Constrained Shortest Path Query in a Large Time-Dependent Graph

Ye Yuan[§] Xiang Lian[†] Guoren Wang[§] Yuliang Ma[♯] Yishu Wang[♯]
[§]*Beijing Institute of Technology* [†]*Kent State University* [♯]*Northeastern University, China*
yuanye@mail.neu.edu.cn, xlian@kent.edu, wanggrbit@126.com, ylma.neuer@gmail.com, yishu_w0124@163.com

ABSTRACT

The constrained shortest path (CSP) query over static graphs has been extensively studied, since it has wide applications in transportation networks, telecommunication networks and etc. Such networks are dynamic and evolve over time, being modeled as *time-dependent graphs*. Therefore, in this paper, we study the CSP query over a large time-dependent graph. Specifically, we study the point CSP (PCSP) query and interval CSP (ICSP) query. We formally prove that it is NP-complete to process a PCSP query and at least EXPSPACE to answer an ICSP query. We propose approximate sequential algorithms to answer the PCSP and ICSP queries efficiently. We also develop parallel algorithms for the queries that guarantee to scale with big time-dependent graphs. Using real-life graphs, we experimentally verify the efficiency and scalability of our algorithms.

PVLDB Reference Format:
Ye Yuan, Xiang Lian, Guoren Wang, Yuliang Ma, and Yishu Wang. Constrained Shortest Path Query in a Large Time-Dependent Graph. *PVLDB*, 12(10): 1058-1070, 2019.
DOI: https://doi.org/10.14778/3339490.3339491

1. INTRODUCTION

The constrained shortest path (CSP) query over a graph is to find the best path from source to destination based on one criterion with a constraint on another criterion [14, 16]. The CSP query in static graphs has been studied extensively [14, 16, 21, 29] because it has wide applications. In route planning over transportation networks, a traveler has a tour plan to Beijing with maximum budgets on different reimbursement categories. His/her travel budget is 1,000 RMB, his/her accommodation budget is 5,000 RMB, and other budget is 6,000 RMB. Thus, he may want to compute a shortest route to Beijing with toll payment within 1,000 RMB. In this scenario, he should compute a CSP query that minimizes the total travel time within the budget for toll payment. In an online navigation system, the constraint can be presented to the user in the form of a slider bar, which drastically simplifies user-system interactions. In telecommunication networks, a routing algorithm not only computes a fastest route, but also guarantees the packet loss rate within a threshold for a reliable transmission [28]. In reality, graphs often evolve over time. For example, buses and trains run at different frequencies on schedule-based public transportation systems, and road networks are consistently congested during rush hours. The Vehicle Information and Communication System (VICS) and the European Traffic Message Channel (TMC) are two transportation systems, which can provide real-time traffic information to users. Such transportation networks are *time-dependent graphs*, i.e., the travel time for a road varies over time. Therefore, in this paper, we study the CSP query over a large time-dependent graph G_t.

Proceedings of the VLDB Endowment, Vol. 12, No. 10
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3339490.3339491

Every edge $e = (u, v)$ in G_t has two types of costs: $f_e(t)$ and $w_e(t)$. $f_e(t)$ is the time cost for specifying how long it takes to travel through an edge e, and $w_e(t)$ is the weight (e.g, the toll fee) for traveling through an edge e. Both $f_e(t)$ and $w_e(t)$ are functions that are dependent on the departure time t at the starting endpoint u of the edge $e = (u, v)$.

When $f_e(t)$ is discrete, we refer G_t to a discrete time-dependent graph. When $f_e(t)$ is continuous, we refer G_t to a continuous time-dependent graph. In this paper, we consider a continuous time-dependent graph for two reasons. First, continuous G_t is a general model, and discrete G_t is a special case of continuous G_t. Second, a continuous G_t can model many real networks, e.g., road networks [19], schedule-based public transportation networks [32] and computer networks [24].

The query types over continuous time-dependent graphs include the point query and the interval query. The point query computes the shortest path for a departure time point, while the interval query has the departure time within a period [24]. In this paper, we study point CSP (PCSP) queries and interval CSP (ICSP) queries over continuous time-dependent graphs. Below is an example.

