2019 IEEE 35th International Conference on Data Engineering (ICDE)

Weight-Constrained Route Planning over Time-Dependent Graphs

Ye Yuan[§] Xiang Lian[†] Guoren Wang[§] Lei Chen[♯] Yuliang Ma[‡] Yishu Wang[‡]
[§]Beijing Institute of Technology [†]Kent State University [‡]Northeastern University, China
[♯]Hong Kong University of Science and Technology

***Abstract*—Weight-constrained route planning (WRP) over static graphs has been extensively studied due to its wide application to transportation networks. However, real transportation networks often evolve over time and are thus modeled as *time-dependent graphs*. In this paper, we study the WRP problem over a large time-dependent graph by incorporating continuous time and weight functions into it. Most existing works regarding route planning over time-dependent graphs are based on the first-in-first-out (FIFO) property. Unfortunately, the FIFO property does not hold for our problem. To solve the problem, we propose two novel route planning algorithms, namely, a baseline algorithm and an advanced algorithm. Specifically, the advanced algorithm is even more efficient than the baseline algorithm, as the advanced algorithm incorporates a fast traversal scheme and tight bounds of time functions to terminate the traversal as early as possible. We confirm the effectiveness and efficiency of our algorithms by extensive experiments on real datasets.**

I. Introduction

Given a source s and a destination d of a static graph G_s, the weight-constrained route planning (WRP) problem over G_s involves finding the best path from s to d based on its length with a constraint on its weight [1], [2]. The WRP problem in static graphs has been extensively studied [1], [2], [3], [4], [5], since it has many real application to transportation networks. For instance, from s to d over G_s, a user wants to compute a route with not only the shortest travel time (length) but also toll payment (weight) within a budget. In this scenario, the user should compute a WRP query that minimizes the total travel time within the budget for toll payment. In reality, transportation networks often evolve over time. For example, the Singapore Vehicle Information System and the American Traffic Message Channel are two transportation systems that can provide real-time traffic information to users. Such transportation networks are *time-dependent graphs*, i.e., the travel time for a road varies with at different times. Therefore, in this paper, we study the WRP query over a large time-dependent graph.

In practice, a transportation network can be modeled as a time-dependent graph G_t as follows. Every edge $e = (u, v)$ in G_t has two types of costs: $f_e(t)$ and $w_e(t)$, where $f_e(t)$ is the time cost to specify how long it takes to travel through an edge e, and $w_e(t)$ is the toll fee (weight) for traveling through an edge e. Both $f_e(t)$ and $w_e(t)$ are the functions that are dependent on the departure time t at the starting endpoint u of the edge $e = (u, v)$. Figure 1 gives a time-dependent graph G_t with cost functions on its edges. For example, in Figure 1(b),

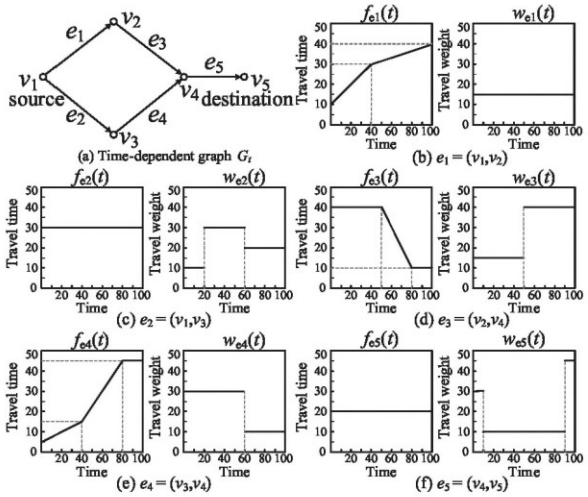


Fig. 1: A time-dependent graph with time and weight functions on edges.

$f_{e_1}(t)$ is the time cost-function of edge e_1 of G_t, and $w_{e_1}(t)$ is the weight cost-function of edge e_1 of G_t.

Consider an application in the time-dependent graph G_t (of Figure 1) with source $s = v_1$ and destination $d = v_5$. Someone has an appointment with her friends at rendezvous d and wants to depart at any time during a certain period $[t_{s1}, t_{s2}]$ from s. Under this condition, she wants to compute the earliest arrival time from s to d that incurs a toll fee within a budget Δ. This type of query is called the *interval WRP (IWRP) query over time-dependent graphs*.

Specifically, in this paper, we study the IWRP query over time-dependent graphs G_t, which can be defined as follows. Given a source s, a destination d, a departure period $[t_{s1}, t_{s2}]$, and a weight constraint Δ in G_t, find an optimal path p from s to d satisfying the following three conditions: (1) the departure from s occurs within the period $[t_{s1}, t_{s2}]$ along path p, (2) the weight (toll fee) of path p is at most Δ, and (3) path p has the earliest arrival time among all the paths satisfying Conditions (1) and (2).

We continue the above application with $[t_{s1}, t_{s2}] = [0, 30]$ and $\Delta = 80$. The answer to the IWRP query is the optimal path $v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_5$ with the earliest arrival time of 70 and the related weight of 75.

There are many works regarding the route planning problem

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00086

in time-dependent graphs G_t [6], [7], [8], [9]. Most of them involve finding an optimal path with the earliest arrival time from the source to the destination when the departure time from the source can be selected from a user-given starting-time interval. These works assume that every edge of G_t has only one time function $f_e(t)$ that is *continuous*. They employ Dijkstra-like algorithms. The premise of applying Dijkstra-like algorithms is that G_t has the *first-in-first-out* (FIFO) property, which means that a vehicle that enters a road segment first will also reach the end of the road segment first [10]. Unfortunately, the FIFO property does not hold for our problem (as detailed in Section II-B). Thus, the existing works cannot solve the problem proposed in this paper.

To solve the problem, we propose a unified algorithmic framework, IWRP_Framework, to answer the IWRP query efficiently. Specifically, IWRP_Framework consists of three steps. In the first step, IWRP_Framework defines a skyline function for every node of G_t and specifies how to calculate it. In this step, IWRP_Framework also proposes a traversal procedure from source s to destination d, during which every skyline function is calculated. After the traversal, we can obtain the minimum arrival-time and weight functions of d, $Arr_d(t)$ and $Wgh_d(t)$, respectively. In the second step, based on $Arr_d(t)$ and $Wgh_d(t)$, IWRP_Framework computes the departure time $Dep(s)$ from s and the arrival time $Arr(d)$ at d satisfying the query. Finally, IWRP_Framework computes the optimal route from s at time $Dep(s)$ to d at time $Arr(d)$.

The traversal procedure is very important for the efficiency of IWRP_Framework. We first propose a baseline traversal algorithm. This algorithm is elegant but has a high time complexity of $O((k_t+k_w)n^3m^2M_e^2)$, where n and m are the node and edge numbers, k_t and k_w are the average numbers of breakpoints of arrival-time and weight functions, and M_e is the maximum value of edge-time functions. To improve the baseline algorithm, we also design an advanced traversal algorithm that incorporates tight bounds on arrival-time functions to prune the search space greatly. The advanced algorithm has a time complexity of $O(M_e(k_t+k_w)nm\log(nM_e))$ which is a great reduction compared to that of the baseline algorithm. Moreover, we further improve the advanced algorithm by adding heuristics so that the advanced algorithm runs very quickly on large time-dependent graphs (e.g., 9.8s on G_t with 5M nodes and 32M edges) as demonstrated in the experiments.

To summarize, we make the following contributions.

Contributions. (1) in Section II, we formulate a novel IWRP query over a large time-dependent graph by incorporating continuous time and weight functions.

(2) We propose a unified framework, IWRP_Framework, to answer the IWRP query over time-dependent graphs in Section III. IWRP_Framework consists of three steps, namely, a skyline function, an evaluation that unifies a traversal algorithm and skyline functions, and a route-retrieval procedure that calculates the actual routes satisfying the IWRP query predicate.

(3) We plug a baseline traversal algorithm into IWRP_Framework to answer the IWRP query in Section IV. To speed up the baseline algorithm, we also develop an advanced algorithm that incorporates a novel traversal strategy and bounds on time-functions to prune the search space in Section V.

(4) We evaluate the effectiveness and efficiency of our proposed algorithms with extensive experiments on road networks, measuring both the running time and memory overhead in Section VI.

II. Problem Definition

A. Weight-Constrained Route Planning

In this section, we define our IWRP queries over time-dependent graphs similar to works [7], [11].

Time-Dependent Graph. A time-dependent graph is a simple directed graph, denoted $G_t(V,E,F,W)$ (or G_t for short), where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and F and W are two sets of non-negative value functions. For every edge $e=(u,v) \in E$, there are two functions: time function $f_e(t) \in F$ and weight function $w_e(t) \in W$, where t is a time variable. A time function $f_e(t)$ specifies how much time it takes to travel from u to v if departing from u at time t. A weight function $w_e(t)$ specifies the cost (e.g., toll fee) it takes to travel from u to v if departing from u at time t. We let $|V|=n$ and $|E|=m$.

In this paper, we assume that $f_e(t)>0$ and $w_e(t) \geq 0$. The assumption is reasonable because the travel time is always positive and the travel cost (weight) cannot be negative in real applications. Our work can be easily extended to handle undirected graphs, in which an undirected edge (u,v) is equivalent to two directed edges $\langle u,v\rangle$ and $\langle v,u\rangle$, where $f_{\langle u,v\rangle}(t) = f_{\langle v,u\rangle}(t)$ and $w_{\langle u,v\rangle}(t) = w_{\langle v,u\rangle}(t)$. Below, we provide more details about time and weight functions.

