Functions)** While discrete-time algorithms avoid the representation and operations of functions, BELLMAN-FORD based algorithm [20], A\* algorithm [15], and ours find the optimal LTT based on four basic function operations: FUNCTION INVERSE,  $f^{-1}(a) \triangleq \max\{t | f(t) = a\}$ , LINEAR COMBINATION,  $a \cdot f(t) + b \cdot g(t)$ , FUNCTION COMPOUND,  $f(g(t))$ , and MINIMUM of two functions,  $\min\{f(t), g(t)\}$ . [20] considers a general class of functions from a theoretical view, whereas [15] focuses on piecewise linear functions with the consideration of the cost to manipulate such functions. In this paper, we will show our approach can also handle a general class of functions as Algorithm OR in [20] does. Sharing the same concerns with [15], we will focus on piecewise-linear functions regarding implementations and performance studies in this paper.  $\square$

## 4. NEW DIJKSTRA BASED ALGORITHM

In the following part, we first focus on answering LTT( $v_s, v_e, T$ ) queries in an FIFO (First-In and First-Out, Definition 5.1) time-dependent graph  $G_T$ , where no waiting time is needed in optimal solutions (Theorem 5.1). We will discuss how to deal with general graphs in Section 5.7. We assume  $G_T$  is a *directed graph*. With minor changes, our algorithm can handle *undirected graphs* as well.

We propose a new algorithm by decoupling *path-selection* and *time-refinement* in the starting-time interval  $T$ . We show that answering a query LTT( $v_s, v_e, T$ ) over a graph  $G_T$  can be done in two steps. In the first step, we focus on time-refinement, i.e., for every node  $v_i \in V$ , to compute the *earliest arrival time*,  $g_i(t)$ , departing from  $v_s$  at any starting time  $t \in T$ . We call  $g_i(t)$  the  *$v_s$ - $v_i$  earliest arrival-time function* in the following part. Based on the earliest arrival-time functions computed, the best starting time  $t^*$  with the minimum  $v_s$ - $v_e$  travel time,  $g_e(t^*) - t^* = \min_t \{g_e(t) - t\}$ , can be identified. In the second step, we select one of the paths from  $v_s$  to  $v_e$ , which matches the optimal travel time  $g_e(t^*) - t^*$ .

As an example to illustrate the main ideas, consider the query LTT( $v_1, v_4, T$ ) over  $G_T$  (Fig. 1) in Example 2.1, where  $T = [0, 60]$ . In the first step, we compute the earliest arrival-time functions,  $g_1(t)$ ,  $g_2(t)$ ,  $g_3(t)$ , and  $g_4(t)$ , for the four nodes,  $v_1$ ,  $v_2$ ,  $v_3$ , and  $v_4$ . The earliest arrival-time function,  $g_4(t)$ , and its corresponding travel time function,  $g_4(t) - t$ , from source  $v_1$  to destina-

| Notation                     | Meaning   |
|------------------------------|---|
| $G_T(V, E, W)$               | time-dependent graph (or $G_T$ for short)   |
| $n, m$                       | number of nodes $ V $ , number of edges $ E $   |
| $w_{i,j}(t)$                 | edge-delay function for $(v_i, v_j) \in E$  |
| $v_s, v_e, T$                | source, destination, starting-time interval   |
| $\mathbf{p}^*$               | optimal path from $v_s$ to $v_e$  |
| $t^*$                        | optimal starting time   |
| $\varpi^*(v_i)$              | optimal waiting time at node $v_i$  |
| $g_i(t)$                     | $v_s$ - $v_i$ earliest arrival-time function  |
| $g_p(t)$                     | arrival-time function (along path $\mathbf{p}$ )  |
| $\alpha(T)$ or $\alpha( T )$ | time/space required to maintain a function or to manipulate a function operation over time interval $T$ |

**Table 1: Important Notations**

tion  $v_4$ , are shown in Fig. 2 (a) and (b), respectively. As shown in Fig. 2 (b), the least total travel time is 30, and the best starting time is  $t^* = 20$ , which is a starting time to arrive at  $v_4$  within the minimum travel time 30. In the second step, we identify the optimal path as  $\mathbf{p}^* = (v_1, v_2)(v_2, v_3)(v_3, v_4)$ . Note in Fig. 2,  $g_4(t)$  and  $g_4(t) - t$  are given in a subinterval  $[0, 30]$  of  $T = [0, 60]$ , because if starting from  $v_1$  later than 30, it will arrive at  $v_4$  later than 60, and thus some edge-delay functions are undefined.

The first step is the dominating factor in terms of computational cost. It needs to compute the earliest arrival-time function  $g_i(t)$  for every node  $v_i \in V$ , as given in Equation (6).

$$g_i(t) = \min_{v_j \in N(v_i), \varpi(v_j)} \{ (g_j(t) + \varpi(v_j)) + w_{i,j}(g_j(t) + \varpi(v_j)) \} \quad (6)$$

Here,  $N(v_i)$  is a set of neighbors of  $v_i$  that can reach  $v_i$  in graph  $G_T$ , i.e.,  $N(v_i) = \{v_j | (v_j, v_i) \in E\}$ .

The challenge of computing Equation (6) is due to the edge-delay functions. The edge delays are not constants, and can vary for different starting times. Therefore, the optimal  $v_s$ - $v_e$  path may be different for different starting time. In a continuous starting-time interval, there are infinite different starting-time values. It is challenging to select the best starting time  $t^*$  and the optimal  $v_s$ - $v_e$  path from an infinite number of possible starting times and an exponential number of  $v_s$ - $v_e$  paths, respectively.

Below, we show our solution TWO-STEP-LTT decouples the two things, namely, *path-selection* and *time-refinement*. We design a DIJKSTRA-based algorithm for the first step (time-refinement), and a linear-time algorithm for the second step (path-selection).

**Outline of Two-Step-LTT (Algorithm 1):** The main part of our two-step algorithm is given in Algorithm 1. We call it TWO-STEP-LTT. As shown in Algorithm 1, it takes four input parameters: time-dependent graph  $G_T$ , source  $v_s$ , destination  $v_e$ , and starting-time interval  $T$  (a query LTT( $v_s, v_e, T$ ) over  $G_T$ ). The first step, *timeRefinement* (Algorithm 3), computes the earliest arrival-time functions  $g_i(t)$ , for nodes  $v_i$  in  $G_T$ , in line 1. The condition in line 2 checks whether there is a path from  $v_s$  to  $v_e$ . The optimal starting time  $t^*$  is identified in line 3. For the second step, it calls *pathSelection* (Algorithm 2) to find a path  $\mathbf{p}^*$  which matches the arrival time  $g_e(t^*)$  for the best starting time  $t^*$  in line 4. Finally, it returns path  $\mathbf{p}^*$  together with the best starting time  $t^*$ . We outline the main ideas behind the two steps below.

**Dijkstra-Based Time-Refinement (Algorithm 3):** In the first step, we compute the earliest arrival-time function  $g_i(t)$ , for every node  $v_i$  in  $V$ . Like the DIJKSTRA algorithm (for the static shortest-path problem) which expands a set of nodes, we refine arrival-time functions,  $g_i(t)$ , incrementally in the given starting-time interval

---

**Algorithm 1** TWO-STEP-LTT ( $G_T(V, E, W), v_s, v_e, T$ )

---

**Input:** a time-dependent graph  $G_T$ , a query LTT( $v_s, v_e, T$ ) - source  $v_s$ , destination  $v_e$ , and starting-time interval  $T = [t_s, t_e]$ ; **Output:** optimal  $v_s$ - $v_e$  path  $\mathbf{p}^*$ , and optimal starting time  $t^*$ .

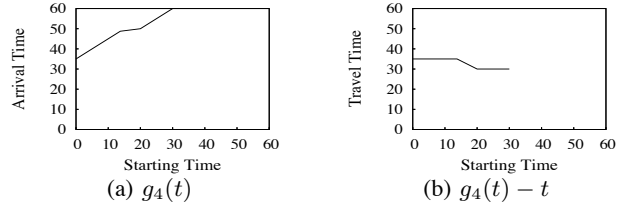
---

```

1:  $\{g_i(t)\} \leftarrow \text{timeRefinement}(G_T, v_s, v_e, T)$ ;
2: if  $\neg(g_e(t) = \infty \text{ for the entire } [t_s, t_e])$  then
3:    $t^* \leftarrow \text{argmin}_{t \in T} \{g_e(t) - t\}$ ;
4:    $\mathbf{p}^* \leftarrow \text{pathSelection}(G_T, \{g_i(t)\}, v_s, v_e, t^*)$ ;
5:   return  $(t^*, \mathbf{p}^*)$ ;
6: else return  $\emptyset$ ;

```

---



**Figure 2: An earliest arrival-time function and its corresponding travel-time function**

$T = [t_s, t_e]$ . For every node  $v_i \in V$ , let  $I_i = [t_s, \tau_i] \subseteq T$  be a starting-time subinterval, where time point  $\tau_i \in T = [t_s, t_e]$ . By "incrementally", we mean: we refine the earliest arrival-time function  $g_i(t)$  by extending  $I_i$  to a larger starting-time subinterval  $I'_i = [t_s, \tau'_i] \subseteq T$ , for  $\tau'_i > \tau_i$ , and computing  $g_i(t)$  in  $[\tau_i, \tau'_i]$ . We say function  $g_i(t)$  is *well-refined* in a starting-time subinterval  $I_i$ , if it specifies the earliest arrival time at  $v_i$  from  $v_s$  for any starting time  $t \in I_i$ . It is possible that  $I_i \neq I_j$  for  $v_i \neq v_j$ .

In our algorithm, we promise function  $g_i(t)$  is well-refined in  $I_i$  for each  $v_i \in V$ . In every iteration, we select a node  $v_i$ , and expand its starting-time subinterval from  $I_i$  to  $I'_i$ , in which function  $g_i(t)$  is well-refined. We update the arrival-time function  $g_j(t)$  for every neighbor,  $v_j$ , of node  $v_i$  in the starting-time subinterval  $I'_i - I_i = [\tau_i, \tau'_i]$ . Then we let  $I_i = I'_i$ , and repeat this time-refinement process, namely DIJKSTRA-based time-refinement, till function  $g_e(t)$  is well-refined, for destination  $v_e$ , in the entire starting-time interval  $T$ . The best starting time,  $t^* \in T$ , is identified as  $\text{argmin}_{t \in T} \{g_e(t) - t\}$ , which minimizes  $g_e(t) - t$ .