Example 1. *Figure 1 shows a continuous time-dependent graph G_t with time-function $f_e(t)$ and weight-function $w_e(t)$ assigned to every edge e of G_t. In Figure 1(c), $f_{e_2}(t)$ is the time-function of edge e_2, which is a piece-wise linear function. Also in this figure, $w_{e_2}(t)$ is the weight-function of edge e_2, which is a piece-wise constant function.*

Assume that a person P would like to travel from a source node $s = v_1$ to a destination node $d = v_3$ in G_t. For a PCSP query, we consider the following scenario. Given a specific departure time t_s from s and a budget constraint Δ, P would like to compute the earliest arrival time point at d, but takes

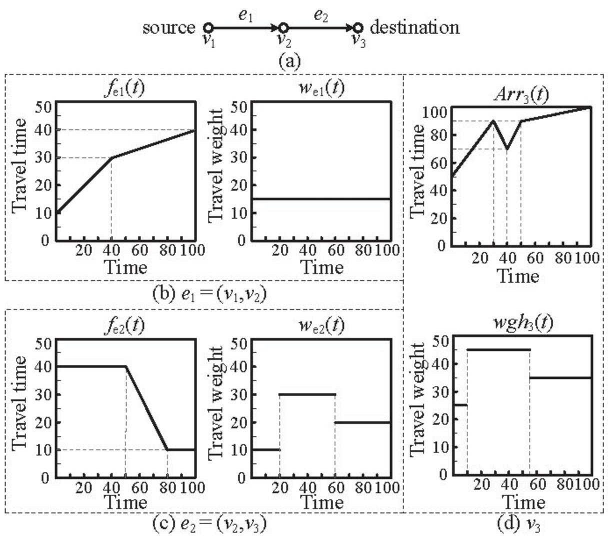


Figure 1: A continuous time-dependent graph.

a toll payment at most Δ. For an ICSP query, we consider the following scenario. Given a departure period $[t_{s1}, t_{s2}]$ from s and a budget constraint Δ, P would like to compute the earliest arrival time function (with domain $[t_{s1}, t_{s2}]$) at d, but takes a toll payment at most Δ. □

An interval query over time-dependent graphs usually computes the time function (also called speed profile [19]), which is very useful in practice. For example, the answer of an interval query gives the result of any point query by inputting the starting time point into the time function [24]. Therefore, an interval CSP also computes the earliest arrival time function in this paper.

There are some challenges in processing PCSP and ICSP queries efficiently, as explained below.

Challenges. The CSP query over a static graph is a classical NP-complete problem [13]. The extension from a static graph to a time-dependent graph increases the expressivity of the CSP query. Therefore, it may be more difficult to process the CSP query over time-dependent graphs than over static graphs. We prove that it is NP-complete to process the PCSP queries over continuous time-dependent graphs. However, the complexity becomes at least EXPSPACE in processing an ICSP query over continuous time-dependent graphs, and there is no polynomial time algorithm that can approximate the query with guarantee.

Our Approaches. To attack the hard problems, we propose novel algorithms. Denote StaCSP by an algorithm that solves the static CSP query. We first show that StaCSP can easily be extended to process the PCSP queries over continuous time-dependent graphs. To attack the harder problem (ICSP queries over continuous time-dependent graphs), we propose acceleration techniques by exploiting the structural properties of time-functions. Moreover, we develop parallel algorithms for the ICSP query that guarantee to scale with big time-dependent graphs.

Contributions. This paper aims to answer these questions.

(1) We conduct the study on CSP queries over large time-dependent graphs by incorporating continuous time and weight functions, and we formally define these problems in Section 2. We also study the problem complexities of the CSP queries in this section.

(2) We build connections between the static CSP query and the dynamic CSP queries, and adapt the algorithm (StaCSP) of the static CSP query to processing the dynamic PCSP query in Section 3. We propose a novel sequential algorithm (SICSP) to answer the dynamic ICSP query by exploiting the structural properties of time and weight functions in Section 3.

(4) We develop two parallel algorithms that guarantee queries to scale with graphs in Section 4.

(5) Using real road networks, we experimentally verify the effectiveness and scalability of SICSP (Section 5). We find the following. (a) SICSP is feasible on large graphs. It takes 17 seconds and 21MB memory on a graph of 5 million nodes and the process is accelerated by 22 times using 12 machines by our proposed method. (b) Our two parallel algorithms are parallel scalable: they are on average 4.3 and 4.6 times faster on large graphs, when the number of machines increases from 4 to 20.

Related Work. We categorize it as follows:

Shortest path over discrete time-dependent graphs. The simplest model of a time-dependent traffic network is the discrete time-dependent graph (or "timetable" graph). The timetable associated with each node consists of time-dependent events (e. g., a vehicle departing from a stop) that happen at discrete points in time.