Time Function. The edge-time function $f_e(t)$ is a continuous and periodic (with time period T) function defined as follows: $\forall k \in \mathbb{N}$, $\forall t \in [0,T)$, $f_e(kT+t) = f_e(t)$, where $f_e : [0,T) \rightarrow [1,M_e]$ such that $\lim_{t \rightarrow T} f_e(t) = f_e(0)$ for some fixed integer M_e denoting the maximum value of $f_e(t)$. Without loss of generality, $f_e(t)$ can be approximately represented by a *piece-wise linear* (PWL) function. In fact, any continuous function can be approximately by a set of PWL functions through a numerical approximation method [12]. Since f_e is a periodic, continuous and PWL function, it can be represented succinctly by a number K_e of breakpoints defining f_e. Let $K=\sum_{e \in E} K_e$ denote the number of breakpoints to represent all of the edge-time functions in G_t.

For example, Figure 1 shows a time-dependent graph G_t with a time function $f_e(t)$ and a weight function $w_e(t)$ for each edge. In Figure 1 (d), $f_{e_3}(t)$ defines the time function of the edge $e_3=(v_2,v_4)$. The period T and M_e of $f_{e_3}(t)$ are 100 and 50, respectively. $f_{e_3}(t)$ has two breakpoints, $(50,40)$ and $(80,10)$.

Arrival-time Function. For a node $v \in V$, we use $Arr(v)$ and $Dep(v)$ to denote the arrival time at v and departure time from v, respectively. Then, for an edge $e=(u,v) \in E$, we

have $Arr(v) = Dep(u) + f_e(Dep(u))$. Note that, we do not consider the waiting time during the routing path (the reasons will be given later.), and thus we have $Arr(v) = Dep(v)$. Let $p = \langle e_1 = (v_1, v_2), e_2 = (v_2, v_3), ..., e_h = (v_h, v_{h+1})\rangle$ be a given path with the departure time t_s. Then, we have

$$\begin{aligned} &Arr(v_1) = Dep(v_1) = t_s, \\ &Arr(v_2) = Arr(v_1) + f_{e_1}(Arr(v_1)), \\ &\quad\cdots \\ &Arr(v_{h+1}) = Arr(v_h) + f_{e_h}(Arr(v_h)). \end{aligned}$$

The *travel time* of path p is defined as $Trv(p) = Arr(v_{h+1}) - t_s$. The *edge-arrival-time function* of an edge $e \in E$ is defined as $Arr_e(t) = t + f_e(t)$, $\forall t \in [0, T)$. Then, the *path-arrival-time function* of a path $p = \langle e_1, ..., e_h\rangle$ is the composition $Arr_p(t) = Arr_{e_h}(Arr_{e_{h-1}}(\cdots(Arr_{e_1}(t))\cdots))$ of the edge-arrival-time functions for the constituent edges. The *path-travel-time function* is then $Trv_p(t) = Arr_p(t) - t$.

Weight Function. We assume that edge-weight function $w_e(t)$ is a piecewise constant function, which can be formalized as follows:

$$w_e(t) = \begin{cases} w_1, & 0 \le t < t_1 \\ w_2, & t_1 \le t < t_2 \\ \cdots & \\ w_\sigma, & t_{\sigma-1} \le t < t_\sigma \end{cases} \quad (1)$$

Here, $[0, t_\sigma]$ is the time domain of function $w_e(t)$ with σ breakpoints. The value of w_x $(1 \le x \le \sigma)$ is a constant, which represents the value of $w_e(t)$ when $t \in [t_{x-1}, t_x]$. The assumption is reasonable. In real applications, the weight functions are always piecewise constant. For example, in road networks, the toll fees for traveling through a road are distinct constant values during the day and night. This fact implies that the weight-function of this road is a piecewise-constant function.

Figure 1 illustrates the weight-functions for five edges $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_3)$, $e_3 = (v_2, v_4)$, $e_4 = (v_3, v_4)$ and $e_5 = (v_4, v_5)$.

Similar to the time function, let $p = \langle e_1 = (v_1, v_2), e_2 = (v_2, v_3), ..., e_h = (v_h, v_{h+1})\rangle$ be a given path with the departure time t_s. For any vertex $v_i \in p$, we use $Wgh(v_i)$ to denote the weight from v_1 to v_i by path p. $Wgh(v_i)$ can be calculated recursively as follows:

$$\begin{aligned} &Wgh(v_1) = 0, Arr(v_1) = t_s, \\ &Wgh(v_2) = Wgh(v_1) + w_{e_1}(Arr(v_1)), \\ &\quad\cdots \\ &Wgh(v_{h+1}) = Wgh(v_h) + w_{e_h}(Arr(v_h)). \end{aligned}$$

The weight of path p is defined as $Wgh(p) = Wgh(v_{h+1})$. Let s and d be the route source and destination nodes in G_t and $[t_{s1}, t_{s2}]$ be a starting time interval at s. Let Δ be a user-specified weight constraint on the route from s to d. Next, we provide the definition of the problem of the interval weight-constrained route planning (IWRP) query in time-dependent graphs.

Definition 1 (Interval Weight-Constrained Route Planning): Given a time-dependent graph $G_t = (V, E, F, W)$, a query $Q = (s, d, t_{s1}, t_{s2}, \Delta)$ is to find a path from s to d, represented as $p = \langle v_0, v_1, ...v_{h+1}\rangle$, such that: (1) $s = v_0$ and $d = v_{h+1}$, (2) $t_{s1} \le Dep(s) \le t_{s2}$, (3) $Wgh(p) \le \Delta$, and (4) $Arr(d)$ is the minimum among all possible paths meeting Conditions (1), (2) and (3). □

In this paper, we only study the problem of computing the earliest arrival time (i.e.,the least $Arr(d)$), as the problems of computing the latest departure time and the minimum travel time can be solved by making minor modifications to our algorithms.

To illustrate the definition, we initiate an IWRP query against the time-dependent graph in Figure 1 with the following time and weight constraints: $s = v_1, d = v_5, [t_{s1}, t_{s2}] = [0, 30]$ and $\Delta = 80$. The optimal path with the smallest $Arr(d)$ is $v_1 \to v_2 \to v_4 \to v_5$. The smallest $Arr(d)$ is 70 and the related weight is 75.

We do not consider the waiting time at the intermediate nodes while processing the IWRP query. The reasons are two-fold: (1) We could take more time to travel along a longer path but with a lower weight, which has the same effect as waiting for a route with a lower weight. (2) In many real applications, vehicles are not allowed to stop during the trip, e.g., driving on a highway.

B. Invalidity of Existing Solutions for the IWRP Problem

Most existing works on route planning over time-dependent graphs involve finding an optimal path with the earliest arrival time. We discuss their common principle and give the reasons why they cannot be used to solve our problem.

These works employ Dijkstra-like algorithms, which, however, are somewhat different from the classical Dijkstra algorithm over static graphs G_s. In the classical Dijkstra algorithm, a node v with the shortest distance from source s is selected in each iteration, and then, the neighbor nodes of v are explored. In G_s, the distance from s to v is a scalar, whereas in G_t, the distance becomes a continuous function because every edge-time function is continuous. In G_t, such a node v may not exist because the function (related to v) that is minimum over the entire domain may not exist.

To solve the problem, the Dijkstra-like algorithms on G_t employ a novel idea [13], [14], [7]: determine the latest time t_ϕ for each node v such that the current earliest arrival-time function for any time less than t_ϕ gives the actual earliest arrival time at v. Thus, for the time before t_ϕ, these algorithms can determine the unique node with the shortest distance from s and select it in each iteration. The premise behind applying this idea is that G_t has the *first-in-first-out* (FIFO) property, which means that a vehicle that enters a road segment first will also reach the end of the road segment first [10]. However,

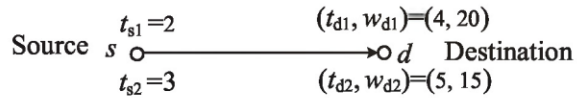


Fig. 2: Example of the invalidity of existing algorithms for the IWRP query.

the FIFO property does not hold for the IWRP query over G_t. Below is a simple example.

Example 1: Figure 2 shows a time-dependent graph G_t with only one edge e from source s to destination d. Assume that a person A departs from s either at time $t_{s1} = 2$ or at time $t_{s1} = 3$ and would like to reach d as early as possible. If G_t has the FIFO property, then we can assume that the related arrival times of A at d are $t_{d1} = 4$ and $t_{d2} = 5$, respectively. This is because $t_{s1} < t_{s2}$ implies $t_{d1} < t_{d2}$, which satisfies the FIFO property. In this case, t_{d1} is better than t_{d2} when computing the earliest arrival time. The Dijkstra-like algorithms [13], [7] on this query return t_{d1} instead of t_{d2}.

In regard to the IWRP query, this query answer is not right. Assume that it will cost A toll fees $w_{d1} = 20$ for t_{d1} and $w_{d2} = 15$ for t_{d2}. We cannot say that t_{d1} is better than t_{d2} anymore because $w_{d1} > w_{d2}$, although $t_{d1} < t_{d2}$ still holds. In other words, the answers (t_{d1}, w_{d1}) and (t_{d2}, w_{d2}) cannot be compared with each other. Algorithms on the IWRP query should return both (t_{d1}, w_{d1}) and (t_{d2}, w_{d2}). □

This example shows that we need to develop novel solutions to process the IWRP query, which is introduced in the following sections.