**Fast Path-Selection (Algorithm 2):** The optimal  $v_s$ - $v_e$  path  $\mathbf{p}^*$  is computed using the *pathSelection* algorithm (Algorithm 2), which takes five inputs: graph  $G_T$ , all the earliest arrival-time functions  $\{g_i(t)\}$ , the optimal starting time  $t^* \in T$ , source  $v_s$ , and destination  $v_e$ . To select the path  $\mathbf{p}^*$  from  $v_s$  to  $v_e$ , we determine the predecessor of every node on  $\mathbf{p}^*$  backward from  $v_e$  to  $v_s$  based on  $\{g_i(t)\}$  and  $t^*$ . The predecessor of  $v_j$  is determined as  $v_i$ , if  $g_j(t^*) = g_i(t^*) + w_{i,j}(g_i(t^*))$ , for  $(v_i, v_j) \in E$ . It means that the arrival time at  $v_j$ ,  $g_j(t^*)$ , is the arrival time at  $v_i$ ,  $g_i(t^*)$ , plus the edge delay from  $v_i$  to  $v_j$  (we assume there is no waiting time here).

In *pathSelection* (Algorithm 2), initially, we set  $v_j$  as destination  $v_e$ , and the optimal path  $\mathbf{p}^*$  empty (line 1-2). In the while loop, we iteratively find a predecessor  $v_i$  of  $v_j$  and add  $(v_i, v_j)$  into  $\mathbf{p}^*$  till  $\mathbf{p}^*$  reaches the source  $v_s$  (line 3-7). The correctness of *pathSelection* is straightforward. Its time complexity is  $O(m\alpha(T))$ , where  $m = |E|$  and  $\alpha(T)$  is the time required for each function operation, because each edge can be examined in line 5 at most once.

In the following, we will focus on the first step of TWO-STEP-LTT, namely, time-refinement.

**Remark 4.1:** Comparing Algorithm TWO-STEP-LTT with Algo-

---

**Algorithm 2** *pathSelection* ( $G_T(V, E, W), \{g_i(t)\}, v_s, v_e, t^*$ )

---

**Input:** a time-dependent graph  $G_T$ , the set of earliest arrival-time functions  $g_i(t)$  for all nodes  $v_i \in V$ , source node  $v_s$ , destination node  $v_e$ , and the optimal starting time  $t^*$ ;

**Output:** an optimal  $v_s$ - $v_e$  path  $p^*$  for starting time  $t^*$ .

```
1:  $v_j \leftarrow v_e$ ;  
2:  $p^* \leftarrow \emptyset$ ;  
3: while  $v_j \neq v_s$  do  
4:   for each  $(v_i, v_j) \in E$  do  
5:     if  $g_i(t^*) + w_{i,j}(g_i(t^*)) = g_j(t^*)$  then  
6:        $v_j \leftarrow v_i$ ; break;  
7:    $p^* \leftarrow (v_i, v_j) \cdot p^*$ ;  
8: return  $p^*$ ;
```

---

rithm OR,  $h_{k,l}(t)$  functions are absent in our TWO-STEP-LTT, and  $g_i(t)$  functions share the same meanings in both. The absence of  $h_{k,l}(t)$  functions in TWO-STEP-LTT does not add more complexity to the construction of  $p^*$  (Algorithm 2). As shown in Section 5 (Algorithm 3), we can use  $g_i(t)$  functions solely to answer query LTT( $v_s, v_e, T$ ) with lower time/space complexity.  $\square$

## 5. TIME-REFINEMENT

In this section, given  $G_T(V, E, W)$  and query LTT( $v_s, v_e, T$ ), we focus on the first step of TWO-STEP-LTT, i.e., time-refinement. By time-refinement, we mean to compute and refine the earliest arrival-time function  $g_i(t)$  for every node  $v_i \in V$ .

First, we introduce a special class of time-dependent graphs, called *FIFO* (First In, First Out) graphs [20]. Second, we discuss our DIJKSTRA-based algorithm *timeRefinement* (Algorithm 3) to compute the earliest arrival-time function  $g_i(t)$  for every node  $v_i$ , for answering a query LTT( $v_s, v_e, T$ ), in a *FIFO* graph  $G_T(V, E, W)$ . It is based on the incremental time-refinement of functions  $g_i(t)$  for nodes  $v_i$  in starting-time interval  $T$ . Third, we explain our algorithm using an example. Fourth, we prove the correctness of our algorithm, and give its time/space complexity. Finally, we discuss some implementation details, and show how our DIJKSTRA-based algorithm can also work on general non-*FIFO* graphs.

### 5.1 FIFO Graphs

*FIFO* property of an edge  $(v_i, v_j)$ , in  $G_T$ , suggests that if departing earlier from  $v_i$ , one arrives earlier at  $v_j$ .

**Definition 5.1: (FIFO)** Time-dependent graph  $G_T(V, E, W)$  is a *FIFO graph*, iff every edge  $(v_i, v_j)$  has *FIFO property*. An edge  $(v_i, v_j)$  has *FIFO property*, iff  $w_{i,j}(t_0) \leq t_\Delta + w_{i,j}(t_0 + t_\Delta)$  for  $t_\Delta \geq 0$ , or  $t_1 + w_{i,j}(t_1) \leq t_2 + w_{i,j}(t_2)$  for  $t_1 \leq t_2$ .  $\square$

**Theorem 5.1: (No Waiting in FIFO Graphs)** For a given query LTT( $v_s, v_e, T$ ) on a *FIFO* time-dependent graph  $G_T$ , there exists an optimal  $v_s$ - $v_e$  path  $p^*$  along which the optimal waiting time  $\varpi^*(v_i) = 0$  for every  $v_i$  on  $p^*$ .  $\square$

**Proof Sketch:** Let  $v_i$  be a node on optimal path  $p^*$ , s.t.  $\varpi^*(v_i) > 0$ , and  $v_j$  be  $v_i$ 's successor on  $p^*$ . Let  $t_i = \text{arrive}(v_i)$  and  $t_j = \text{arrive}(v_j)$  be the arrival time at  $v_i$  and  $v_j$ , respectively, along  $p^*$  for starting time  $t^*$ . From *FIFO* property, we have  $t_i + w_{i,j}(t_i) \leq (t_i + \varpi^*(v_i)) + w_{i,j}(t_i + \varpi^*(v_i)) = t_j$ . That is, the arrival time at  $v_j$  without waiting on  $v_i$  (i.e.,  $t_i + w_{i,j}(t_i)$ ) is no later than the arrival time at  $v_j$  with waiting time  $\varpi^*(v_i)$  on  $v_i$  (i.e.,  $t_j$ ). By induction, we can prove if  $\varpi^*(v_i) = 0$  for each node  $v_i$ , the travel time at  $v_e$  along  $p^*$  do not increase. Details are omitted.  $\square$

---

**Algorithm 3** *timeRefinement* ( $G_T(V, E, W), v_s, v_e, T$ )

---

**Input:** a time-dependent graph  $G_T$ , a query LTT( $v_s, v_e, T$ ) - source  $v_s$ , destination  $v_e$ , and starting-time interval  $T = [t_s, t_e]$ ;

**Output:**  $\{g_i(t) | v_i \in V\}$  - all earliest arrival-time functions.

```
1:  $g_s(t) \leftarrow t$  for  $t \in T$ ;  $\tau_s \leftarrow t_s$ ;  
2: for each  $v_i \neq v_s$  do  
3:    $g_i(t) \leftarrow \infty$  for  $t \in T$ ;  $\tau_i \leftarrow t_s$ ;  
4: Let  $Q$  be a priority queue initially containing pairs,  $(\tau_i, g_i(t))$ ,  
   for all nodes  $v_i \in V$ , ordered by  $g_i(\tau_i)$  in ascending order;  
5: while  $|Q| \geq 2$  do  
6:    $(\tau_i, g_i(t)) \leftarrow \text{dequeue}(Q)$ ;  
7:    $(\tau_k, g_k(t)) \leftarrow \text{head}(Q)$ ;  
8:    $\Delta \leftarrow \min\{w_{f,i}(g_k(\tau_k)) | (v_f, v_i) \in E\}$ ;  
9:    $\tau'_i \leftarrow \max\{t | g_i(t) \leq g_k(\tau_k) + \Delta\}$ ;  
10:  for each  $(v_i, v_j) \in E$  do  
11:     $g'_j(t) \leftarrow g_i(t) + w_{i,j}(g_i(t))$  for  $t \in [\tau_i, \tau'_i]$ ;  
12:     $g_j(t) \leftarrow \min\{g_j(t), g'_j(t)\}$  for  $t \in [\tau_i, \tau'_i]$ ;  
13:     $\text{update}(Q, (\tau_j, g_j(t)))$ ;  
14:   $\tau_i \leftarrow \tau'_i$ ;  
15:  if  $\tau_i \geq t_e$  then  
16:    if  $v_i = v_e$  then  
17:      return  $\{g_i(t) | v_i \in V\}$ ;  
18:  else  
19:     $\text{enqueue}(Q, (\tau_i, g_i(t)))$ ;  
20: return  $\{g_i(t) | v_i \in V\}$ .
```

---

Theorem 5.1 implies that, to find an optimal solution to query LTT( $v_s, v_e, T$ ) over a *FIFO* graph  $G_T$ , we can safely assume waiting time  $\varpi(v_i) = 0$  for each node  $v_i \in V$ , although waiting at nodes is allowed. The *road network model* studied in [15] is a *FIFO* graph (we will explain this in details in the appendix). Thus, waiting is not needed in road networks.

### 5.2 Time-Refinement for FIFO Graphs

In this subsection, we discuss how to process DIJKSTRA-based time-refinement for *FIFO* graphs, i.e., how to refine the earliest arrival-time function  $g_i(t)$  in the starting-time interval  $T$  for every node  $v_i$  in  $G_T$ . The *timeRefinement* algorithm is outlined in Algorithm 3. It takes four parameters as the input: time-dependent graph  $G_T(V, E, W)$ , source node  $v_s$ , destination node  $v_e$ , and starting-time interval  $T = [t_s, t_e]$ . Here, by time-refinement we mean two things: *arrival-time function refinement* of  $g_i(t)$  and *starting-time interval refinement* of the starting-time subinterval  $I_i = [t_s, \tau_i]$ , for every node  $v_i$  in  $G_T$ . Recall  $I_i = [t_s, \tau_i]$  denotes the starting-time subinterval, on which function  $g_i(t)$  is well-refined (or  $g_i(t)$  specifies the earliest  $v_s$ - $v_i$  arrival time for any starting time  $t \in I_i$ ).