A basic version of the model [27] contains a node for every departure and arrival event, with consecutive departure and arrival events linked by connection (or travel) edges. Several path planning algorithms (such as earliest arrival time path, latest departure time path, and shortest duration time path) have been proposed for such graphs. Cooke et al. [9] 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 several techniques, such as indexing have therefore been proposed to improve efficiency [33, 32, 34]. All these studies aim to optimize the time objective, and the algorithms there are almost the same. For example, our algorithms can directly solve the problems concerning the latest departure time path and the shortest duration time path with minor modifications. Some studies [7, 26] have aimed to optimize the earliest arrival time and number of transfers for time-table graphs. Their methods cannot solve our problem as the other optimized objective is different (ours is the weight-optimal objective).

With respect to the weight-optimal objective, several studies in the field of operation research consider the weight-optimal path problem in the context of the discrete time model [8]. They develop dynamic programming schemes to obtain the exact solution, but their time complexities are very high and cannot cope with a large graph. Close to our work is [35], 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 the minimum weight, not the minimum time.

Shortest path over continuous time-dependent graphs. The drawbacks of the discrete time model are two-fold. First, this model cannot represent the state of the graph between two discrete time points, which might yield inaccurate results. Second, the memory and processing requirements are high. A more precise way to describe a time-dependent traffic network is to use the continuous time-dependent function. For the point query, computing the earliest arrival time can also be done by making a minor modification to Dijkstra's algorithm if the first-in-first-out (FIFO) property holds at the continuous model [9, 23]. Concerning the interval query, recent work [19] have studied the shortest duration time path problem without the FIFO property as this approach allows to wait 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-based algorithm. On the other hand, the earliest arrival time path cannot be computed by this algorithm, because the whole graph does not have the FIFO property.

Other studies [24, 10, 12] have also provided Dijkstra-based algorithms to solve the problems with the continuous model, and the algorithm in [12] is the most efficient thereof because it applies the most precise refinement approach that expands the time interval step-by-step rather than computing the entire time interval iteratively. Other research has built different kinds of indices to accelerate the query, such as time-dependent CH [6] and time-dependent SHARC [11]. As far as we know, [30] is the only work to study the bi-criteria shortest path problem over continuous time-dependent graphs. The algorithm in [30] can give the exact answer to an ICSP query. However, the algorithm traverses all paths from source to destination, which may take exponential steps. Our proposed algorithm is efficient by only computing time functions of destination node rather than those of intermediate nodes from s to d. Therefore, our algorithm is nearly 200 times faster than the algorithm in [30] as shown in the experiments. [36] also studies the CSP query over time-dependent graphs. But it only gives exact solutions, and does not propose any approximate algorithms and parallel algorithms.

CSP query over static graphs. The CSP is a classical NP-complete problem. Handler and Zang [14] proposed a method for exact CSP processing: one method formulated CSP as an integer linear programming (ILP) problem, and solved it with a standard ILP solver. This same methodology was used by Mehlhorn and Ziegelmann [22]. The state-of-the-art solution for the exact CSP problem is that proposed in [15], which we call Sky-Dijk because it follows the general idea of Dijkstra's algorithm. To combat the hard problem, Hansen [15] proposed the first c-approximate solution, which runs in polynomial time, but has a high complexity. Lorenz and Raz. [20] reduced this complexity. However, this solution is much slower than an exact CSP algorithm, as shown in [18].

Our work differs from previous works in several ways. (a) The works on (discrete and continuous) time-dependent graphs advocate the Dijkstra-based algorithms, because all their algorithms utilize the following property: the earliest arrival time of a node v_i can be computed by the earliest arrival time of $v_i^{\prime}s$ incoming neighbors. However, this property no longer holds true for the problems proposed in this paper. (b) Previous works focus either on the time-optimal objective or the weight-optimal objective, whereas our problems concentrate on both time-optimal and weight-optimal objectives. One study on both optimal objectives is very inefficient, as it uses a very naive strategy. (c) All the works on the static CSP query do not consider the dynamic nature of time-dependent graphs. Therefore, we should propose novel algorithms to process CSP queries over continuous time-dependent graphs.

2. PROBLEM DEFINITION

In this section, we will present the definition of continuous graph time-dependent graphs, based on which we define PCSP and ICSP queries.