III. Algorithmic Framework

Procedure IWRP_Framework {
Input: G_t, $Q(s, d, t_{s1}, t_{s2}, \Delta)$
Output: $Arr(d)$, P_{sd}
(1) $\{Arr_d(t), Wgh_d(t)\}$=IWRP_Fn(G_t, Q);
/*Compute the minimum arrival and weight functions of d.*/
(2) $\{Dep(s), Arr(d)\}$=IWRP_Arr($Arr_d(t)$, $Wgh_d(t)$, Q);
/* Compute the departure and arrival times. */
(3) P_{sd}=IWRP_Path(G_t, Q, $Dep(s)$, $Arr(d)$);
/* Compute the routing path from s to d.*/ }

Fig. 3: Framework of the IWRP query algorithm.

This section introduces the framework of our approach to solve the IWRP query. This framework consists of three phases, which are denoted IWRP_Fn, IWRP_Arr and IWRP_Path. Figure 3 shows the three phases, introduced as follows:

(1) Given a time-dependent graph G_t and a query Q, IWRP_Fn computes the minimum arrival-time and weight functions of d ($Arr_d(t)$ and $Wgh_d(t)$) in G_t (line 1). The idea of IWRP_Fn is to traverse G_t from s to d, during which the minimum arrival-time and weight functions of every node v of G_t are calculated. To calculate the functions, we define a skyline function for each v and propose how to compute it. Based on the skyline function, we propose two traversal algorithms in the next two sections, namely, a baseline algorithm Base_Fn and an advanced algorithm Adv_Fn.

(2) Based on the the minimum $Arr_d(t)$ and $Wgh_d(t)$, IWRP_Arr computes the earliest arrival time $Arr(d)$ and the related departure time $Dep(s)$ satisfying Q (line 2). The minimum $Arr_d(t)$ and $Wgh_d(t)$ contain all desired information, and hence, IWRP_Arr can very easily compute $Dep(s)$ and $Arr(d)$ by inputting Q into the minimum $Arr_d(t)$ and $Wgh_d(t)$.

(3) Based on $Dep(s)$ and $Arr(d)$, IWRP_Path returns the routing path $P_{s,d}$ from s at time $Dep(s)$ to d at time $Arr(d)$ (line 3). In this phase, we can advocate the use of existing traversal algorithms (e.g., [15]) over static graphs to compute $P_{s,d}$ because we have specific departure and arrival times, based on which the minimum arrival time at every node of G_t is a scalar instead of a function.

From the discussion above, we see that IWRP_Fn is the most important phase, as IWRP_Arr and IWRP_Path can be easily calculated based on $\{Arr_d(t), Wgh_d(t)\}$ output from IWRP_Fn. Therefore, we only propose IWRP_Fn (i.e., Base_Fn and Adv_Fn) in this paper, and the detailed introductions to IWRP_Arr and IWRP_Path can be found in the full version of this paper [16].

IV. Baseline Algorithm

This section introduces the baseline algorithm (i.e., Base_Fn) to perform the first phase of the algorithmic framework. We first show how to calculate the arrival-time and weight functions for each node of G_t. After that, we present the entire process of Base_Fn.

A. Arrival-time and Weight Functions

As introduced in Section II-A, both the time and weight functions are continuous. Therefore, we construct arrival-time and weight functions for each node of G_t instead of scalar values, as for the discrete functions. Given a node $v_i \in G_t$, the arrival-time function of v_i and the weight function of v_i are denoted by $Arr_i(t)$ and $Wgh_i(t)$, respectively. Recall that the query is $Q = (s, d, t_{s1}, t_{s2}, \Delta)$. $Arr_i(t)$ monitors the arrival time at v_i via a route R that departs from s at time t. $Wgh_i(t)$ monitors the total weight of R to v_i from s at time t. Thus, the two functions of v_i can be denoted by a pair $F_i(t) = (Arr_i(t), Wgh_i(t))$. The domain of $Arr_i(t)$ (resp. $Wgh_i(t)$) is the departure time from s within the interval $[t_{s1}, t_{s2}]$. For example, $Arr_i(20) = 30$ means that a route R starts from s at time 20 and arrives at v_i at time 30. $Weight_i(20) = 600$ means that the total weight for R from s to d is 600.

In the routing algorithm, we need to minimize $Arr_i(t)$ and $Wgh_i(t)$ together. Thus, as a multi-objective optimization problem, two important concepts in Base_Fn are the *dominance relationship* and *skyline*. They are defined as follows.

Definition 2: In a two-dimensional space M with real numbers in each dimension, given two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, P_1 dominates P_2 iff $x_1 \leq x_2$ and $y_1 \leq y_2$.

A set S of points in M is called a skyline set iff no point in S is dominated by another in the same set S. We say that a point P is a skyline point if P is in a skyline set. □

In the traditional two-objective optimization problem, we need to compute the skyline set as the result [17]. A skyline point consists of two scalar values, but in Base_Fn, a skyline result consists of two continuous functions (i.e., $Arr_i(t)$ and $Wgh_i(t)$). Given $F_1(t) = (Arr_1(t), Wgh_1(t))$ and $F_2(t) = (Arr_2(t), Wgh_2(t))$, we define the *skyline function* of $F_1(t)$ and $F_1(t)$, denoted $Sky(F_1, F_2)$.

We need to calculate $Sky(F_1, F_2)$ in which the values of $F_1(t)$ and $F_1(t)$ are changed. The calculation of the skyline function is denoted $Sky(F_1, F_2) = (Sky(F_1), Sky(F_2))$, where $Sky(F_1)$ (resp. $Sky(F_1)$) is the changed result of $F_1(t)$ during the calculation.

To calculate $Sky(F_1, F_2) = (Sky(F_1), Sky(F_2))$, we first give the data structure used in the calculation.

Data Structure. As stated in Section II-A, the edge-time function $f_e(t)$ can be represented by linear piecewise functions. Specifically, $f_e(t)$ equals a set of consecutive discrete linear functions. These functions share the end points and are maintained in ascending order of time. Based on this, $f_e(t)$ is denoted as an ordered point set $f_e(t) = \{(t_1, f_e(t_1)), ..., (t_{K_e}, T_i(t_{K_e}))\}$. For example, in Figure 1 (e), $f_{3,4}(t)$ is denoted by the order set $\{(0,5), (40,15), (80, 45), (100,45)\}$. Similarly, the edge-weight function $w_e(t)$ equals a set of consecutive discrete constant functions denoted $w_e(t) = \{(t_1, w_i(t_1)), ..., (t_\sigma, w_i(t_\sigma))\}$. For example, in Figure 1 (f), $w_{3,4}(t)$ is denoted by the order set $\{(0,30), (10,10), (90,45)\}$. $Arr_i(t)$ is also piecewise linear and can be represented as an ordered set because it is constructed from the edge-time functions. Similarly, $Wgh_i(t)$ is also piecewise constant and can be represented as an ordered set because it is constructed from the edge-weight functions.

The procedure to calculate $Sky(F_1, F_2)$ is denoted Skyline. Given a set $\{F_1(t_a), F_2(t_a)\}$ at time t_a, the main idea of Skyline is to determine which is a skyline point ($F_1(t_a)$ or $F_2(t_a)$) in the set. However, either $Arr_i(t)$ or $Wgh_i(t)$ is continuous; hence, there are universal points in either function, and we cannot enumerate all points. To overcome this issue, we compare two segments that are parts of the two functions. For example, in Figure 4 (a), we can compare the two segments S_1 (OB) and S_2 (OA) between two points $[0, 10]$ on the lateral axis.

Algorithm 1 shows the details of Skyline. Skyline inputs any two pairs of functions $F_1(t) = (Arr_1(t), Wgh_1(t))$ and $F_2(t) = (Arr_2(t), Wgh_2(t))$ and outputs the skyline function $Sky(F_1, F_2) = \{Sky(F_1), Sky(F_2)\}$. Specifically, Skyline implements the following two phases:

Phase 1: compute all intersection points and breakpoints. In this phase, Skyline first constructs data structures (as given above) to represent $Arr_1(t)$, $Wgh_1(t)$, $Arr_2(t)$ and $Wgh_2(t)$ (line 1). After that, Skyline calculates all intersection points of $Arr_1(t)$ and $Arr_2(t)$ instead of $Wgh_1(t)$ and $Wgh_2(t)$ (line 4) because $Wgh_1(t)$ and $Wgh_2(t)$ contain constant values and cannot intersect each other. Skyline lastly sorts all the breakpoints (from $Arr_1(t)$, $Wgh_1(t)$, $Arr_2(t)$ and $Wgh_2(t)$) and intersection points in ascending order of time and keeps the result in a list TL (line 6).

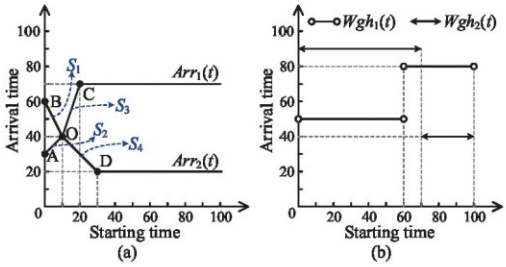


Fig. 4: Calculation of the skyline function of two pairs of functions $F_1(t)$ and $F_2(t)$.

Phase 2: determine the skyline function. For each consecutive $[t_j, t_{j+1}]$ of TL obtained in the first phase, Skyline compares two constant weights $Wgh_1(t_j) = w_1^j$ and $Wgh_2(t_j) = w_2^j$. If $w_1^j \leq w_2^j$, it compares the function values of $Arr_1(t)$ and $Arr_2(t)$ and gives the skyline result (lines 7-19). In calculating the skyline result, Skyline adopts the smaller part to prune the dominated part of functions and maintain the undominated part of functions. For example, if $Arr_1(t)$ is the smaller part, it maintains $Arr_1(t)$ and $Wgh_1(t)$ unchanged (lines 11-12) and removes the dominated part of $Arr_2(t)$ and $Wgh_2(t)$ (lines 13-14). Otherwise, it maintains $Arr_2(t)$, $Wgh_2(t)$, $Arr_1(t)$ and $Wgh_1(t)$ all unchanged (lines 16-19). If $w_1^j > w_2^j$, Skyline reverses the calculations in the steps above (lines 20-31). Finally, we can obtain the result $Sky(F_1, F_2)$ after traversing every consecutive time interval of TL.