Initially, for source  $v_s$ ,  $g_s(t)$  and  $\tau_s$  are initialized:  $g_s(t) \leftarrow t$  and  $\tau_s \leftarrow t_s$  (line 1). It means a trivial case: if it departs from source  $v_s$  at any time  $t_0$ , it will arrive at the same node  $v_s$  at the same time  $t_0$ , and its travel time is  $g_s(t_0) - t_0 = 0$ . For all other nodes,  $v_i \neq v_s$ , the earliest arrival-time functions,  $g_i(t)$ , are initialized as  $g_i(t) \leftarrow \infty$ , which means that they are undetermined yet, and all  $\tau_i$  are initialized as  $\tau_i \leftarrow t_s$  (line 2-3). For each node  $v_i \in V$ ,  $g_i(t)$  is ensured to be the earliest arrival time (well-refined) in  $I_i = [t_s, \tau_i]$ , which is a *loop invariant* in *timeRefinement* algorithm. Note: initially  $I_i$  is an empty subinterval.

Our algorithm uses a priority queue,  $Q$ , which initially contains pairs  $(\tau_i, g_i(t))$ 's for all nodes  $v_i \in G_T$  in the ascending order of  $g_i(\tau_i)$ . The top pair in  $Q$  is  $(\tau_s, g_s(t))$  initially. The while statement (line 5-19) conducts time-refinement for every node  $v_i$

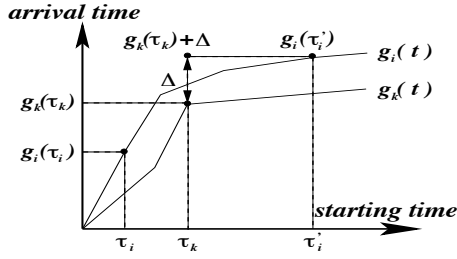


Figure 3: Starting-Time Interval Refinement

in  $G_T$ . It is ensured that the earliest arrival-time function  $g_i(t)$  is well-refined in the starting-time subinterval  $I_i = [t_s, \tau_i]$  for node  $v_i$ . The algorithm will terminate if  $g_e(t)$ , for destination  $v_e$ , is well-refined in the entire interval  $T$  (line 17), or  $Q$  contains no more than one pair (line 5). In every iteration in the while loop, the starting-time interval refinement is conducted in line 6-9 and line 14, and the arrival-time function refinement is conducted in line 10-13.

Next, we discuss starting-time interval refinement and arrival-time function refinement in details.

**Starting-Time Interval Refinement:** In every iteration, it first dequeues the top pair from  $Q$ , denoted as  $(\tau_i, g_i(t))$  (line 6). After dequeuing, it will use the current top pair in  $Q$ , denoted as  $(\tau_k, g_k(t))$ , as the basis for starting-time interval refinement (line 7 - The operation  $\text{head}(Q)$  retrieves the top pair but does not dequeue it from  $Q$ ). Therefore,  $g_i(\tau_i)$  is the earliest arrival time from source node  $v_s$ , followed by  $g_k(\tau_k)$ , among all pairs  $(\tau_l, g_l(t))$ 's in  $Q$ .

It is important to note that for any node  $v_f$  (except  $v_i$ ), if the starting time is taken in  $[\tau_f, t_e]$ , it is impossible to arrive at  $v_f$  before the arrival time  $g_k(\tau_k)$ , from source node. The two reasons are given below. Let  $(\tau_f, g_f(t))$  be in  $Q$  for node  $v_f \in V$ , and it arrives at node  $v_f$  at the arrival time  $g_f(\tau_f)$  for starting time  $\tau_f$ . First,  $g_l(\tau_f) \geq g_k(\tau_k)$ , because the sorting order in  $Q$ . Second, graph  $G_T$  is *FIFO*, and thus it arrives at  $v_f$  no earlier than time  $g_k(\tau_k)$ , if the starting time is taken in  $[\tau_f, t_e]$ . It can be formally proved based on the choices of  $(\tau_i, g_i(t))$  and  $(\tau_k, g_k(t))$  in  $Q$ , and the *FIFO* property of  $G_T$  (refer to Section 5.4).

Fix node  $v_i$ , and consider an edge  $(v_f, v_i) \in E$  at time  $g_k(\tau_k)$ . If starting time  $t$  is taken in  $[\tau_f, t_e]$ , from the above discussion, it arrives at  $v_f$  no earlier than time  $g_k(\tau_k)$ . Suppose that it arrives at  $v_f$  at time  $g_k(\tau_k)$ . The minimum travel time from  $v_f$  to  $v_i$  can be computed as  $\Delta \leftarrow \min\{w_{f,i}(g_k(\tau_k)) | (v_f, v_i) \in E\}$  (line 8). Therefore, due to the *FIFO* property of  $G_T$ , next earliest possible arrival time from  $v_s$  to  $v_i$  via any edge  $(v_f, v_i)$  is  $g_k(\tau_k) + \Delta$ , if starting time  $t \geq \tau_f$ . We attempt to find the latest starting time  $t$  that satisfies  $g_i(t) \leq g_k(\tau_k) + \Delta$ , and set it as  $\tau'_i$  (line 9). With the choice of  $\tau'_i$ , we can prove that (refer to Section 5.4) function  $g_i(t)$  is well-refined in  $[t_s, \tau'_i]$ , i.e.,  $g_i(t)$  is the earliest arrival time from  $v_s$  to  $v_i$  for starting time  $t \in [t_s, \tau'_i]$ , because (the intuition)  $g_i(t) \leq g_k(\tau_k) + \Delta$  for  $t \in [\tau_i, \tau'_i]$ . We emphasize that in the previous iteration,  $g_i(t)$  is ensured to be well-refined in  $I_i = [t_s, \tau_i]$ , and it is now ensured in  $I'_i = [t_s, \tau'_i]$ , where  $I_i \subset I'_i$ . Let  $\tau_i \leftarrow \tau'_i$  and  $I_i \leftarrow I'_i$  (line 14). It is what we call starting-time interval refinement. Fig. 3 illustrates the relationships between starting times and arrival times in the starting-time interval refinement.

As discussed above, the starting-time subinterval  $I_i$  for the dequeued node  $v_i$  is enlarged, while its earliest arrival-time function,  $g_i(t)$ , remains unchanged. Next, we discuss how to update the arrival-time function  $g_j(t)$  for a node  $v_j$ , when its incoming neighbor  $v_i$ 's starting-time interval is refined ( $(v_i, v_j) \in E$ ).

**Arrival-Time Function Refinement:** As shown above, the arrival-

time function  $g_i(t)$ , for node  $v_i$ , is well-refined as the earliest  $v_s$ - $v_i$  arrival-time function in both the original starting-time subinterval,  $I_i = [t_s, \tau_i]$ , and the enlarged one,  $I'_i = [t_s, \tau'_i]$ , in the previous and the current iterations, respectively. It can then be used to refine arrival-time functions,  $g_j(t)$ , in starting-time subinterval  $[\tau_i, \tau'_i]$  for all of  $v_i$ 's outgoing neighbors  $v_j$  ( $(v_i, v_j) \in E$ ). It is done in line 10-13. First, it computes the arrival time  $g'_j(t)$  at  $v_j$  via edge  $(v_i, v_j)$  for starting time  $t \in [\tau_i, \tau'_i]$  (line 11). Then  $g_j(t)$  is refined as  $\min\{g_j(t), g'_j(t)\}$  on interval  $[\tau_i, \tau'_i]$  (line 12). We only refine  $g_j(t)$  on  $[\tau_i, \tau'_i]$ , because we have refined it with  $g_i(t)$  on  $[t_s, \tau_i]$  already in previous iterations.  $Q$  is updated for node  $v_j$  with its newly refined arrival-time function  $g_j(t)$  (line 13).

**Terminating Condition:** After arrival-time function refinement,  $\tau_i$  is set as  $\tau'_i$  (line 14). If in the entire interval  $T = [t_s, t_e]$ , for node  $v_i$ ,  $g_i(t)$  has been well-refined, as specified in the condition " $\tau_i \geq t_e$ " (line 15), the algorithm further checks whether  $v_i$  is the destination. If true, it terminates in line 17. If  $g_i(t)$  has not been well-refined in the entire starting-time interval, pair  $(\tau_i, g_i(t))$  is enqueued back into  $Q$  (line 19) for further time refinement.

Note: the while loop terminates when there is only one pair left in the priority queue (line 5). Let the last pair be  $(\tau_i, g_i(t))$  for node  $v_i$ . There is no need to further refine it for the following reason. The starting-time subintervals  $I_j$  and earliest arrival-time functions  $g_j(t)$  of all other nodes  $v_j$  have already been well refined. So  $g_i(t)$  has been refined by every well-refined  $g_j(t)$ , if  $(v_j, v_i) \in E$ . Therefore,  $g_i(t)$  in the starting-time interval  $[\tau_i, t_e]$  also specifies the earliest arrival time, if  $g_i(t) \neq \infty$  for  $t \in [\tau_i, t_e]$ .

In summary, for source  $v_s$ , the *timeRefinement* algorithm initializes  $g_s(t) \leftarrow t$ , which becomes the starting point to refine its own starting-time interval, and to refine the earliest arrival-time functions for other nodes. The starting-time interval refinement and the arrival-time function refinement repeat in every iteration.