2.1 Continuous Graph Model

Time-Dependent Graph. A time-dependent graph is a simple directed graph, denoted as $G_t(V, E, F, W)$ (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 how many weights (e.g., toll fee) it takes to travel from u to v, if departing from u at time t. We define $|V| = n$ and $|E| = m$.

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 $f_e : [0, T) \to [1, T_e]$ such that $\lim_{t \to T} f_e(t) = f_e(0)$, for some fixed integer T_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 could be approximated by a set of PWL functions by applying the numerical approximation method [25]. Since f_e is a periodic, continuous PWL function, it can be represented succinctly by the number K_e of breakpoints defining f_e. Let $K = \sum_{e \in E} K_e$ denote the number of breakpoints to represent all the edge-time functions in G_t.

Figure 1 shows an example of a continuous time-dependent graph G_t with time function $f_e(t)$ and weight function $w_e(t)$ for each edge. In Figure 1(c), $f_{e_2}(t)$ defines the time functions of the edge $e_2 = (v_2, v_3)$. The period T and T_e of $f_{e_2}(t)$ are 100 and 40, respectively. $f_{e_2}(t)$ has two breakpoints $(50, 40)$ and $(80, 10)$.

FIFO Property. In this paper, we assume that the time functions have the *first-in-first-out* (FIFO) property. The FIFO property for an edge (u, v) implies that if departing earlier from u, one arrives earlier at v. We say G_t is a FIFO graph only if the time function $f_e(t)$ of every edge $e = (u, v)$ has the FIFO property, i.e., $t_1 + f_e(t_1) < t_2 + f_e(t_2)$ for $t_1 < t_2 \in [0, T)$. For example, consider a road network, for two cars towards the same road segment, the first one reaching the starting point should leave the end point first. From the PWL function perspective, $f_e(t)$ will satisfy the FIFO property only if each of its linear coefficients is $LC_i > -1$ for $i \in \{1, 2, ..., K_e\}$ and there are no discontinuities at which $f_e(t)$ drops to a lower value. For example, in Figure 1, all edge time functions have the FIFO property.

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 $Arr(v) = Dep(u) + f_e(Dep(u))$. As shown in the problem below, we aim to compute the earliest arrival time at a destination node d from a source s of G_t. Given a path

p from s to d, based on the FIFO property, the waiting at any node of p is never beneficial to a route algorithm for the problem. Thus, we let $Arr(v) = Dep(v)$ in this paper. 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 calculate

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

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 weight-function $w_e(t)$ is a piecewise constant function, calculated as follows:

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

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 and 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 constants. For example, in road networks, the toll fees for traveling through a road are distinct constant values during day and night. This means that the weight-function of this road is a piecewise constant function.

Figure 1 also illustrates the weight-functions for the two edges $e_1 = (v_1, v_2)$ and $e_2 = (v_2, v_3)$. Let $W = \sum_{e \in E} \sigma_e$ denote the number of breakpoints to represent all the edge-weight functions in G_t.

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:

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

The weight of path p is defined as $Wgh(p) = Wgh(v_{h+1})$.

2.2 Problem Statement

Let s and d be the route source and destination nodes in G_t, let t_s be a starting time point at s, and $[t_{s1}, t_{s2}]$ be a starting time interval. Let Δ be a user specified the weight constraint during the route from s to d. Next, we give the definition of the problem of PCSP and ICSP queries over time-dependent graphs.

Definition 1 (Point Constrained Shortest Path). (PCSP) *Given a continuous time-dependent graph $G_t = (V, E, F, W)$, a PCSP query $Q = (s, d, t_s, \Delta)$ 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)$Dep(s) = t_s$, and $Arr(d)$ is the minimum among all the possible paths meeting the conditions (1) and (2).*

Define $Arr_d(t)$ as the arrival-time function from s to d. Also define $Wgh_d(t)$ as the weight function from s to d. Specifically, $Arr_d(t)$ monitors the arrival time at d of a route R that departs from s at time t. $Wgh_d(t)$ monitors the total weight of R to d from s at time t. We then define an ICSP query over continuous time-dependent graphs.