Example 2: Figure 4(a) gives two arrival-time functions $Arr_1(t)$ and $Arr_2(t)$, and Figure 4(b) shows two weight functions $Wgh_1(t)$ and $Wgh_2(t)$. Here, $Arr_1(t)$ and $Wgh_1(t)$ form a pair $F_1(t) = (Arr_1(t), Wgh_1(t))$, and $Arr_2(t)$ and $Wgh_2(t)$ form another pair $F_2(t) = (Arr_2(t), Wgh_2(t))$. As given in Figure 4(a), $Arr_1(t)$ and $Arr_2(t)$ have two breakpoints (20, 70) and (30, 20) and one intersection point (10, 40). As shown in Figure 4(b), $Wgh_1(t)$ and $Wgh_2(t)$ only have two breakpoints (60, 80) and (70, 40) and no intersection point. To calculate $Sky(F_1, F_2)$, we sort all the breakpoints and the intersection points in increasing order of their projections on the lateral axis: (10, 40), (20, 70), (30, 20), (60, 80) and (70, 40). Then, we obtain consecutive time intervals on the lateral axis (besides the end points 0 and 100): [0, 10], [10, 20], [20, 30], [30, 60], [60, 70] and [70, 100], which finishes the first phase of Skyline.

In the second phase of Skyline, we compute a skyline function for each consecutive time interval. Here, we only show an example for one time interval; the process for other time intervals is similar. In the time interval [10, 20], we compare the related weights $Wgh_1(t) = 50$ and $Wgh_2(t) = 90$ and the related arrival-time functions $Arr_1(t)$ (of segment $S_3 = OC$) and $Arr_2(t)$ (of segment $S_4 = OD$), which are depicted in Figure 4(a). Since $Wgh_1(t) < Wgh_2(t)$ and

Algorithm 1: Skyline

Input: $\{F_1(t), F_2(t)\} = \{(Arr_1(t), Wgh_1(t)), (Arr_2(t), Wgh_2(t))\}$
Output: $Sky(F_1, F_2) = \{Sky(F_1), Sky(F_2)\}$

Construct data structures $Arr_1(t) = \{(t^i_{a_1}, a^i_1)\}$, $Wgh_1(t) = \{(t^i_{w_1}, w^i_1)\}$, $Arr_2(t) = \{(t^i_{a_2}, a^i_2)\}$, $Wgh_2(t) = \{(t^i_{w_2}, w^i_2)\}$;
Construct an empty list LT;
Insert $\{(t^i_{a_1}, a^i_1)\}, \{(t^i_{w_1}, w^i_1)\}, \{(t^i_{a_2}, a^i_2)\}, \{(t^i_{w_2}, w^i_2)\}$ into LT;
Calculate all intersection points of $Arr_1(t)$ and $Arr_2(t)$: $Int(t) = \{(t^i_{a_3}, a^i_3)\}$;
Insert $Int(t) = \{(t^i_{a_3}, a^i_3)\}$ into LT;
Sort all elements of LT in the ascending order of time;
for *each consecutive interval* $[t_j, t_{j+1}]$ *in LT as it is ordered* **do**
  if $w^j_1 \leq w^j_2$ **then**
    tmp=Min(a^j_1,a^j_2);
    if $tmp == a^j_1$ **then**
      In $Arr_1(t)$, maintain $\{(t^j_{a_1}, a^j_1)\}$ and $\{(t^{j+1}_{a_1}, a^{j+1}_1)\}$;
      In $Wgh_1(t)$, maintain $\{(t^j_{w_1}, w^j_1)\}$ and $\{(t^{j+1}_{w_1}, w^{j+1}_1)\}$;
      In $Arr_2(t)$, remove $\{(t^j_{a_2}, a^j_2)\}$ and $\{(t^{j+1}_{a_2}, a^{j+1}_2)\}$;
      In $Wgh_2(t)$, remove $\{(t^j_{w_2}, w^j_2)\}$ and $\{(t^{j+1}_{w_2}, w^{j+1}_2)\}$;
    else
      In $Arr_1(t)$, maintain $\{(t^j_{a_1}, a^j_1)\}$ and $\{(t^{j+1}_{a_1}, a^{j+1}_1)\}$;
      In $Wgh_1(t)$, maintain $\{(t^j_{w_1}, w^j_1)\}$ and $\{(t^{j+1}_{w_1}, w^{j+1}_1)\}$;
      In $Arr_2(t)$, maintain $\{(t^j_{a_2}, a^j_2)\}$ and $\{(t^{j+1}_{a_2}, a^{j+1}_2)\}$;
      In $Wgh_2(t)$, maintain $\{(t^j_{w_2}, w^j_2)\}$ and $\{(t^{j+1}_{w_2}, w^{j+1}_2)\}$;
  else
    tmp=Min(a^j_1,a^j_2);
    if $tmp == a^j_1$ **then**
      In $Arr_1(t)$, maintain $\{(t^j_{a_1}, a^j_1)\}$ and $\{(t^{j+1}_{a_1}, a^{j+1}_1)\}$;
      In $Wgh_1(t)$, maintain $\{(t^j_{w_1}, w^j_1)\}$ and $\{(t^{j+1}_{w_1}, w^{j+1}_1)\}$;
      In $Arr_2(t)$, maintain $\{(t^j_{a_2}, a^j_2)\}$ and $\{(t^{j+1}_{a_2}, a^{j+1}_2)\}$;
      In $Wgh_2(t)$, maintain $\{(t^j_{w_2}, w^j_2)\}$ and $\{(t^{j+1}_{w_2}, w^{j+1}_2)\}$;
    else
      In $Arr_1(t)$, remove $\{(t^j_{a_1}, a^j_1)\}$ and $\{(t^{j+1}_{a_1}, a^{j+1}_1)\}$;
      In $Arr_2(t)$, maintain $\{(t^j_{a_2}, a^j_2)\}$ and $\{(t^{j+1}_{a_2}, a^{j+1}_2)\}$;
      In $Wgh_1(t)$, remove $\{(t^j_{w_1}, w^j_1)\}$ and $\{(t^{j+1}_{w_1}, w^{j+1}_1)\}$;
      In $Wgh_2(t)$, maintain $\{(t^j_{w_2}, w^j_2)\}$ and $\{(t^{j+1}_{w_2}, w^{j+1}_2)\}$;

Algorithm 2: Base_Fn

Input: $G_t(V, E)$, $Q(s, , t_{s1}, t_{s2}, \Delta)$
Output: $\{(Arr_d(t), Wgh_d(t))\}$ of node d

for *(each node* v_i *in* G_t*)* **do**
  $F_i = \{Arr_i(t) = inf, Wgh_i(t) = inf\}$;
  $F_s = \{Arr_s(t) = 0, Wgh_s(t) = 0\}$; Create an entry list $L(v_i)$ for each node v_i in V, and insert F_i into $L(v_i)$;
while *any* $F_d(t) = (Arr_d(t), Wgh_d(t))$ *in* $L(d)$ *do not change)* **do**
  if *exists* $F_d(t)$ *s.t. Min(*$Wgh_d(t)$*)* $< \Delta$ **then**
    for *each edge* $e = (v_i, v_j) \in E$ *with* $f_e(t)$ *and* $w_e(t)$ **do**
      for *each* $F_i(t) = (Arr_i(t), Wgh_i(t))$ *in* $L(v_i)$ **do**
        $Arr'_j(t) = Arr_i(t) + f_e(Arr_i(t))$;
        $Wgh'_j(t) = Wgh_i(t) + W_e(Arr_i(t))$;
        for *each* $F_j(t) = (Arr_j(t), Wgh_j(t))$ *in* $L(v_j)$ **do**
          $LW(A'_j)$ =Min$Arr'_j(t)$,
          $UP(A'_j)$ =Max$Arr'_j(t)$,
          $LW(W'_j)$ =Min$Wgh'_j(t)$,
          $UP(W'_j)$ =Max$Wgh'_j(t)$;
          $LW(A_j)$ =Min$Arr_j(t)$,
          $UP(A_j)$ =Max$Arr_j(t)$,
          $LW(W_j)$ =Min$Wgh_j(t)$,
          UPW_j =Max$Wgh_j(t)$;
          if $UPA'_j < LWA_j$ *and* $UPW'_j < LWW_j$ **then**
            Remove $F_j(t) = (Wgh_j(t), Arr_j(t))$ from $L(v_j)$;
            Insert $F'_j(t) = (Wgh'_j(t), Arr'_j(t))$ into $L(v_j)$;
          if $UPA'_j > LWA_j$ *and* $UPW'_j > LWW_j$ **then**
            Discard $F'_j(t) = (Wgh'_j(t), Arr'_j(t))$;
          $\{F_1(t), F_2(t)\} = \{(Arr'_j(t), Wgh'_j(t)), (Arr_j(t), Wgh_j(t))\}$;
          $Sky(F_1, F_2)$=**Skyline**$(F_1(t), F_2(t))$;
          Insert $Sky(F_1, F_2)$ into $L(v_j)$;
return $L(d)$;

$Arr_1(t) > Arr_2(t)$, the skyline function for the time-interval [20, 30] is exactly $\{F_1(t), F_2(t)\}$. □

B. Routing Algorithm

Based on Skyline, we present the complete procedure of Base_Fn in this subsection.