### 5.3 A Running Example

Reconsider Example 2.1 to compute the query  $\text{LTT}(v_1, v_4, T = [0, 60])$ , over  $G_T$  (Fig. 1). Algorithm 3 takes  $G_T$ ,  $v_s = v_1$ ,  $v_e = v_4$ , and  $T = [0, 60]$  as the input. Initially,  $g_1(t) = t$  (Fig. 4 (a)). It states that if it departs from  $v_1$  at time  $t_0$ , then it arrives at  $v_1$  at the same time  $t_0$ , and the travel time is  $g_1(t_0) - t_0 = 0$  for any  $t_0 \in T$ . At the initial stage, the starting-time subinterval  $I_1$  for  $v_1$  is  $[0, \tau_1]$ , where  $\tau_1 = 0$ . The black box indicates  $\tau_1$  (x-value) and  $g_1(\tau_1)$  (y-value), which states that  $g_1(t)$  is ensured to specify the earliest arrival time for  $v_1$  in  $I_1 = [0, \tau_1] = [0, 0]$ . For other nodes,  $v_i$  ( $i = 2, 3, 4$ ),  $g_i(t) = \infty$  and  $I_i = [0, \tau_i]$  where  $\tau_i = 0$ . It implies that  $g_i(t)$  has not been refined yet.

In the first iteration, the top pair dequeued from the priority queue  $Q$  is  $(\tau_1, g_1(t))$  where  $\tau_1 = 0$  and  $g_1(\tau_1) = 0$  (line 6). In other words,  $g_1(t)$  specifies the earliest arrival time in the starting-time subinterval  $I_1 = [0, \tau_1] = [0, 0]$ . It picks  $(\tau_3, g_3(t))$  as  $(\tau_k, g_k(t))$ , where  $\tau_3 = 0$  and  $g_3(\tau_3) = \infty$  (line 7). Then, the newly enlarged starting-time subinterval,  $I'_1$ , for  $v_1$ , becomes  $I'_1 = [0, \tau'_1] = [0, 60]$  (line 9), because  $\tau'_1$  is the latest starting time  $t$  satisfying  $g_1(t) \leq g_3(\tau_3) + \Delta = \infty + \infty$ , where  $\Delta = \infty$  as there are no coming edges to source node  $v_1$  (line 8-9). The resulting  $g_1(t)$  is shown in Fig. 4 (b). Because  $v_1$ 's starting-time subinterval  $I_1 = [0, 60]$ ,  $v_1$  can be removed from the queue  $Q$ . In this iteration, it will update the arrival-time functions for nodes  $v_2$  and  $v_3$  based on  $v_1$ . The resulting arrival-time functions,  $g_2(t)$  and  $g_3(t)$ , for  $v_2$  and  $v_3$ , are shown in Fig. 4 (c) and (d), respectively.

In the second iteration, the top pair dequeued from queue  $Q$  is  $(\tau_3, g_3(t))$ , where  $\tau_3 = 0$  and  $g_3(\tau_3) = 5$  (line 6). It will then pick  $(\tau_2, g_2(t))$  as  $(\tau_k, g_k(t))$ , where  $\tau_2 = 0$  and  $g_2(\tau_2) = 10$  (line 7). The newly enlarged starting-time subinterval,  $I'_3$ , for  $v_3$ , becomes

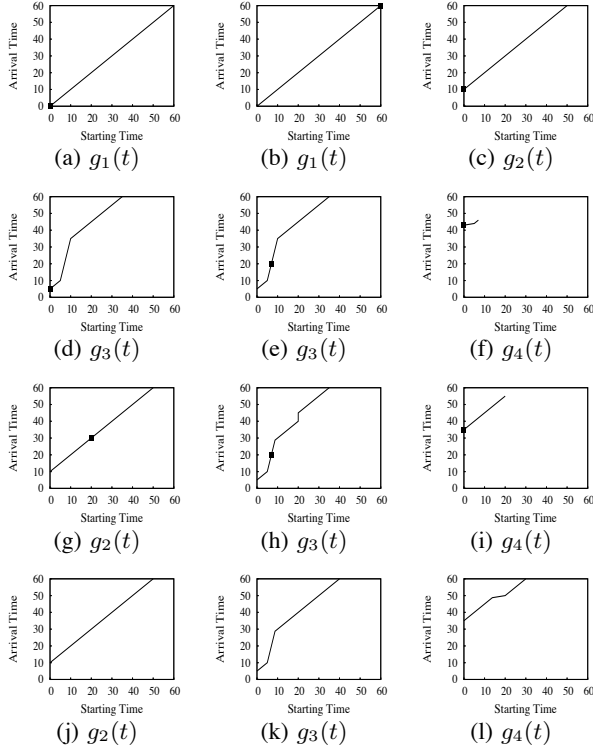


Figure 4: Arrival-Time Functions

$I'_3 = [0, \tau'_3] = [0, 7]$  (line 9), because  $\tau'_3 = \max\{t | g_3(t) \leq g_2(\tau_2) + \Delta\} = \max\{t | g_3(t) \leq 20\} = 7$  (line 9), where  $\Delta = \min\{w_{1,3}(g_2(\tau_2)), w_{2,3}(g_2(\tau_2))\} = \min\{w_{1,3}(10), w_{2,3}(10)\} = 10$  (line 8). The resulting  $g_3(t)$  in the new starting-time subinterval  $I_3 \leftarrow I'_3 = [0, 7]$  (starting-time interval refinement) is shown in Fig. 4 (e). It will also update  $g_4(t)$  on  $[0, 7]$  (arrival-time function refinement), which is shown in Fig. 4 (f).

In the third iteration, the top pair dequeued from the priority queue  $Q$  is  $(\tau_2, g_2(t))$  where  $\tau_2 = 0$  and  $g_2(\tau_2) = 10$ . The resulting  $g_2(t)$  for  $v_2$  is shown in Fig. 4 (g). The updated  $g_3(t)$  and  $g_4(t)$  are shown in Fig. 4 (h) and (i).

The iteration repeats 11 times. Functions  $g_1(t), g_2(t), g_3(t)$ , and  $g_4(t)$  are well-refined, as in Fig. 4 (b), (j), (k), and (l), respectively. The optimal starting time from source  $v_1$  is 20, and the minimized LTT is  $g_4(20) - 20 = 30$ . Based on functions  $g_1(t) \cdots g_4(t)$ , the optimal path  $\mathbf{p}^*$  can be constructed using *pathSelection* (Algorithm 2). Let  $t^* = 20$ . First, in *pathSelection*, it finds that  $g_3(t^*) + w_{3,4}(g_3(t^*)) = g_4(t^*)$ , so the predecessor to the destination node is  $v_3$ . Second, it finds that  $g_2(t^*) + w_{2,3}(g_2(t^*)) = g_3(t^*)$ , so the predecessor to  $v_3$  is  $v_2$ . Finally, in a similar fashion, it reaches the source  $v_1$ , and path  $\mathbf{p}^*$  is identified as  $(v_1, v_2)(v_2, v_3)(v_3, v_4)$ .

## 5.4 Correctness

**Theorem 5.2:** Given a FIFO time-dependent graph  $G_T(V, E, W)$  and a query LTT( $v_s, v_e, T$ ), where  $T = [t_s, t_e]$ . TWO-STEP-LTT (Algorithm 1) finds the optimal answer to LTT( $v_s, v_e, T$ ).  $\square$

**Proof Sketch:** As given in Theorem 5.1, there is no need to consider waiting time at nodes. TWO-STEP-LTT is a two-step algorithm, namely, *timeRefinement* algorithm and *pathSelection* algorithm. In the first step, *timeRefinement* refines the earliest arrival-time functions,  $g_i(t)$ , for all needed nodes  $v_i$ , to reach  $v_e$  from  $v_s$ . The optimal starting time is  $t^* \in T$  which minimizes travel time

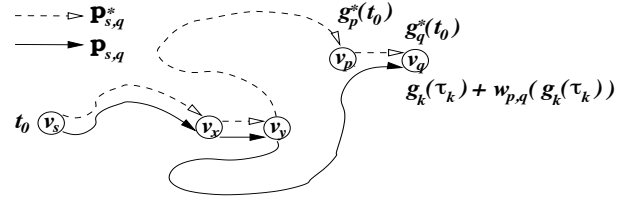


Figure 5: Intuition of the Proof

$g_e(t) - t$  from  $v_s$  to  $v_e$ . In the second step, based on  $t^*$  and arrival-time functions,  $g_i(t)$ , *pathSelection* recovers the optimal path  $\mathbf{p}^*$ . The proof for the correctness of the second step is straightforward as discussed in Section 4. In Theorem 5.3, we will prove the first step is correct, to complete the proof of Theorem 5.2.  $\square$

**Lemma 5.1:** For every  $v_i \in V$ ,  $g_i(t_1) \leq g_i(t_2)$ , if  $t_1 \leq t_2$ , is always true in *timeRefinement* (Algorithm 3).  $\square$

This lemma shows the monotonicity of  $g_i(t)$ . It can be proved directly by the definition of FIFO property and the way how  $g_i(t)$  is initialized and updated. It will be used to prove Theorem 5.3.

**Theorem 5.3:** Given a FIFO time-dependent graph  $G_T(V, E, W)$  and a query LTT( $v_s, v_e, T$ ), where  $T = [t_s, t_e]$ , *timeRefinement* (Algorithm 3) well refines the earliest arrival-time function  $g_e(t)$ , which specifies the earliest arrival time at destination node  $v_e$  for starting time  $t \in T$ .  $\square$

**Proof:** Let  $g_i^*(t)$  denote the arrival-time function that specifies the earliest arrival time from  $v_s$  to  $v_i$  for starting time  $t$ . We need prove that Algorithm *timeRefinement* terminates with  $g_e(t) = g_e^*(t)$  for  $t \in T$  (well-refined). We prove this by proving a loop invariant: *at the beginning of every iteration of the while loop (line 6), for  $v_i \in V$  and  $t \in [t_s, \tau_i]$ ,  $g_i^*(t) = g_i(t)$  is true. Or equivalently, after line 9 of every iteration, for  $v_i \in V$  and  $t \in [\tau_i, \tau'_i]$ ,  $g_i^*(t) = g_i(t)$  is true.* The proof of this loop invariant completes the proof, because initially  $\tau_i = t_s$  (line 3), and finally  $\tau_i$  is greater than or equal to  $t_e$  (line 15). Therefore, with this loop invariant, when *timeRefinement* terminates, we must have  $g_e(t) = g_e^*(t)$  for  $t \in T = [t_s, t_e]$ .

In the following, we prove the loop invariant. Based on the way how arrival time functions,  $g_i(t)$ , are initialized and updated, we have  $g_i(t) \geq g_i^*(t)$  for every  $v_i \in V$  and any  $t \in T$ . It suffices to show  $g_i(t) \leq g_i^*(t)$  in the loop invariant, to prove  $g_e(t) = g_e^*(t)$ .