Definition 2 (Interval Constrained Shortest Path). (ICSP) *Given a continuous time-dependent graph $G_t = (V, E, F, W)$, an ICSP query $Q = (s, d, t_{s1}, t_{s2}, \Delta)$ is to compute the earliest arrival-time function $Arr_d(t)$ from s to d, such that (1) $t \in [t_{s1}, t_{s2}]$ and (2) $Wgh_d(t) \le \Delta$.*

From the two definitions, we see that a ICSP query computes the minimum $Arr_d(t)$, whereas a PCSP query calculates the minimum $Arr_d(t_s)$ for $t = t_s$. Thus, the PCSP query is a special case of the ICSP query.

For example, we initiate an ICSP query against the time-dependent graph in Figure 1(a) with time and weight constraints: $s = v_1$, $d = v_3$, $[t_{s1}, t_{s2}] = [0, 30]$, $\Delta = 80$. The optimal $Arr_d(t)$ and $Wgh_d(t)$ are shown in Figure 1(d)[1].

Definition 3 (Path Retrieval). *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 the conditions (1), (2) and (3).*

2.3 Problem Complexity

In terms of the problem complexities associated with two queries, we propose the following theorems.

Theorem 1. *It is NP-complete to answer an ICSP query over continuous time-dependent graphs.*

The proof can be found in the full version of this paper [1].

Theorem 2. *The complexity lower bound of an ICSP query over continuous time-dependent graphs is EXPSPACE. In particular, it takes $2^\Omega(n)$ memory costs to answer the query and there is no polynomial time algorithm that can approximate it with any ratio bound. When every $f_e(t)$ is a constant function, the approximation ratio is constant.*

The proof can be found in the full version of this paper [1].

3. ALGORITHMS FOR CSP QUERIES

This section will propose algorithms for PCSP and ICSP queries over continuous time-dependent graphs.

Solution of PCSP Queries. Denote StaCSP by an algorithm that solves the static CSP query. We first give a good result for PCSP queries based on the principle of StaCSP.

Theorem 3. *StaCSP solves the PCSP query $Q_t = (s, d, t_s, \Delta)$ over a continuous time-dependent graph G_t.*

[1]For convenience, the two functions in this figure show a domain not restricted by $[0, 30]$.

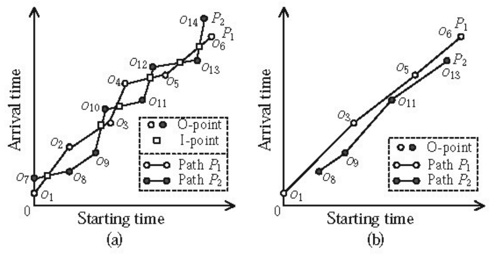


Figure 2: An illustration of O and I-points of two paths.

The proof can be found in the full version of this paper [1].

Solution of ICSP Queries. As shown in Theorem 2, it may take exponential time and space costs to process an IC-SP query over continuous time-dependent graphs. To speed up the ICSP query, we first exploit the structural properties of the functions $Arr_d(t)$ and $Wgh_d(t)$ in this section. We then propose efficient algorithms to compute the minimum $Arr_d(t)$ and $Wgh_d(t)$ based on the structural properties.

Note that once we obtain the minimum $Arr_d(t)$ and $Wgh_d(t)$, we can compute the valid earliest arrival time function by $Wgh_d(t) \leq \Delta$. Therefore, we only propose how to compute the minimum $Arr_d(t)$ and $Wgh_d(t)$ in the rest of this study.

3.1 Main Idea of Our Solution

3.1.1 Arrival-time and Weight Functions

Since both the time and weight functions are continuous, we construct the arrival-time and weight functions for node d of G_t instead of scalar values (as in previous works) for the discrete functions. Given node d, the arrival-time function and weight function of d are denoted by $Arr_d(t)$ and $Wgh_d(t)$, respectively. Recall that the query is $Q = (s, d, t_{s1}, t_{s2}, \Delta)$. $Arr_d(t)$ monitors the arrival time at d of a route R that departs from s at time t. $Wgh_d(t)$ monitors the total weight of R to d from s at time t. Thus, the two functions of d can be denoted by a pair $F_d(t) = (Arr_d(t), Wgh_d(t))$. The domain of $Arr_d(t)$ (resp. $Wgh_d(t)$) is the departure time from s within the interval $[t_{s1}, t_{s2}]$. For example, $Arr_d(20) = 30$ means that a route R starts from s at time 20 and arrives at d at time 30. $Weight_d(20) = 600$ means the total weight taken by R from s to d is 600.