1) Idea of Base_Fn: The main idea of Base_Fn is to use the *Bellman-Ford* method to find the minimum $Arr_d(t)$ and $Wgh_d(t)$ from s to node d of G_t. Instead of updating the distance from s to d, Base_Fn updates $Arr_d(t)$ and $Wgh_d(t)$ until they are stable.

Base_Fn is based on the principle of relaxation, in which approximations to the minimum $Arr_d(t)$ and $Wgh_i(d)$ are gradually replaced by more accurate functions until eventually, the two functions are stable. In Base_Fn, both approximations of $Arr_d(t)$ and $Wgh_d(t)$ to d are always overestimates of the minimum $Arr_d(t)$ and $Wgh_d(t)$ and are replaced by the skyline function of their old functions and the functions of

a newly found path. As discussed before, we cannot use the Dijkstra-like algorithm that uses a priority queue to greedily select the node with the least $Arr_i(t)$ that has not yet been processed and performs this relaxation process on all of its outgoing edges; by contrast, Base_Fn simply relaxes all the edges and does this until the arrival-time and weight functions of d do not change. In each of these repetitions, the number of nodes with correctly calculated minimum $Arr_i(t)$ and $Wgh_i(t)$ grows, from which it follows that eventually node d will has its minimum arrival-time and weight functions.

2) Algorithm Details: Algorithm 2 shows the detailed steps of Base_Fn. Simply put, Base_Fn initializes the $Arr_s(t)$ and $Wgh_s(t)$ to the source s to 0 and those to all other nodes to infinity (lines 1-2). In the meantime, for each node $v_i \in G_t$, it creates a list L_i that maintains the skyline functions (line 3). Then, Base_Fn performs a while loop, for which the halting condition is that any pair of functions in $L(d)$ are stable (line 4). When the loop stops, we can obtain the desired pairs $\{(Arr_d(t), Wgh_d(t))\}$ of d from $L(d)$ (line 20).

In the while loop, we also should guarantee that the weight constraint is satisfied (line 4). Also in the loop, for each edge $e = (v_i, v_j)$, Base_Fn updates each pair of functions F_j in $L(v_j)$ by summarizing the related pair of functions F_i in $L(v_i)$ and the edge pair of functions F_e (lines 7-20). Specifically, Base_Fn first obtains a new F'_j through calculations of F_i and F_e (addition, compounding and assignment) for pair F_i in $L(v_i)$ (lines 7-9).

After that, for each pair F_j, Base_Fn updates F'_j and F_j by calculating their skyline functions (lines 10-20). In the calculation, Base_Fn determines the smallest and largest values of any function in F'_j and F_j (lines 10-11). Based on these values, Base_Fn could prune the functions not involved in calculating the skyline function. By comparing the values, Base_Fn removes F_j from $L(v_j)$ and inserts F'_j into $L(v_j)$ if F_j is dominated by F'_j (lines 13-14); it discards F'_j if F'_j is dominated by F_j (lines 16-17). After the pruning, Skyline is invoked to compute the skyline function from candidate functions (lines 17-18) that are maintained in $L(v_j)$ (line 19). When the loop stops, we obtain all stable pairs of functions $\{Arr_d(t), Wgh_d(t)\}$ contained in $L(d)$ for d.

Example 3: Following the example in Section II-A, we initiate an IWRP query over the time-dependent graph G_t in Figure 1 with the following time and weight constraints: $s = v_1$, $d = v_5$, $[t_{s1}, t_{s2}] = [0, 30]$, and $\Delta = 80$.

We process the edges of G_t in the order e_1, e_2, e_3, e_4 and e_5. We have no functions for v_1 since it is the source node. To process $e_1 = (v_1, v_2)$, we obtain the functions of v_2, $Arr_2(t)$ by $t + f_{1,2}(t)$ and $Wgh_2(t)$ by $0 + w_{1,2}(t)$. The result of $Arr_2(t)$ is given in Figure 5(a), and $Wgh_2(t)$ is just the figure shown in Figure 1(a). Similarly, to process $e_2 = (v_1, v_3)$, we obtain the functions of v_3, $Arr_3(t)$ shown in Figure 5(b) and $Wgh_3(t)$, which is the same as $w_{1,3}(t)$ shown in Figure 1(b). To process $e_3 = (v_2, v_4)$, we obtain the functions of v_4, $Arr_4(t)$ by compounding $Arr_2(t)$ and $f_{2,4}(t)$ and $Wgh_2(t)$ by $Wgh_2(t)+w_{2,4}(t)$. To process $e_4 = (v_3, v_4)$, we first obtain new functions of v_4, $Arr'_4(t)$ by compounding

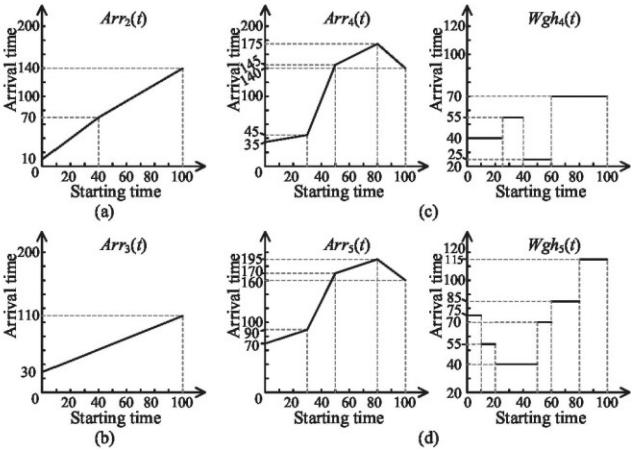


Fig. 5: Running example of Base_Fn over the time-dependent graph shown in Figure 1.

$Arr_3(t)$ and $f_{3,4}(t)$ and $Wgh'_4(t)$ by $Wgh_3(t)+w_{3,4}(t)$. After that, we obtain the final functions of v_4 by computing the skyline function as shown in Figure 5(c). Finally, we obtain the functions of v_5, $Arr_5(t)$ by compounding $Arr_4(t)$ and $f_{4,5}(t)$ and $Wgh_5(t)$ by $Wgh_4(t) + w_{4,5}(t)$. The result is given in Figure 5(d). □

For $v_i \in G_t$, let $Arr_i(t)$ and $Wgh_i(t)$ have average breakpoints k_t and k_w, respectively, after Base_Fn finishes. For analysis of Base_Fn, we have the following theorem:

Theorem 1:

- Base_Fn outputs the corrected minimum functions $\{F_d(t)\} = \{(Arr_d(t), Wgh_d(t))\}$.
- Base_Fn has $O((k_t + k_w)n^3m^2M_e^2)$ time complexity, which is the time complexity of IWRP_Framework. □

The proof can be found in the full version of this paper [16].

V. Advanced Routing Algorithm

Base_Fn has a high time complexity because of its expensive traversal manner over G_t. To solve this problem, in this section, we propose an advanced traversal algorithm (denoted Adv_Fn) to compute $\{Arr_d(t), Wgh_d(t)\}$ in IWRP_Framework such that the time complexity of IWRP_Framework is reduced from $O((k_t + k_w)n^3m^2M_e^2)$ to $O(M_e(k_t + k_w)nm \log(nM_e))$.

A. Algorithm of Adv_Fn

Idea of Adv_Fn. The main idea of Adv_Fn is to use the *Dikjstra* method to find the minimum $Arr_i(t)$ and $Wgh_i(t)$ from s to every other node v_i of G_t. Specifically, during the traversal from s, Adv_Fn maintains a label set $L(v)$ for each node v, which contains the current set of skyline functions (related to all paths) from the source s to v. Adv_Fn is not finished until Adv_Fn traverses all the paths from s to destination d.

As shown in Section II-B, the existing Dijkstra-like algorithms cannot solve the IWRP problem. However, the idea of Adv_Fn is different from the existing Dijkstra-like algorithms

in that Adv_Fn traverses every edge instead of every node, as in the existing Dijkstra-like algorithms. This small difference makes Adv_Fn process the IWRP query correctly and efficiently.

Algorithm Details. Algorithm 3 shows the detailed steps of Adv_Fn. Similar to Base_Fn, Adv_Fn initializes the $Arr_s(t)$ and $Wgh_s(t)$ to the source s to 0 and those to all other nodes to infinity (lines 1-2). In the meantime, for each node $v_i \in G_t$, it creates a list L_i that maintains the skyline functions (line 3). Adv_Fn also constructs a data structure min-heap H, which stores the paths in ascending order of the smallest value of $Arr_i(t)$, i.e., Min($Arr_i(t)$) (line 4). Initially, H contains only one trivial path with no edge, which both starts and ends at the source node s. Then, Adv_Fn utilizes H to iteratively compute the skyline function for every other node v_i originated from s (lines 5-18).

In each iteration, Adv_Fn pops the top path P from H (line 6). Let v_i be the last node in P. We can stop the current iteration (line 8) if v_i is the destination d and the current $Wgh_d(t)$ is beyond the weight constraint Δ (line 7). Otherwise, $\langle P, Sky_i(t) = (Arr_i(t), Wgh_i(t))\rangle$ is inserted into $l(v_i)$ as a candidate path (line 9). Adv_Fn enumerates each new path P_{new} that can be obtained by appending an edge (v_i, v_j) at the end of P (lines 10-11) and obtains a new pair of functions $F_1(t) = (Arr'_j(t), Wgh'_j(t))$ for v_j (lines 12-13). After that, Adv_Fn computes the skyline function (i.e., Skyline) between $F_1(t)$ and every other pair of functions $(F_2(t) = (Arr_j(t), Wgh_j(t)))$ in $L(v_j)$ and obtains the new functions ($Sky(F_1)$ and $Sky(F_2)$) for $F_1(t)$ and $F_2(t)$ (lines 15-16). At the end of each iteration, Adv_Fn updates $L(v_j)$ by inserting the new functions ($Sky(F_1)$ and $Sky(F_2)$) into it and pushes P_{new} and its new function $Sky(F_1)$ into H for the next iteration (lines 17-18).