For the purpose of contradiction, suppose  $v_q$  is dequeued in line 6, and  $g_q(t_0) > g_q^*(t_0)$ , for certain starting time  $t_0 \in [\tau_q, \tau'_q]$ , where the loop invariant is violated for the first time. As shown in Fig. 5, let the  $v_s$ - $v_q$  path with arrival time  $g_q(t_0)$  for starting time  $t_0$  be  $\mathbf{p}_{s,q}$ , and let the optimal  $v_s$ - $v_q$  path for starting time  $t_0$  be  $\mathbf{p}_{s,q}^* = (v_s, v_i) \cdots (v_x, v_y) \cdots (v_p, v_q)$ . We prove that  $\mathbf{p}_{s,q}^*$  is no better than  $\mathbf{p}_{s,q}$  for starting time  $t_0$  by showing  $g_q(t_0) \leq g_q^*(t_0)$ .

Consider the node  $v_y$  on path  $\mathbf{p}_{s,q}^*$  such that (i)  $t_0 > \tau_y$  and (ii)  $t_0 \leq \tau_l$  for all nodes,  $v_l$ , on the path from  $v_s$  to  $v_x$ . By (i) it means that  $g_y(t_0)$  may be not well-refined because  $t_0 \notin [t_s, \tau_y]$ . We can prove actually,  $g_y(t_0) = g_y^*(t_0)$  as follows: Since  $t_0 \leq \tau_x$ , we have  $g_x(t_0) = g_x^*(t_0)$ . Thus,  $g_y(t_0)$  is well-refined (line 11) with  $g_x(t_0)$  as  $g_y^*(t_0) = g_x(t_0) + w_{x,y}(g_x(t_0))$  in previous iterations.

If  $v_y = v_q$ , the proof of  $g_q(t_0) \leq g_q^*(t_0)$  is already completed. We will focus on the case where  $v_y \neq v_q$  in the following part.

First, since  $v_y$  appears before  $v_p$  on  $\mathbf{p}_{s,q}^*$  and all edge delays are nonnegative, we have

$$g_y(t_0) = g_y^*(t_0) \leq g_p^*(t_0). \quad (7)$$

Second, based on the choice of  $\tau'_q$  and  $\Delta$ , and the monotonicity of

$g_q(t)$  (Lemma 5.1), we have

$$\begin{aligned} g_q(t_0) &\leq g_q(\tau'_q) \text{ (for } t_0 < \tau'_q \text{ and } g_q(t) \text{ is monotone)} \\ &\leq g_k(\tau_k) + \Delta \text{ (for the choice of } \tau'_q \text{ in line 9)} \\ &\leq g_k(\tau_k) + w_{p,q}(g_k(\tau_k)) \\ &\quad \text{(for the choice of } \Delta \text{ in line 8, note } (v_p, v_q) \in E) \end{aligned} \quad (8)$$

Third, because of the choice of  $(\tau_k, g_k(t))$  in line 7,  $g_k(\tau_k)$  is the second earliest arrival time in  $Q$  following  $g_q(\tau_q)$ . We have

$$g_k(\tau_k) \leq g_y(\tau_y). \quad (9)$$

Fourth, because of the choice of  $v_y$  ( $t_0 > \tau_y$ ) and the monotonicity of  $g_y(t)$  (Lemma 5.1), we have

$$g_y(\tau_y) \leq g_y(t_0). \quad (10)$$

Then, based on Equation (9), Equation (10), and Equation (7), we have

$$g_k(\tau_k) \leq g_y(\tau_y) \leq g_y(t_0) \leq g_p^*(t_0). \quad (11)$$

Recall  $G_T$  is a FIFO graph. Based on Equation (11) and the FIFO property of edge  $(v_p, v_q)$ , we have

$$g_k(\tau_k) + w_{p,q}(g_k(\tau_k)) \leq g_p^*(t_0) + w_{p,q}(g_p^*(t_0)). \quad (12)$$

Note:  $p_{s,q}^*$  is the optimal path for the starting time  $t_0$ . We have  $g_q^*(t_0) = g_p^*(t_0) + w_{p,q}(g_p^*(t_0))$ . Based on Equation (8) and Equation (12), we can conclude that  $g_q(t_0) \leq g_p^*(t_0) + w_{p,q}(g_p^*(t_0)) = g_q^*(t_0)$ , which completes the proof.  $\square$

## 5.5 Time/Space Complexity

In this subsection, we give time complexity for manipulating piecewise-linear functions followed by the time/space complexity of our algorithms.

**Representing Functions:** Let  $f(t)$  and  $g(t)$  be piecewise-linear functions, defined on a time interval  $T = [t_s, t_e]$ , and suppose that  $f(t)$  and  $g(t)$  can be represented as  $p$  and  $q$  pieces of linear functions on subintervals of  $T$ , respectively, such that  $f(t) = \langle (f_1, t_1^f), (f_2, t_2^f), \dots, (f_p, t_p^f) \rangle$  and  $g(t) = \langle (g_1, t_1^g), (g_2, t_2^g), \dots, (g_q, t_q^g) \rangle$ . Each pair  $(f_i, t_i^f)$  represents a linear function  $f_i(t)$  on the subinterval  $[t_i^f, t_{i+1}^f]$ , and each pair  $(g_i, t_i^g)$  represents a linear function  $g_i(t)$  on the subinterval  $[t_i^g, t_{i+1}^g]$ . Note: we let  $t_1^f = t_s = t_1^g$  and  $t_{p+1}^f = t_{q+1}^g = t_e$ , where  $t_s$  and  $t_e$  are the two ends of  $T$ . General functions can be represented in a similar way.

**Implementing Function Operations:** Given two such functions  $f(t)$  and  $g(t)$ , let  $a$  and  $b$  be two constants. Four operations are defined and used in our algorithms, namely, FUNCTION INVERSE,  $f^{-1}(a) \triangleq \max\{t | f(t) = a\}$ , LINEAR COMBINATION,  $a \cdot f(t) + b \cdot g(t)$ , FUNCTION COMPOUND,  $f(g(t))$ , and MINIMUM of two functions,  $\min\{f(t), g(t)\}$ . Each operation outputs a piecewise-linear function. The time complexity for the function inverse is  $O(p)$  by swapping  $p$  pairs of  $(f_i, t_i^f)$ . The time complexity for the other three operations is  $O(p + q)$  by sweeping each of the two sequences of pairs only once. In addition, the function value  $f(t_0)$  for a given time instance  $t_0$  ( $t_s \leq t_0 \leq t_e$ ) can be computed in  $O(\log p)$  time using binary search. Details are omitted here.

In the following analysis of algorithm TWO-STEP-LTT, we take the cost of function operations into consideration, and use the similar notations for functional complexity used in [20]. We use  $\alpha(T)$  or  $\alpha(|T|)$  to denote the time/space complexity of maintaining a piecewise-linear function, or manipulating a function operation, defined in time interval  $T$  ( $|T|$  is the length of  $T$ ). Based on the representation and implementation of functions as introduced above, the

time/space required for a function operation is linearly proportional to the number of pieces needed to represent the function in  $T$ . So if assuming the number of pieces needed is linearly proportional to  $|T|$ , we have  $O(\alpha(|T_1 \cup T_2|)) = O(\alpha(|T_1|)) + O(\alpha(|T_2|))$  for  $T_1 \cap T_2 = \emptyset$ . This assumption is used in our following analysis of complexity. Note algorithms, BELLMAN-FORD, KDXZ, and our TWO-STEP-LTT, manipulate functions in the same manner.

**Complexity of Two-Step-LTT:** Given a graph  $G_T$  with  $n$  nodes and  $m$  edges in total, consider query LTT( $v_s, v_e, T$ ).

**Lemma 5.2:** The time complexity of timeRefinement (Algorithm 3) is  $O((n \log n + m)\alpha(T))$ .  $\square$

**Proof Sketch:** In each iteration of the while loop, the priority queue  $Q$  of length at most  $n$  is accessed in line 6, 7, 13, and 19. Using Fibonacci Heap [4], both  $dequeue(Q)$  and  $head(Q)$  require  $O(\log n)$  amortized time, and both  $update(Q, (\tau_j, g_j(t)))$  (when  $g_j(t)$  is updated) and  $enqueue(Q, (\tau_i, g_i(t)))$  (when the new pair  $(\tau_i, g_i(t))$  is inserted) require  $O(1)$  amortized time. Moreover, line 8 requires  $O(d_i)$  time to find  $\Delta$ , where  $d_i$  is the in-degree of node  $v_i$ , and line 9 requires  $O(\alpha(\tau'_i - \tau_i))$  time to find  $\tau'_i$ . In line 11-12, the arrival-time functions,  $g_j(t)$ , are refined within the starting-time subinterval  $[\tau_i, \tau'_i]$  in  $O(\alpha(\tau'_i - \tau_i))$  time. Therefore, for each iteration of the while loop, it needs  $O(\log n + d_i + d_i\alpha(\tau'_i - \tau_i)) \leq O((\log n + d_i)\alpha(\tau'_i - \tau_i))$  time.

Let  $\tau_i^{(k)}$  denote the value of  $\tau_i$  when  $v_i$  is dequeued from  $Q$  for the  $k^{th}$  time, and let  $l_i$  denote the number of times that  $v_i$  is dequeued from  $Q$  in total ( $k \leq l_i$ ). Then the total time complexity is  $O(\sum_{v_i \in V} \sum_{k=1}^{l_i} ((\log n + d_i)\alpha(\tau_i^{(k)} - \tau_i^{(k-1)})))$ . Because

$$\begin{aligned} &\sum_{k=1}^{l_i} ((\log n + d_i)\alpha(\tau_i^{(k)} - \tau_i^{(k-1)})) \\ &= (\log n + d_i)\alpha(\sum_{k=1}^{l_i} (\tau_i^{(k)} - \tau_i^{(k-1)})) \\ &= (\log n + d_i)\alpha(|t_e - t_s|) = (\log n + d_i)\alpha(T), \end{aligned}$$

the total time complexity is:

$$O(\sum_{v_i \in V} (\log n + d_i)\alpha(T)) = O((n \log n + m)\alpha(T)). \quad \square$$

**Lemma 5.3:** The time complexity of pathSelection (Algorithm 2) is  $O(m\alpha(T))$ .  $\square$

**Proof Sketch:** Because the value of  $g_j(t^*)$  is strictly decreasing in every iteration, every node  $v_j \in V$  will be examined at most once in  $pathSelection$  (line 5). Let  $d_j$  denote the in-degree of node  $v_j$ . The while loop requires  $O(d_j\alpha(T))$  time for each  $v_j$ . The time complexity of  $pathSelection$  can be computed as  $O(\sum_{v_j \in V} d_j\alpha(T)) = O(m\alpha(T))$ .  $\square$

From the above two lemmas (Lemma 5.2 and 5.3), we can prove the time complexity of TWO-STEP-LTT.