3.1.2 Structural Properties of $Arr_d(t)$

The earliest arrival time function from s to d, $Arr_d(t)$, is a PWL function since all input arrival-time functions are assumed to be PWL functions and the function operators used to compute $Arr_d(t)$ do not change the linearity of the result. We are interested in the *breakpoints* on the curve $Arr_d(t)$ that connect its different linear pieces. We differentiate between two types of breakpoints. First, a breakpoint may represent the intersection between two pieces of arrival-time functions on different paths, referred to *I-point*. Second, a breakpoint may represent a breakpoint on one of the arrival-time functions for a path from s to d, referred to *O-point*. Figure 2(a) depicts an arrangement of the arrival-time functions for two paths and identifies the I-points and O-points. From this figure, we observe: (1) Once we obtain all O-points, we can establish $Arr_d(t)$ by connecting two neighboring O-points on the same path. (2) The I-points are the results (intersections) by these connections and need not be computed explicitly in order to establish $Arr_d(t)$. Based on this observation, we only show how to determine O-points as follows.

Every O-point corresponds to a breakpoint on the arrival-time function, $Arr_p(t)$, for some path p from s to d. Each breakpoint on the $Arr_p(t)$ function is the result of a breakpoint between two linear pieces of arrival-time functions on an edge of p introduced because of a compound operation for computing $Arr_p(t)$. In the following lemma, we demonstrate that every breakpoint of an edge arrival-time function can create at most one O-point on $Arr_p(t)$.

Lemma 1. *Suppose P is the set of all paths that go through edge $e = (u, v) \in E$ and $f_e(t)$ is the arrival-time function for e and has K_e breakpoints. Then, every path-arrival time function $Arr_p(t)$, $p \in P$, creates, a maximum total of K_e O-points on $Arr_d(t)$, i.e., each breakpoint t_i of $f_e(t)$ creates only one O-point on $Arr_p(t)$.*

Consider the following representation of the PWL function $f_e(t)$:

$$f_e(t) = \begin{cases} \alpha_1 t + \beta_1, & 0 \leq t < t_1 \\ \alpha_2 t + \beta_2, & t_1 \leq t < t_2 \\ \ldots \\ \alpha_{K_e} t + \beta_{K_e}, & t_{K_e - 1} \leq t < t_{K_e} = T \end{cases} \tag{2}$$

For every breakpoint $t_i, i = 1, ..., K_e$ of $f_e(t)$, consider path p_i to be the concatenation of a path with the latest starting time (LST) from s, which arrives at v at time t_i, link (u, v), and a path with an earliest arrival time (EAT) to d, which starts from v at time $\alpha^i t_i + \beta_i$. Additionally, recall that the starting time interval from s is $[t_{s1}, t_{s2}]$. A PCSP query with a starting time t_{s2} will return an arrival time of t_d at d. Based on the FIFO property, any departure within the interval $[t_{s1}, t_{s2}]$ will arrive at d before the time t_d.

Based on Lemma 1 and the FIFO property, t_i will create O-points only within the rectangular region with four corner points as: (t_{s1}, EAT), (LST, EAT), (t_{s1}, t_d) and (LST, t_d) (Figure 3(a)). To compute the related O-points of t_i, we first enumerate the path set $P_{s,u}$ from s to u and the path set $P_{v,d}$ from v to d. Thereafter, for every $p_{s,u} \in P_{s,u}$, we compute the departure time Dep by traversing $p_{s,u}$ from u at time t_i to s. Similarly, we compute the arrival time Arr by traversing $p_{v,d}$ from u at time $\alpha^i t_i + \beta_i$ to d. Finally, we obtain all the related O-points of t_i as (Dep, Arr). In order to make sure that each (Dep, Arr) is on the final $Arr_d(t)$, we also calculate the weight WGH of the route departing from s at time Dep to d at time Arr. (Dep, Arr) is an O-point of $Arr_d(t)$ if (Arr, WGH) is not dominated by any other pair (with a smaller arrival time and weight) with the same Dep.

For all O-points of $Arr_d(t)$, we classify the O-points on the same path into one group GP and sort them in the ascending order of their departure times. In every GP, we connect each pair of neighboring O-points by a linear piece, which is one part of $Arr_d(t)$. We then can obtain the complete $Arr_d(t)$. We naturally obtain all I-points, which are the intersection points of two different pieces.