Adv_Fn terminates when H becomes empty (line 5). Finally, Adv_Fn returns all pairs of functions maintained in $L(d)$ (line 19).

Example 4:

As in Example 3, we initiate an IWRP query against the time-dependent graph G_t in Figure 1 with the following time and weight constraints: $s = v_1$, $d = v_5$, $[t_{s1}, t_{s2}] = [0, 30]$, and $\Delta = 80$.

Adv_Fn first traverses edges $e_1 = (v_1, v_2)$ and $e_2 = (v_1, v_3)$ from v_1, and we obtain the functions of v_2 (i.e., $Arr_2(t)$ and $Wgh_2(t)$) and the functions of v_3 (i.e., $Arr_3(t)$ and $Wgh_3(t)$). The results of $Arr_2(t)$ and $Wgh_2(t)$) are given in Figures 5(a) and 1(a), respectively. Additionally, the results of $Arr_3(t)$ and $Wgh_3(t)$ are given in Figures 5(b) and 1(b), respectively. In the second step, Adv_Fn expands v_2 and processes edges $e_3 = (v_2, v_4)$ since Min($Arr_2(t)$) = 10 <Min($Arr_3(t)$) = 30. In this step, we obtain the functions of v_4, $Arr_4(t)$ and $Wgh_4(t)$ shown in Figure 6(a). In the third step, Adv_Fn expands v_4 and processes edge $e_5 = (v_4, v_5)$ since Min($Arr_4(t)$) = 35 is the smallest. In this step, we obtain the functions of v_5, $Arr_5(t)$ and $Wgh_5(t)$ shown in Figure 6(b).

Algorithm 3: Adv_Fn

Input: $G_t(V, E)$, $Q(s, , t_{s1}, t_{s2}, \Delta)$
Output: $\{(Arr_d(t), Wgh_d(t))\}$ of d

1 **for** *(each node v_i in G_t)* **do**
2 $F_i = \{Arr_i(t) = inf, Wgh_i(t) = inf\}$; $F_s = \{Arr_s(t) = 0, Wgh_s(t) = 0\}$;
3 Create an entry list $L(v_i)$ for each node v_i in V, and insert F_i into $L(v_i)$;
4 Create a min-heap H with entries in the form $\rho = \langle P, Sky_i(t) = (Arr_i(t), Wgh_i(t))\rangle$, sorted in ascending order of Min($Arr_i(t)$);
5 **while** *H is not empty* **do**
6 Pop the top entry $\rho = \langle P, Sky_i(t)\rangle$ from H;
7 **if** *$v_i = d$ and Min($Wgh_d(t)$) > Δ* **then**
8 **continue**;
9 Insert ρ into $L(v_i)$;
10 **for** *each outgoing edge $e = (v_i, v_j)$ of v_i with $f_e(t)$ and $w_e(t)$* **do**
11 Construct path P_{new} by extending P with e;
12 $Arr'_j(t) = Arr_i(t) + f_e(Arr_i(t))$;
13 $Wgh'_j(t) = Wgh_i(t) + W_e(Arr_i(t))$;
14 **for** *each $F_j(t) = (Arr_j(t), Wgh_j(t))$ in $L(v_j)$* **do**
15 $\{F_1(t), F_2(t)\} = \{(Arr'_j(t), Wgh'_j(t)), (Arr_j(t), Wgh_j(t))\}$;
16 $(Sky(F_1), Sky(F_2))$=**Skyline**$(F_1(t), F_2(t))$;
17 Insert $Sky(F_1)$ and $Sky(F_2)$ into $L(v_j)$;
18 Push $\langle P_{new}, Sky(F_1)\rangle$ into H;
19 **return** $L(d)$;

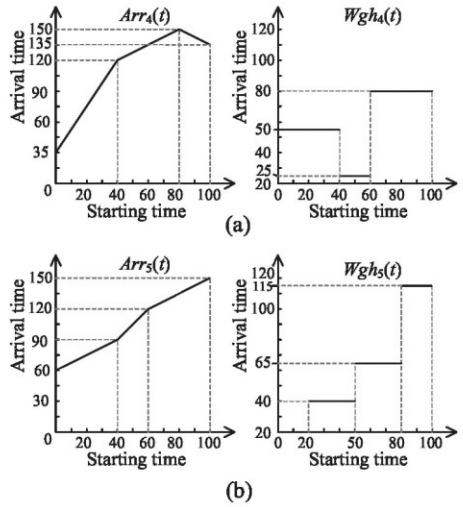


Fig. 6: Running example of Adv_Fn over the time-dependent graph in Figure 1.

Now, Adv_Fn has not finished, as edge $e_4 = (v_3, v_4)$ has not been processed. In the fourth step, Adv_Fn expands v_3 again and processes edge e_4. We obtain new functions of v_4, $Arr'_4(t)$ and $Wgh'_4(t)$. After that, we obtain the final functions of v_4 by computing the skyline function, as shown in Figure 5(c). Finally, Adv_Fn expands v_4 and processes edge $e_5 = (v_4, v_5)$ again, and the results of $Arr_5(t)$ and $Wgh_5(t)$

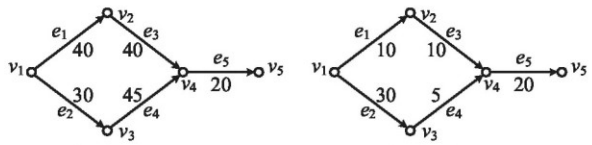


Fig. 7: Graphs G_u and G_l of time-dependent graph G_t in Figure 1.

are shown in Figure 5(d). □

Theorem 2:

- Adv_Fn outputs the corrected minimum functions $\{F_d(t)\} = \{(Arr_d(t), Wgh_d(t))\}$ for destination d.
- Adv_Fn has $O((k_t + k_w)M_e nm \log(nM_e))$ time complexity, which is also the time complexity of IWRP_Framework by employing Adv_Fn in it. □

The proof can be found in the full version of this paper [16].

B. Speeding Up Adv_Fn

We can further speed up Adv_Fn by a heuristic approach, which is introduced below.

We first define some concepts. For an edge $e \in G_t$, we define $U(e)$ as the largest value of function $f_e(t)$ and $L(e)$ as the smallest value of $f_e(t)$. For a time-dependent graph $G_t = (V_t, E_t, f_e(t))$, we also define static graphs $G_u(V_u, E_u, F_u)$ as $V_u = V_t$, $E_u = E_t$ and $F_u(e) = U(e)$ and $G_l(V_l, E_l, F_l)$ as $V_l = V_t$, $E_l = E_t$ and $F_l(e) = L(e)$. Intuitively, for an edge $e_u \in G_u$, $F_u(e_u)$ maintains the upper bound on the time function of $e_t \in G_t$. Similarly, for an edge $e_l \in G_l$, $F_l(e_l)$ maintains the lower bound on the time function of $e_t \in G_t$.

For example, Figure 7 shows the static graphs G_u and G_l of time-dependent graph G_t in Figure 1. As shown in G_u, the value (40) assigned to edge e_1 is the upper bound of $f_{e_1}(t)$ in Figure 1. Similarly, the value (10) assigned to edge e_1 in G_l is the lower bound of $f_{e_1}(t)$ in Figure 1.

Heuristic Approach. Now, we provide the speeding up approach, denoted SAdv_Fn. Recall that in each iteration of Algorithm 3, SAdv_Fn expands the node with the smallest Min($Arr_i(t)$) among all non-expanded nodes. Now, SAdv_Fn will expand the node with the smallest value of Min($Arr_i(t)$)+($D_l(v_i, d)$+$D_u(v_i, d)$)/2. Here, $D_l(v_i, d)$ is the shortest distance from v_i to d in G_l, and $D_u(v_i, d)$ is the shortest distance from v_i to d in G_u. Both $D_l(v_i, d)$ and $D_u(v_i, d)$ can be computed in advance. SAdv_Fn terminates when the destination d is first expanded.

Denote by $Arr_{id}(t)$ the arrival-time function from v_i to d. Then, we have $D_l(v_i, d) \leq Arr_{id}(t) \leq D_u(v_i, d)$. Thus, intuitively, SAdv_Fn expands v_i closer to d among all non-expanded nodes. The final query results change a little, as shown in the experiments, but the order in which the nodes are expanded can change a lot. If d is expanded earlier, we have to expand fewer nodes before the computation stops. This can save much running time.

VI. Performance Evaluation

A. Experimental Settings

Datasets. We employ the following two real road networks.

MA: This dataset is the Maine road network, including 187,315 nodes and 422,998 edges. A node represents an intersection or a road endpoint, and an edge represents a road segment.

W-US: This network describes the Western USA road network, and it includes 6,262,104 nodes and 15,248,146 edges. As for MA, a node represents an intersection or a road endpoint, and an edge represents a road segment.

F-US: This network describes the full USA road network, and it contains 23,885,296 nodes and 58,327,516 edges.