**Theorem 5.4:** The time complexity of TWO-STEP-LTT (Algorithm 1) is  $O((n \log n + m)\alpha(T))$ .  $\square$

In both  $timeRefinement$  and  $pathSelection$ , we need maintain graph  $G(V, E, W)$  with  $m$  edges and  $m$  functions,  $w_{i,j}(t)$ , for  $(v_i, v_j) \in E$ . During the execution of algorithms, we need maintain a priority queue  $Q$  with at most  $n$  elements, and  $n$  arrival-time functions,  $g_i(t)$ , for  $v_i \in V$ . Therefore, the total space complexity is  $O((n + m)\alpha(T))$ .

**Theorem 5.5:** The space complexity of TWO-STEP-LTT (Algorithm 1) is  $O((n + m)\alpha(T))$ .  $\square$



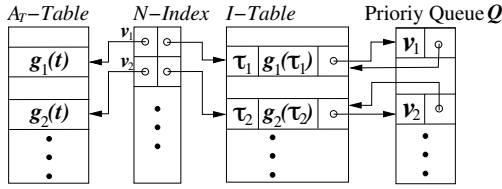


Figure 6: Runtime Data Structures

## 5.6 Storage Model and Implementation

In this subsection, we give some implementation details of our solution. A time-dependent graph  $G_T(V, E, W)$  is maintained using an edge representation, where an edge is stored as a triple  $(v_i, v_j, w_{i,j}(t))$ . The edges can be stored in a table. The first column ( $v_i$ ) and the second column ( $v_j$ ) are fixed-size whereas the third column (edge-delay function) is variable-size. Two B+-trees are built on the top of the table. One is built on the first column ( $v_i$ ), and the other is built on the second column ( $v_j$ ). They can efficiently support all the necessary operations w.r.t.  $G_T$ .

For a given  $\text{LTT}(v_s, v_e, T)$  where  $T = [t_s, t_e]$ , as shown in Fig. 6, TWO-STEP-LTT needs to maintain four runtime data structures, namely,  $N\text{-Index}$  (a list of node identifiers),  $A_T\text{-Table}$  (a list of arrival-time functions  $g_i(t)$ 's, for all  $v_i \in V$ ),  $I\text{-Table}$  (a list of pairs  $(\tau_i, g_i(\tau_i))$ 's, for all  $v_i \in V$ ), and the priority query  $Q$ .

The  $N\text{-Index}$  is a list of two pointers, which is sorted by node identifiers in order to be accessed efficiently. Given a node  $v_i$ , one of the two pointers in  $N\text{-Index}$  points to its arrival-time function,  $g_i(t)$ , which is separately maintained in  $A_T\text{-Table}$ . The arrival-time functions are maintained separately from  $N\text{-Index}$ , because they are variable-size. The separation also allows  $N\text{-Index}$  to remain unchanged, when arrival-time functions need to be updated. The other pointer in  $N\text{-Index}$  points to  $I\text{-Table}$  where  $(\tau_i, g_i(\tau_i))$ 's are maintained. Given  $v_i$ , it allows us to quickly find the corresponding pair  $(\tau_i, g_i(\tau_i))$ , when it needs to be updated.

The priority queue  $Q$  sorts  $(\tau_i, g_i(t))$  in the ascending order of  $g_i(\tau_i)$ . Every element of  $Q$  in our storage model is a pointer pointing to the corresponding pair  $(\tau_i, g_i(\tau_i))$  which is maintained in  $I\text{-Table}$ . From each element in  $I\text{-Table}$ , there is also a pointer pointing back to the position of the pair in  $Q$ . Such implementation is designed to reduce the size of  $Q$  in the running-time storage.

Since  $N\text{-Index}$  and priority queue  $Q$  are not large, they can be maintained in memory. When  $G_T$  is too large to be stored in the main memory, we maintain  $A_T\text{-Table}$ ,  $I\text{-Table}$ , and  $G_T$  on disk.

## 5.7 Solution for Non-FIFO Graphs

In this subsection, we discuss how to find the optimal LTT over a (general) non-FIFO time-dependent graph. We show that we can transform such a non-FIFO graph  $G'_T(V, E, W')$  into a FIFO graph  $G_T(V, E, W)$  where both  $V$  and  $E$  remain unchanged. Then we can process  $\text{LTT}(v_s, v_e, T)$  on the FIFO graph  $G_T$  using our proposed TWO-STEP-LTT algorithm. The optimal path  $p^*$  found in  $G_T$  can be converted into an optimal path  $p'^*$  in the original non-FIFO graph  $G'_T$ , by inserting some waiting time on each node in path  $p^*$ . The similar idea was also used in [20].

For each edge-delay function  $w'_{i,j}(t)$  in the non-FIFO graph  $G'_T$ , we define  $w_{i,j}(t)$  to construct a FIFO graph  $G_T$ .

$$\begin{aligned} w_{i,j}(t) &= \Delta_{i,j}(t) + w'_{i,j}(t + \Delta_{i,j}(t)) \\ &= \min_{0 \leq t_\Delta \leq t_e - t} \{t_\Delta + w'_{i,j}(t + t_\Delta)\} \end{aligned} \quad (13)$$

Since the starting-time interval  $T = [t_s, t_e]$  is a closed interval,  $w_{i,j}(t)$  and  $\Delta_{i,j}(t)$  in Equation (13) are well-defined. Intuitively,

$\Delta_{i,j}(t)$  is the optimal waiting time to traverse edge  $(v_i, v_j)$ , if arriving at  $v_i$  at time  $t$ . If there are multiple possible values of  $t_\Delta$  to minimize  $w'_{i,j}(t + t_\Delta) + t_\Delta$ , we select any of them as  $\Delta_{i,j}(t)$ . It is easy to verify that edge  $(v_i, v_j)$  with edge delay function  $w_{i,j}(t)$  has the FIFO property. Let  $W$  be the set of newly defined edge delays  $w_{i,j}(t)$ 's, and then  $G_T(V, E, W)$  is a FIFO graph.

Suppose using TWO-STEP-LTT algorithm, we find the optimal path  $p^* = (v_1, v_2) \cdots (v_{k-1}, v_k)$ , where  $v_1 = v_s$  and  $v_k = v_e$ , together with the best starting time  $t^* \in T$ , for  $\text{LTT}(v_s, v_e, T)$  on the converted FIFO graph  $G_T(V, E, W)$ . We construct the optimal path  $p'^*$  for  $\text{LTT}(v_s, v_e, T)$  on the original non-FIFO graph  $G'_T(V, E, W')$  by inserting waiting time  $\varpi^*(v_i) = \Delta_{i,i+1}(t)$  at node  $v_i$  for  $1 \leq i \leq k-1$ , where  $t$  is the arrival time at node  $v_i$  along path  $p^*$  in  $G_T$  for starting time  $t^*$ .

## 6. PERFORMANCE STUDIES

In this section, we conducted extensive experimental studies to compare our solution, TWO-STEP-LTT, with other three algorithms for the TDSP problem, namely, the most efficient discrete-time algorithm DOT [2], BELLMAN-FORD based algorithm OR [20], and A\* algorithm KDXZ [15]. We implemented all algorithms using C++. Note we denote our TWO-STEP-LTT as "2S" in figures below for conciseness. For DOT, let  $\delta$  be the length of the interval between two adjacent time points, and we used  $\delta = 0.1$  (unit).

**Experiment Setup:** We use a real dataset with 16,326 nodes and 26,528 edges, representing the road-map in the Maryland State in US. The dataset is extracted from the US Census Bureau 2005 TIGER/Line<sup>4</sup>. The nodes represent the starts, ends, and intersections of roads, while the edges represent road segments. Note the four algorithms can handle both undirected and directed graphs. In experiments, we represent the real database as a directed time-dependent graph  $G_T$ . We further generate 10 subgraphs  $G_1, \dots, G_{10}$  from  $G_T$  with the number of nodes varying from 40 to 10K. Each subgraph corresponds to a subarea of  $G_T$ . The numbers of nodes and edges of  $G_1 \cdots G_{10}$  are listed in Table 2.

We test the class of continuous piecewise-linear edge-delay functions  $W = \{w_{i,j}(t)\}$ , whose operations are implemented as described in Section 5.5. Each  $w_{i,j}(t)$  is generated randomly with four parameters, *average-delay*  $\bar{w}$ , *range-delay*  $w_\Delta$ , *length-domain*  $L_T$ , and *number-segment*  $N_T$ , in domain  $\mathcal{T} = [0, L_T]$  independently as follows:  $\mathcal{T}$  is randomly divided into  $N_T$  subintervals; within each one,  $w_{i,j}(t)$  is a linear function; the value of  $w_{i,j}(t)$  at the start/end of each subinterval is randomly generated as a number in  $[\bar{w} - w_\Delta, \bar{w} + w_\Delta]$  uniformly. Note: the  $w_{i,j}(t)$ 's generated as described above are general edge-delay functions (some may have FIFO properties while the others may be non-FIFO).

In Experiment-1 and Experiment-2,  $\bar{w} = 11$ ,  $w_\Delta = 9$ ,  $L_T = 2,000$ , and  $N_T$  is randomly picked from 4 to 8. In Experiment-3, we will vary some of the four parameters to test the scalability of algorithms w.r.t. different types of edge-delay functions.

The set of queries,  $\{\text{LTT}(v_s, v_e, T)\}$ , used in each test is constructed by fixing source  $v_s$  as the center of graph, and varying destination  $v_e$  over all the other nodes in graph.

We conducted all tests on a 2.8GHz CPU/1G memory PC running XP. We report the processing time (second), and the memory consumed (byte).

**Experiment-1 (Graph-Scalability):** For query  $\text{LTT}(v_s, v_e, T)$ , we fix starting-time interval  $T = [0, 500]$ , and vary (i) the number of nodes, and (ii) the number of edges, in the time-dependent graph.

<sup>4</sup>Topologically Integrated Geographic Encoding and Referencing system: <http://www.census.gov/geo/www/tiger/>

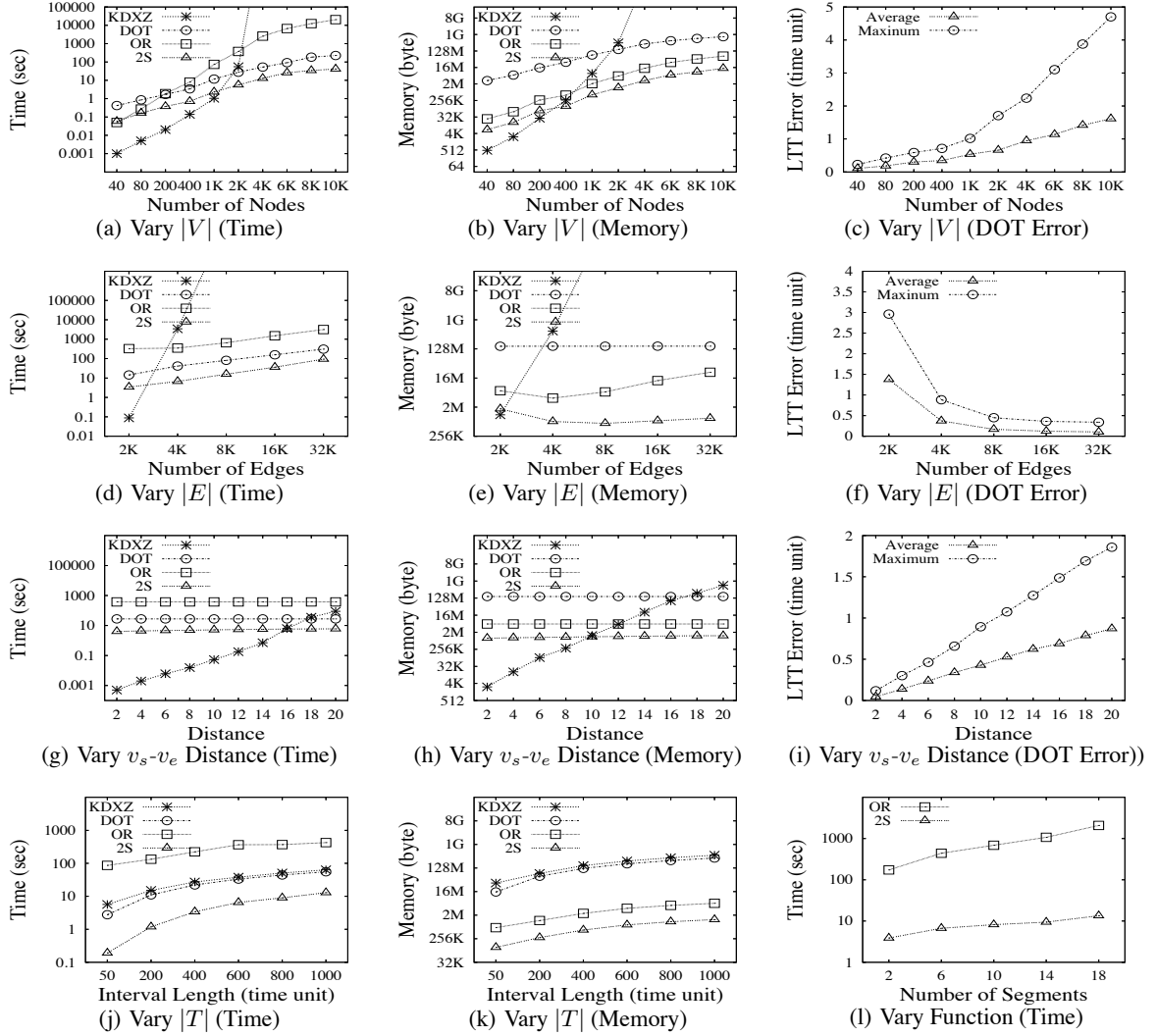


Figure 7: Testing Algorithms for TDSP Problem

|       | $G_1$ | $G_2$ | $G_3$ | $G_4$ | $G_5$ | $G_6$ | $G_7$ | $G_8$ | $G_9$ | $G_{10}$ |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|
| $ V $ | 40    | 80    | 200   | 400   | 1K    | 2K    | 4K    | 6K    | 8K    | 10K      |
| $ E $ | 52    | 107   | 262   | 548   | 1.5K  | 3K    | 6.4K  | 9.7K  | 13K   | 16K      |

Table 2: Datasets

First, with the number of nodes increasing from 40 to 10K in  $G_1, \dots, G_{10}$ , the average processing time and memory consumed are shown in Fig. 7(a) and Fig. 7(b). The average processing time of DOT is about 10 times larger than our TWO-STEP-LTT consistently. TWO-STEP-LTT outperforms OR when the number of nodes is  $> 80$ . KDXZ algorithm is the fastest when there are less than 1K nodes, but it becomes slower than TWO-STEP-LTT when there are more than 1K nodes. The average processing time of KDXZ increases exponentially, and it cannot finish for most queries in this experiment when the number of nodes is  $> 4K$ , because its search space is exponentially w.r.t. the size of graph. TWO-STEP-LTT outperforms OR significantly when the graph is large in size. The average memory consumed by OR is 5 times larger than TWO-STEP-LTT, because OR maintains two sets of functions,  $\{g_i(t)|v_i \in V\}$  and  $\{h_{j,k}(t)|(v_j, v_k) \in E\}$ , while TWO-STEP-LTT only maintains  $\{g_i(t)|v_i \in V\}$ . The memory consumed by DOT is 50 times larger than TWO-STEP-LTT con-

sistently. KDXZ consumes much more than TWO-STEP-LTT does when the number of nodes is  $> 1K$ . Note: KDXZ, OR, and TWO-STEP-LTT find the optimal LTT. Fig. 7(c) shows the average and maximum LTT error of DOT. By LTT error, we mean the difference between the LTT obtained by DOT and the optimal LTT. When the number of nodes increases, LTT error becomes larger, because the average distance between two nodes becomes larger.

Second, we vary the density of  $G_6$  by fixing the number of nodes as 2K while changing the number of edges. 5 graphs are generated with 2K, 4K, 8K, 16K and 32K edges. We report average processing time and memory consumed in Fig. 7(d) and 7(e) respectively. The average processing time increases when the number of edges increases. TWO-STEP-LTT significantly outperforms DOT and KDXZ in all the cases, and outperforms KDXZ when the number of edges is  $> 2K$ . KDXZ cannot find LTT in reasonable time for most queries tested when the number of edges is  $> 4K$ . In terms of memory consumption, TWO-STEP-LTT performs the best followed by OR and then DOT. The amount of memory KDXZ consumes is exponentially proportional to the number of edges. Fig. 7(f) shows the average/maximum LTT error of the DOT. The error becomes smaller when the number of edges increases, because with more edges, the average distance between

two nodes becomes smaller, which decreases the LTT error.

**Experiment-2 (Query-Scalability):** We use  $G_6$  (with 2K nodes and 3K edges). For query  $LTT(v_s, v_e, T)$ , (i) we vary the number of nodes on the shortest  $v_s-v_e$  path (i.e.,  $v_s-v_e$  distance, x-axis “Distance” in Fig. 7(g)-7(i)); (ii) we change the length of starting-time interval  $T$  (i.e., x-axis “Interval Length” in Fig. 7(j)-7(k)).

First, the starting-time interval  $T$  is fixed to be  $[0, 500]$  for all queries. With the number of nodes on the shortest  $v_s-v_e$  path increasing from 2 to 20, we report the average processing time and memory consumed in Fig. 7(g) and 7(h) respectively. Note the time/memory consumed by DOT and OR are nearly unchanged, because they cannot terminate until the LTT from  $v_s$  to every other node is determined. KDXZ performs well if the  $v_s-v_e$  distance is  $< 15$ , but quickly deteriorates otherwise. TWO-STEP-LTT constantly outperforms KDXZ after the  $v_s-v_e$  distance is  $> 16$  on both time and memory consumed, because the size of the search space in KDXZ increases exponentially w.r.t.  $v_s-v_e$  distance. Fig. 7(i) shows the average/max LTT error of DOT which becomes larger, while the number of nodes on the  $v_s-v_e$  shortest path increases.

Second, we vary the length of starting-time interval  $T$  from 50 to 1,000, and report the average of processing time and memory consumed in Fig. 7(j) and 7(k) respectively. It is shown that all algorithms need additional time/memory with the length of interval  $T$  increases, because the increment of  $|T|$  incurs both additional function-operation time and search space. TWO-STEP-LTT outperforms the others consistently.

**Experiment-3 (Edge-Delay Function):** We test the effect of edge-delay functions on the processing time of TWO-STEP-LTT and OR, since both request larger numbers of function operations than the other two. We use  $G_6$ , and fix  $T = [0, 500]$ .

First, for every edge, we fix  $\bar{w} = 11$ ,  $w_\Delta = 9$ ,  $L_T = 2000$ , and vary  $N_T$  from 2 to 18. When  $N_T$  increases, the edge delay function fluctuates more frequently. We report the average processing time consumed by TWO-STEP-LTT and OR in Fig. 7(l). TWO-STEP-LTT outperforms OR.

Second, we fix  $\bar{w} = 11$ ,  $L_T = 2000$ ,  $N_T = 8$ , and vary  $w_\Delta$  from 2 to 10. When  $w_\Delta$  increases, both TWO-STEP-LTT and OR consume more time, because resulting functions  $g_i(t)$ 's can be more complicated, and hence require more function-operation time in both. TWO-STEP-LTT again outperforms OR in this test. Due to the limit of space, we do not report the detailed result of this test.

## 7. RELATED WORK

As shown in [26], answering  $LTT(v_s, v_e, t)$  for a given starting time  $t$  (not a starting-time interval) in a time-dependent graph  $G_T$  can be solved similarly as a single-source shortest-path problem in a static graph with constant edge delays [26]. Chon et al. in [3] proposed a system architecture to answer  $LTT(v_s, v_e, t)$  for a given starting time  $t$  in a distributed environment. The variations of single-source shortest-path problem and related issues have been intensively studied in the areas of transportation [15, 12, 13, 17, 2, 29, 21], navigation systems [24, 14, 3, 9], and networks [18, 19].