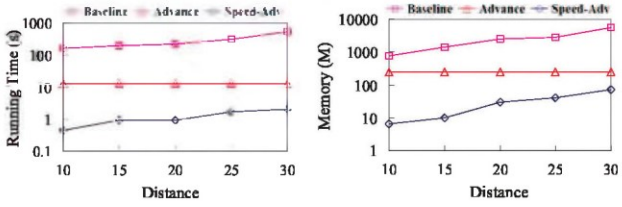
Following the work [11], we generate time-dependent graphs using the MA and W-US datasets as follows. We first generate the travel time according to the road length. The travel time for an edge (u, v) is greater if the road represented by (u, v) is longer. The time domain is set as $T = [0, 2000]$, i.e., the departure time t can be selected from [0, 2000] for source node. Here, 2000 means 2000 time units. For every $w_e(t)$, we split the time domain T into k subintervals and assign a constant value randomly for every subinterval; it is then a piecewise-constant function. For every $f_e(t)$, the time domain T is also randomly divided into k subintervals $([t_0, t_1], [t_1, t_2], \cdots, [t_{k-1}, t_k])$, where t_0 and t_k are the start and end of the time domain T, respectively. The value of $f_e(t_0)$ is first generated as a random number from $[0, \bar{f}]$, where $\bar{f}$ is a number to restrict the max value of $f_e(t)$. Within each subinterval $[t_{x-1}, t_x]$ $(1 \leq x \leq k)$, $f_e(t)$ is a linear function $f_e^x(t)$, $f_e^x(t_{x-1}) = f_e^{x-1}(t_{x-1})$, and $f_e^x(t_x)$ is generated as a random number from $[max(0, f_e^x(t_{x-1}) - \triangle t_x), \bar{f}]$, where $\triangle t_x = t_x - t_{x-1}$.

Algorithms. We evaluate the baseline routing algorithm, the advanced routing algorithm and the speeding up of the advanced algorithm in processing IWRP queries over time-dependent graphs. The three algorithms are denoted Baseline, Advance and Speed-Adv, respectively.

Metrics. We are interested in the following aspects to evaluate the performances of Baseline, Advance and Speed-Adv: (1) the impact of the number of nodes ($|V_t|$), (2) the impact of the number of edges ($|E_t|$), (3) the impact of the distance (d_s) between source and destination, (4) the impact of the length (l_s) of the starting time interval $[t_{s1}, t_{s2}]$ (i.e., $l_s = t_{s2} - t_{s1}$), and (5) the impact of the average numbers (n_f and n_w) of segments of $f_e(t)$ and $w_e(t)$. The parameters to be evaluated are the (1) querying time and (2) memory overhead.

B. Experimental Results

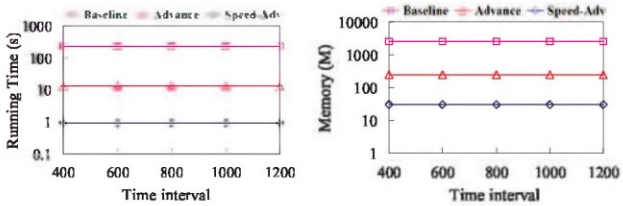
Exp-1: varying d_s. We vary d_s from 10 to 30 on the MA and W-USA graphs. In this test, we set $l_s = 800$, $n_f = 15$ and $n_w = 15$. As shown in Figure 8, (1) Speed-Adv has the highest efficiency and least memory overhead, followed by Advance and then by Baseline. (2) Both Baseline and Speed-Adv consume more time and memory overhead as d_s increases, as longer distances require more computations to



(a) MA (b) MA

(c) W-US (d) W-US

Fig. 8: Impact of the distance (d_s) between the source and destination.



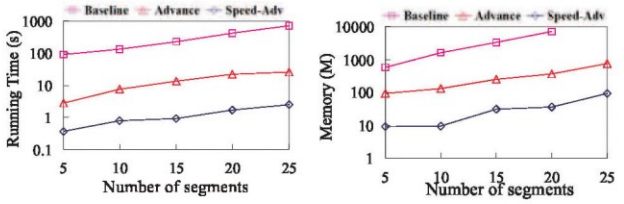
(a) MA (b) MA

(c) W-US (d) W-US

Fig. 9: Impact of the length (l_s) of time interval $[t_{s1}, t_{s2}]$.

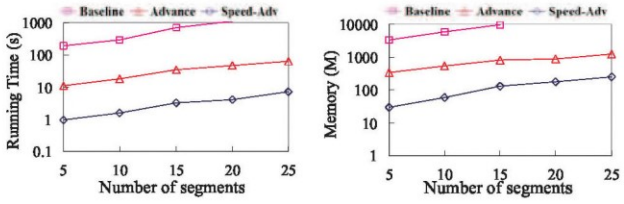
reach the destination in Baseline and Speed-Adv. Advance is not affected by the change of d_s due to Advance computing the arrival and weight functions from s to all other nodes.

Exp-2: varying l_s. We vary l_s from 400 to 1200 on the MA and W-USA graphs. In this test, we set $d_s = 20$, $n_f = 15$ and $n_w = 15$. As shown in Figure 9, (1) Speed-Adv is 13 and 220 times faster than Advance and Baseline and consumes 1/8 and 1/80 of the memory overhead of Advance and Baseline. This is because the Dijkstra-like traversal in Advance is much faster than the Bellman-Ford-like traversal in Baseline, and



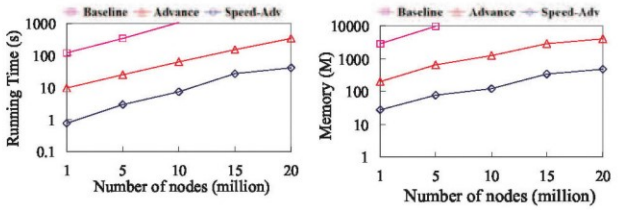
(a) Query time (b) Memory overhead

Fig. 10: Impact of the number of segments (n_f) of edge-time functions on the MA graph.



(a) Query time (b) Memory overhead

Fig. 11: Impact of the number of segments (n_w) of edge-weight functions on the W-US graph.



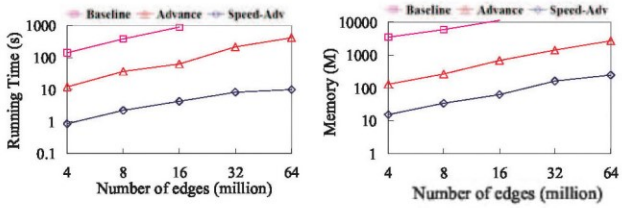
(a) Query time (b) Memory overhead

Fig. 12: Impact of node size $|V_t|$.

Speed-Adv further speeds up Advance by adding heuristics. (2) Both the querying time and memory overhead of Speed-Adv, Advance and Advance are not affected, as both $f_e(t)$ and $w_e(t)$ do not change.

Exp-3: varying n_f and n_w. We vary n_f and n_w from 5 to 25 on the MA graph and W-USA graphs, respectively. In this test, we set $d_s = 20$ and $l_s = 800$. As shown in Figure 10 and 11, (1) the querying time and memory overhead of Baseline, Advance and Speed-Adv increase with increasing number of segments (i.e., n_f and n_w). The reason is that Skyline will require more calculations in the three algorithms when n_f and n_w increase. (2) All curves on n_f increase more rapidly than those on n_w, as the change in n_f directly affects $Arr_d(t)$, which is the query objective.

Exp-4: varying $|V_t|$. We vary $|V_t|$ from 1M to 20M, where graphs with 1M to 20M nodes are generated from the F-US



(a) Query time (b) Memory overhead

Fig. 13: Impact of edge size $|E_t|$.

dataset. In this test, we set $d_s = 20$, $l_s = 800$, $n_f = 15$ and $n_w = 15$. As shown in Figure 12, (1) the advanced algorithm yields much better running and storage efficiencies than the baseline algorithm. For example, Speed-Adv and Advance are nearly 260 and 50 times faster than Baseline. For the memory overhead, the improvements can reach 180 and 30 times. (2) The querying time and memory overhead of Speed-Adv increase slowly with increasing number of nodes. The curves of the querying time and memory overhead for Baseline increase very rapidly and exceed 1000 s and 10G for the graph with 10M and 5 nodes, respectively. However, the curves for Speed-Adv and Advance become smooth, especially for Speed-Adv. This experimental result shows that Speed-Adv scales well with the node number.

Exp-5: varying $|E_t|$. We vary $|E_t|$ from 4M to 64M by fixing $|V_t|$ =2M on the F-US dataset. In this test, we set $d_s = 20$, $l_s = 800$, $n_f = 15$ and $n_w = 15$. As shown in Figure 13, (1) the querying time and memory overhead of Speed-Adv increase slightly and are 4.2 s and 61M for the graph with 16 million edges. The querying time and memory overhead of Baseline grow exponentially and are greater than 1,000s and 10G for the graph with 16 million edges. (2) All curves for increasing $|V_t|$ grow more obviously compared to those for increasing $|E_t|$. This is because the traverses by all algorithms converge more rapidly, as the average distance of graph becomes shorter with the increasing of $|E_t|$.

The results of all of the above experiments justify that Speed-Adv is efficient and lightweight and scales well with all metrics. Additionally, Speed-Adv almost returns optimal query answers, although Speed-Adv is a heuristic algorithm.

VII. Related Work

We categorize the related work as follows.

Route planning over time-dependent graphs. These works are classified into two categories: one category is based on the discrete time model, and the other one is based on the continuous time model. A basic version of the discrete time model [18], [19] contains a node for every departure and arrival event, with consecutive departure and arrival events connected by connection (or travel) edges. A few route planning algorithms, such as the earliest-arrival-time path, the latest-departure-time path, and the shortest- duration-time path, have been proposed for such graphs. Cooke et al. [20] proved that these queries could be solved with a modified version of Dijkstra's algorithm. However, it does not scale well with the size of the graph, and hence, several techniques, such as indexing, have been proposed to improve the efficiency [8], [21], [22], [23]. A few works [24], [25] aimed to optimize the earliest arrival time and the number of transfers on discrete time-dependent graphs. Their methods cannot solve our problem, as the other optimized objective is different (ours is the weight-optimal objective) and the graph models are different (ours is continuous).