**Dynamic Shortest-Path (DSP):** The DSP problem is to recompute shortest-paths repeatedly, while the underneath graph with constant edge delays is allowed to be updated from time to time. The updates include insertion/deletion of edges and edge-weight updates. Frigioni et al. in [8] proposed fully dynamic algorithms with optimal space requirements and query time for single-source DSP problem, and King in [16] presented the first fully dynamic algorithm for all-pair DSP problem. Demetrescu and Italiano in [6] presented an improved all-pair DSP algorithm, which can find the shortest paths

in optimal worst-case time. Experimental evaluations for DSP algorithms can be found in [7, 5]. Roditty discussed the hardness of the DSP problems in [22]. The TDSP problem studied in this paper deals with edge-delay functions over a fixed time-dependent graph, whereas DSP deals with unpredictable updates against a static graph. They are two different problems and the techniques used in one cannot be directly applied to the other.

**Hierarchy-Based Method:** In order to deal with a large graph, hierarchy-based methods partition the graph into small fragments and materialize the shortest-paths between border nodes in different fragments. The shortest-path between two nodes in the graph is obtained by combining the shortest-paths from different fragments [12, 13, 14]. Different graph partitioning methods for the shortest-path problem were studied, such as disjoint edge-set partition [12, 13] and disjoint node-set partition [14]. Shekhar et al. in [23] studied the materialization trade-offs. [27] proposed a linear-time algorithm for the static single-source shortest-path problem using graph partitioning idea. Graph-partition techniques can be also embedded into our algorithm to find LTT over time-dependent graphs.

**Storage of Graph and I/O Efficiency:** Shekhar et al in [25] proposed CCAM (Connectivity-Clustered Access Method), and studied how to store large graphs on disk using connectivity clustering and to support basic operations, such as insert, delete, create, find, and get-successor, which are necessary for most graph algorithms (including our algorithm presented in this paper). Graph update is also discussed in [25]. Huang et al. in [10] studied spatial partition clustering which creates balanced partitions of links based on the spatial proximity of nodes. Woo et al. in [28] studied network traversal clustering for the storage of graphs based on graph partitioning. For the shortest-path problem, Jiang in [11] analyzed the I/O-efficiency of several representative algorithms, and their properties regarding database applications. Experimental results regarding I/O-efficiency of shortest-path algorithms can be found in [24].

## 8. CONCLUSIONS

In this paper, we studied the time-dependent shortest-path problem, that is, answering query  $LTT(v_s, v_e, T)$  in a time-dependent graph  $G_T(V, E, W)$ . We proposed a new DIJKSTRA-based algorithm to find the optimal  $LTT(v_s, v_e, T)$  with time complexity  $O((n \log n + m)\alpha(T))$  and space complexity  $O((n + m)\alpha(T))$ , where  $n$  is the number of nodes,  $m$  is the number of edges, and  $\alpha(T)$  is the cost required for each function operation. We conducted extensive studies over large time-dependent graphs, and confirmed that our algorithm can obtain the optimal  $LTT(v_s, v_e, T)$  efficiently for handling large time-dependent graphs.

**Acknowledgment:** The authors thank Donghui Zhang and Yang Du for the helpful discussions and the code provided. This work was supported by a grant of RGC, Hong Kong SAR, China (No. 418206).

## 9. REFERENCES

- [1] X. Cai, T. Kloks, and C. K. Wong. Time-varying shortest path problems with constraints. *Networks*, 29(3):141–150, 1997.
- [2] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Record*, 1645:170–175, 1998.
- [3] H. D. Chon, D. Agrawal, and A. E. Abbadi. Fates: Finding a time dependent shortest path. In *Mobile Data Management*,

pages 165–180, 2003.

- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press.
- [5] C. Demetrescu, S. Emiliozzi, and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *SODA*, pages 369–378, 2004.
- [6] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *J. ACM*, 51(6):968–992, 2004.
- [7] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasquale. Experimental analysis of dynamic algorithms for the single-source shortest-path problem. *ACM Journal of Experimental Algorithms*, 3:5, 1998.
- [8] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem (extended abstract). In *SODA*, pages 212–221, 1996.
- [9] S. Handley, P. Langley, and F. A. Rauscher. Learning to predict the duration of an automobile trip. In *KDD*, pages 219–223, 1998.
- [10] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Effective graph clustering for path queries in digital map databases. In *CIKM*, pages 215–222, 1996.
- [11] B. Jiang. I/O-efficiency of shortest path algorithms: An analysis. In *ICDE*, pages 12–19, 1992.
- [12] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportation applications. In *CIKM*, pages 261–268, 1996.
- [13] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10(3):409–432, 1998.
- [14] S. Jung and S. Pramanik. Hiti graph model of topographical roadmaps in navigation systems. In *ICDE*, pages 76–84, 1996.
- [15] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, pages 10–19, 2006.
- [16] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *FOCS*, pages 81–91, 1999.
- [17] K. Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83:154–166, 1995.
- [18] P. Narváez, K.-Y. Siu, and H.-Y. Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Trans. Netw.*, 8(6):734–746, 2000.
- [19] P. Narváez, K.-Y. Siu, and H.-Y. Tzeng. New dynamic SPT algorithm based on a ball-and-string model. *IEEE/ACM Trans. Netw.*, 9(6):706–718, 2001.
- [20] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *J. ACM*, 37(3):607–625, 1990.
- [21] S. Pallottino and M. G. Scutellà. Shortest path algorithms in transportation models: classical and innovative aspects. In *Equilibrium and Advanced Transportation Modelling*, pages 245–281, 1998.
- [22] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *ESA*, pages 580–591, 2004.
- [23] S. Shekhar, A. Fetterer, and B. Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In *SSD*, pages 94–111, 1997.

- [24] S. Shekhar, A. Kohli, and M. Coyle. Path computation algorithms for advanced traveller information system (ATIS). In *ICDE*, pages 31–39, 1993.
- [25] S. Shekhar and D.-R. Liu. CCAM: A connectivity-clustered access method for networks and network computations. *IEEE Trans. Knowl. Data Eng.*, 9(1):102–119, 1997.
- [26] K. Sung, M. G. Bell, M. Seong, and S. Park. Shortest paths in a network with time-dependent flow speeds. *European Journal of Operational Research*, 121(12):32–39, 2000.
- [27] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
- [28] S.-H. Woo and S.-B. Yang. An improved network clustering method for I/O-efficient query processing. In *ACM-GIS*, pages 62–68, 2000.
- [29] J. L. Zhao and A. Zaki. Spatial data traversal in road map databases: A graph indexing approach. In *CIKM*, pages 355–362, 1994.

## APPENDIX

### A. ABOUT ROAD NETWORK

Kanoulas et al. in [15] studied finding the optimal LTT over a road network with speed-patterns, which is defined as a graph  $G_S(V, E, L, S)$ :  $V = \{v_i\}$  is a set of nodes;  $E \subseteq V \times V$  is a set of edges;  $L$  is a set of edge lengths;  $S$  is a set of speed-pattern functions. Each edge (road)  $(v_i, v_j) \in E$  is associated with a length  $l_{i,j} \in L$ , and a speed-pattern function  $s_{i,j}(t) \in S$ . The speed of all vehicles on edge  $(v_i, v_j)$  is at most  $s_{i,j}(t)$  at time  $t$  in domain  $T$ .

For finding LTT over a road network, Kanoulas et al. construct an equivalent time-dependent graph  $G_T(V, E, W)$  from the road network  $G_S(V, E, L, S)$ , where the node/edge set ( $V$  and  $E$ ) of  $G_S$  is the same as the node/edge set of  $G_T$ . The relationship between  $w_{i,j}(t)$  and  $(l_{i,j}, s_{i,j}(t))$  is given below, where  $t$  is the departure time from  $v_i$ .

$$w_{i,j}(t) = \min\{w \mid \int_t^{t+w} s_{i,j}(z) dz = l_{i,j}\} \quad (14)$$

To show time-dependent graph  $G_T$  constructed from road network  $G_S$  is an *FIFO* graph, we only need to show, for  $w_{i,j}(t)$  defined in Equation (14),  $w_{i,j}(t_0) \leq t_\Delta + w_{i,j}(t_0 + t_\Delta)$  for  $t_0 \in T$  and  $t_\Delta \geq 0$  (Definition 5.1). For the purpose of contradiction, we suppose  $w_{i,j}(t_0) = t_\Delta + w_{i,j}(t_0 + t_\Delta) + \epsilon$  for  $\epsilon > 0$ . Based on Equation (14), for departure time  $t = t_0$  and  $t = t_0 + t_\Delta$ , we have

$$l_{i,j} = \int_{t_0}^{t_0 + w_{i,j}(t_0) = t_0 + (t_\Delta + w_{i,j}(t_0 + t_\Delta) + \epsilon)} s_{i,j}(z) dz \quad (15)$$

and

$$l_{i,j} = \int_{(t_0 + t_\Delta)}^{(t_0 + t_\Delta) + w_{i,j}(t_0 + t_\Delta)} s_{i,j}(z) dz, \quad (16)$$

respectively. From Equation (15)-(16) and  $s_{i,j}(z) \geq 0$ , we must have  $s_{i,j}(z) = 0$  for  $z \in [t_0 + t_\Delta + w_{i,j}(t_0 + t_\Delta), t_0 + t_\Delta + w_{i,j}(t_0 + t_\Delta) + \epsilon]$ , and thus from Equation (15), we have

$$l_{i,j} = \int_{t_0}^{t_0 + (t_\Delta + w_{i,j}(t_0 + t_\Delta))} s_{i,j}(z) dz. \quad (17)$$

From the definition of  $w_{i,j}(\cdot)$  in Equation (14), the above equation contradicts with  $w_{i,j}(t_0) = t_\Delta + w_{i,j}(t_0 + t_\Delta) + \epsilon$ . Therefore,  $w_{i,j}(t_0) \leq t_\Delta + w_{i,j}(t_0 + t_\Delta)$ , and time-dependent graph  $G_T$  constructed from road network  $G_S$  is an *FIFO* graph.