Several studies in the field of operations research consider the weight-optimal path problem under the discrete time model [26]. They develop dynamic programming schemes to obtain the exact solution, but their time complexities are very high. Close to our work is [11]: they define a discrete time function $f_{i,j}(v_i, v_j)$ and a discrete weight function $w_{i,j}(v_i, v_j)$ for each edge (v_i, v_j) and aim to find the path with minimum weight, not the minimum time.

A more precise method to describe a time-dependent traffic network is to use the continuous time-dependent function. The earliest arrival time can also be computed by a Dijkstra-like algorithm if the FIFO property holds for the continuous model [20], [13], [7]. The algorithm in [7] is the most efficient because it applies a more precise refinement approach that expands the time interval step by step rather than computing the entire time interval iteratively. The recent work [14] studied the shortest- duration-time path problem without the FIFO property, as it allows waiting at intermediate nodes during the route. However, it assumes that every edge function still has the FIFO property, and thus, the problem can be easily solved by a Dijkstra-like algorithm. On the other hand, the earliest-arrival-time path cannot be computed by this algorithm because the entire graph does not have the FIFO property. Other works further build different indices to speed up the query, such as time-dependent CH [27] and time-dependent SHARC [28].

Our work differs from previous works in the following aspects. Our problem minimizes both time and weight objectives such that it has a non-FIFO characteristic (detailed in Section II-B), whereas all the previous works assume the FIFO property. Thus, we develop completely novel techniques to address the difficult problem incurred by the non-FIFO characteristic.

Weight-constrained route planning over static graphs. The WRP is a classical NP-complete problem. Joksch [29] first studied the WRP problem and proposed a dynamic programming algorithm for the exact WRP. Subsequently, Handler and Zang [1] proposed two methods for exact WRP processing: one method formulated the WRP as an integer linear programming (ILP) problem and solved it with a standard ILP solver. This same methodology was used by Mehlhorn and Ziegelmann [30]. The state-of-the-art solution for the exact WRP problem is the one proposed in [15], which followed the general idea of Dijkstra's algorithm. To address the hard problem, Hansen [15] proposed the first c-approximate solution, which runs in polynomial time

but has a high complexity: $O(m^2 \frac{n^2}{c-1} \log \frac{n}{c-1})$. Lorenz and Raz [31] lowered the complexity. However, this solution is orders of magnitude slower than an exact WRP algorithm, as shown in [32]. Wang et al. [5] speed up the c-approximate WRP query by proposing an effective index. The works [33], [34] study the routing problems over road networks from practice to theory. All of these algorithms only work for static graphs and cannot address dynamic or time-dependent graphs.

VIII. Conclusion

We have proposed an IWRP query by extending traditional static weight-constrained route planning to time-dependent graphs. The IWRP query aims to find an optimal path with the earliest arrival time from the source to the destination while keeping the total weight within a budget when the departure time from the source can be selected from a user-given starting-time interval. We have developed baseline and advanced algorithms to process the IWRP query, but the advanced algorithm is more efficient. Our experimental study verified the feasibility of our proposed algorithms for real-life graphs.

The IWRP query has a non-FIFO characteristic, and thus, our two proposed algorithms could process queries over non-FIFO graphs. We will study this in more detail. We will also study parallel scalable algorithms and indexing techniques to speed up the IWRP query over large time-dependent graphs.

IX. Acknowledgment

Ye Yuan is supported by the NSFC (Grant No. 61572119 and 61622202) and the Fundamental Research Funds for the Central Universities (Grant No. N150402005). Lian Xiang is supported by NSF OAC No. 1739491 and Lian Start Up No. 220981, Kent State University. Guoren Wang is supported by the NSFC (Grant No. U1401256 and 61732003). Lei Chen is partially supported by the Hong Kong RGC GRF Project 16214716, the National Science Foundation of China (NSFC) under Grant No. 61729201, Science and Technology Planning Project of Guangdong Province, China, No. 2015B010110006, Hong Kong ITC ITF grants ITS/391/15FX and ITS/212/16FP, Didi-HKUST joint research lab project, Microsoft Research Asia Collaborative Research Grant and Wechat Research Grant.

References

[1] G. Y. Handler and I. Zang, "A dual algorithm for the constrained shortest path problem," *Networks*, vol. 10, no. 4, pp. 293–309, 1980.

[2] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36–42, 1992.

[3] D. H. Lorenz and D. Raz, "A simple efficient approximation scheme for the restricted shortest path problem," *Operations Research Letters*, vol. 28, no. 5, pp. 213–219, 2001.

[4] G. Tsaggouris and C. Zaroliagis, "Multiobjective optimization: Improved fptas for shortest paths and non-linear objectives with applications," *Theory of Computing Systems*, vol. 45, no. 1, pp. 162–186, 2009.

[5] S. Wang, X. Xiao, Y. Yin, and W. Lin, "Effective indexing for approximate constrained shortest path queries on large road networks," *Proceedings of the Vldb Endowment*, vol. 10, no. 2, pp. 61–72, 2016.

[6] A. Orda and R. Rom, "Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length," *Journal of the Acm*, vol. 37, no. 3, pp. 607–625, 1990.

[7] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *EDBT*, 2008, pp. 205–216.

[8] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proceedings of the Vldb Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[9] S. Shang, D. Guo, J. Liu, and J. R. Wen, "Prediction-based unobstructed route planning," *Neurocomputing*, vol. 213, pp. 147–154, 2016.

[10] L. Foschini, J. Hershberger, and S. Suri, "On the complexity of time-dependent shortest paths." in *Acm-Siam Symposium on Discrete Algorithms*, 2011, pp. 327–341.

[11] Y. Yang, H. Gao, J. X. Yu, and J. Li, "Finding the cost-optimal path with time constraint over time-dependent graphs," *Proceedings of the Vldb Endowment*, vol. 7, no. 9, pp. 673–684, 2014.

[12] C. S. Peskina, "Numerical analysis of blood flow in the heart," *Journal of Computational Physics*, vol. 25, no. 3, pp. 220–252, 2015.

[13] Dean and C. Brian, "Continuous-time dynamics shortest path algorithms," *Massachusetts Institute of Technology*, 1999.

[14] L. Li, W. Hua, X. Du, X. Zhou, L. Li, W. Hua, X. Du, and X. Zhou, "Minimal on-road time route scheduling on time-dependent graphs," *Proceedings of the Vldb Endowment*, vol. 10, no. 11, pp. 1274–1285, 2017.

[15] P. Hansen, "Bicriterion path problems," *Lecture Notes in Economics & Mathematical Systems*, vol. 177, pp. 109–127, 1980.

[16] http://faculty.neu.edu.cn/ise/yuanye/full.pdf.

[17] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *SIGMOD*, 2003.

[18] F. Schulz, D. Wagner, and K. Weihe, "Dijkstra's algorithm on-line: An empirical case study from public railroad transport," *Journal of Experimental Algorithmics*, vol. 5, no. 2, p. 12, 2000.

[19] S. Shang, J. Liu, Z. Kai, L. Hua, T. B. Pedersen, and J. R. Wen, "Planning unobstructed paths in traffic-aware spatial networks," *Geoinformatica*, vol. 19, no. 4, pp. 723–746, 2015.

[20] K. L. Cooke and E. Halsey, "The shortest route through a network with time-dependent internodal transit times," *Journal of Mathematical Analysis & Applications*, vol. 14, no. 3, pp. 493–498, 1997.

[21] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks:a labelling approach," in *ACM SIGMOD International Conference on Management of Data*, 2015, pp. 967–982.

[22] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *IEEE International Conference on Data Engineering*, 2016, pp. 145–156.

[23] S. Shang, S. Zhu, D. Guo, and M. Lu, "Discovery of probabilistic nearest neighbors in traffic-aware spatial networks," *World Wide Web*.

[24] G. S. Brodal and R. Jacob, "Time-dependent networks as models to achieve fast exact time-table queries," *Electronic Notes in Theoretical Computer Science*, vol. 92, pp. 3–15, 2004.

[25] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis, "Efficient models for timetable information in public transportation systems," *Journal of Experimental Algorithmics*, vol. 12, no. 1, p. 2.4, 2008.

[26] X. Cai, T. Kloks, and C. K. Wong, "Time-varying shortest path problems," *Networks*, vol. 29, no. 3, pp. 141–150, 2015.

[27] G. V. Batz, D. Delling, P. Sanders, and C. Vetter, "Time-dependent contraction hierarchies." in *Meeting on Algorithm Engineering & Expermiments*, 2009, pp. 97–105.

[28] D. Delling, "Time-dependent sharc-routing," in *European Symposium on Algorithms*, 2008, pp. 332–343.

[29] H. C. Joksch, "The shortest route problem with constraints," *Journal of Mathematical Analysis and Applications*, vol. 14, no. 2, pp. 191–197, 1966.

[30] K. Mehlhorn and M. Ziegelmann, "Resource constrained shortest paths," in *European Symposium on Algorithms*, 2000, pp. 326–337.

[31] . D. H. Lorenz and D. R. Raz, "A simple efficient approximation acheme for the restricted shortest path problem," in *Operations Research Letters*, 1999, pp. 213–219.

[32] F. Kuipers, A. Orda, D. Raz, and P. V. Mieghem, "A comparison of exact and -approximation algorithms for constrained routing," in *International Conference on Research in Networking*, 2006, pp. 197–208.

[33] A. D. Zhu, M. Hui, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: Towards bridging theory and practice," in *SIGMOD*, 2013.

[34] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou, "Shortest path and distance queries on road networks: An experimental evaluation," *PVLDB*, vol. 5, no. 5, pp. 406–417, 2012.