3.1.3 Structural Properties of $Wgh_d(t)$

All breakpoints of $Wgh_d(t)$, referred to *W-points*, are only created from the breakpoints of $w_e(t)$ of every edge $e \in G_t$. Thus, the W-points of $Wgh_d(t)$ are similar to the O-points

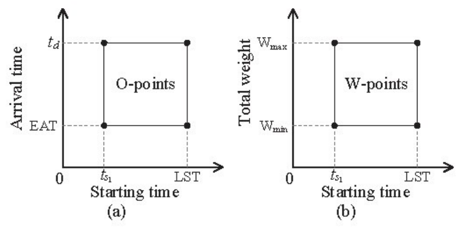


Figure 3: An illustration of the rectangle containing O-points and W-points.

of $Arr_d(t)$. $Wgh_d(t)$ does not have breakpoints that are similar to the I-points of $Arr_d(t)$, since all pieces of $Wgh_d(t)$ are horizontal and cannot intersect.

For every breakpoint $t_i, i = 1, ..., \sigma$ of $w_e(t)$ of an edge $e = (u, v) \in G_t$, t_i will create W-points within the rectangular region with four corner points as: (t_{s1}, W_{min}), (LST, W_{min}), (t_{s1}, W_{max}) and (LST, W_{max}). Here, LST is the latest starting time of the route from s which arrives at v at time t_i; W_{min} is the smallest weight of the route starting from s within the time interval $[t_{s1}, LST]$ to d with a time no later than t_d, passing through edge (u, v); and W_{max} is the largest weight of the route starting from s within the time interval $[t_{s1}, LST]$ to d with a time no later than t_d, passing through edge (u, v). Figure 3(b) shows such a rectangle. To compute the related W-points of t_i, we first enumerate the path set $P_{s,u}$ from s to u and the path set $P_{v,d}$ from v to d. Thereafter, for every $p_{s,u} \in P_{s,u}$, we compute the weight WGH_1 of a route by traversing $p_{s,u}$ from u at time t_i to s. At the same time, we also compute the weight WGH_2 of a route by traversing $p_{v,d}$ from v at time $\alpha^i t_i + \beta_i$ to d. Thus, we obtain the total weight $WGH_t = WGH_1 + WGH_2 + w_e(t_i)$ of the route R from s to d passing through $e = (u, v)$. Surely we obtain the departure time Dep and arrival time Arr of R. Finally, we can obtain all the related W-points of t_i as (Dep, WGH_t). (Dep, WGH_t) could be a final point of $Wgh_d(t)$ if its (Arr, WGH_t) is a skyline point among all the points (i.e., not dominated) with the same departure time Dep. We can obtain complete pieces of $Wgh_d(t)$ by connecting the W-points with the same WGH_t.

3.1.4 Acceleration Techniques

Theorem 2 tells us that it takes at least 2^n memory to compute the minimum $Arr_d(t)$ and $Wgh_d(t)$, and there is no polynomial-time approximation algorithm with a guarantee. This negative result forces us to resort to heuristic strategies. We minimize both $Arr_d(t)$ and $Wgh_d(t)$ at the same time, which is too expensive. Our heuristic strategy sets its first priority as minimizing $Arr_d(t)$ and then to minimize $Wgh_d(t)$. Intuitively, the heuristic strategy first computes fast routes (from s to d), among which the route with the least weight is selected.

To establish $Arr_d(t)$, the heuristic scheme first computes O-points with Arr as small as possible. Among such points, the heuristic scheme selects the skyline points as the final O-points. Thereafter, $Arr_d(t)$ is constructed according to the scheme proposed in the previous subsection. Similarly, to construct $Wgh_d(t)$, we compute the W-points with small values for the earliest arrival time, and we then select the skyline W-points.

For a t_i of $f_e(t)$, the heuristic scheme only selects the point (LST, EAT) (the lower-right corner of the rectangle in Figure 3(a)) as its candidate O-point. There are two reasons for this approach. First, (LST, EAT) can achieve the smallest arrival time for t_i. Second, for any other point (Dep, EAT) of the rectangle with $Dep < LST$, the departure time Dep might have a related smaller arrival time than LST, though (Dep, EAT) also achieves the smallest arrival time EAT. Similarly, we construct $Wgh_d(t)$ by computing (LST, EAT). In particular, the heuristic strategy works as follows:

Heuristic Strategy. All the possible O-points on $Arr_d(t)$ could be captured by computing, for every breakpoint at time t_i on the edge-time function $f_e(t)$ of each edge $e = (u, v)$, the latest stating time (LST) at s for arriving at u at time t_i, and the earliest arrival time (EAT) at d for departure time $f_e(t_i)$ at v. The point (LST, EAT) is a potential O-point on $Arr_d(t)$. In order to make sure that (LST, EAT) is on the final $Arr_d(t)$, we also calculate the weight WGH of the route departing from s at time LST to d at time EAT and obtain a point of $Arr_d(t)$ as (LST, WGH). If (EAT, WGH) is not dominated by any other pair with the same LST, (LST, EAT) is an O-point of $Arr_d(t)$. Similarly, all the W-points of $Wgh_d(t)$ can be computed, for every breakpoint at time t_i on the edge-weight function $w_e(t)$ of each edge $e = (u, v)$, as (LST, WGH). (LST, WGH) could be a final point of $Wgh_d(t)$ if (Arr, WGH) is a skyline point among all the points with the same LST.

Although we adopt a heuristic strategy, the query quality is very high, as shown in the experiments. Furthermore, the heuristic strategy results in a very fast query response time.

3.2 Algorithm Details

Based on the heuristic strategy, we propose two efficient procedures that compute the minimum $Arr_d(t)$ and $Wgh_d(t)$. The two procedures are denoted as ICSP_Arr and ICSP_Wgh. In this section, we show how to perform ICSP_Arr whose pseudo-codes are illustrated in Algorithm 1.

ICSP_Arr uses a list L to maintain the O-points of $Arr_d(t)$. In particular, ICSP_Arr consists of three phases: (1) calculate the O-points of $Arr_d(t)$, (2) validate them to obtain the final O-points, and (3) construct $Arr_d(t)$.

Phase 1: Calculate the O-points of $Arr_d(t)$ (lines 3-8). To compute the candidate O-points of $Arr_d(t)$, for each breakpoint of each edge $e = (u, v)$, ICSP_Arr first determines the latest departure time LST by the Dijkstra-based algorithm from u on each time t_i of $f_e(t)$ to s, and the earliest arrival time EAT by the Dijkstra-based algorithm from v at time $f_e(t_i)$ to d (lines 5-6). Let the two shortest paths be SP_1 and SP_2. ICSP_Arr then computes the weight of the path from s to d by summarizing the weights of SP_1, SP_2 and e (line 7). Finally, ICSP_Arr adds the information of every computed point to L as candidate O-points (line 8).

Phase 2: Validate the Candidate O-points (lines 9-18). To validate every candidate O-point T_i of L, ICSP_Arr verifies whether T_i is dominated by a point (on some path P from s to d) with the same departure time as T_i. Note that P must contain several candidate O-points in L. To achieve this aim, ICSP_Arr first classifies the points of L on the same path into one group and obtains a partition of L (line 9). ICSP_Arr easily obtains the classification, as the paths have been determined in computing the candidate O-points. After the classification, ICSP_Arr validates every candidate O-point T_i of L by comparing the informa-

Algorithm 1: ICSP_Arr

```
Input: $G_t(V,E)$, $Q(v_s, v_d, t_{s1}, t_{s2}, \Delta)$
Output: $\{Arr_d(t)\}$ of $d$
1  $Arr_d(t) = NULL$;
2  List $L$;
3  for every edge $e = (u,v) \in E$ do
4      for $i = 0$ to $|f(e)|$ do
5          $LST = ReverseDijkstra(u, s, t_i)$;
6          $EAT = Dijkstra(v, d, f_e(t_i))$;
7          Determine the weight $WGH$ of the corresponding path
           from $s$ to $d$ through $(u,v)$;
8          $InsertToList(L, T = \{LST, EAT, WGH\})$;
9  Classify the points of $L$ on the same path into one group and
   obtain a partition of $L$;
10 for each $T_i = \{LST_i, EAT_i, WGH_i\}$ in $L$ do
11     Sort points $T_j$ of each group of $L$ in the ascending order of
       $|LST_i - LST_j|$;
12     for each group $GP$ in $L$ except for the group including $T_i$
       do
13         for each $T_j = \{LST_j, EAT_j, WGH_j\}$ of $GP$ as the
           order do
14             Determine the path $P$ related to $GP$;
15             Determine the weight $WT$ of $P$ from $s$ at time
               $LST_i$ to $d$ at time $Arr_p(LST_i)$;
16             if $Arr_p(LST_i) < EAT_i$ and $WT < WGH_i$ then
17                 Remove O-point $T_i = \{LST_i, EAT_i, WGH_i\}$
                   from $L$;
18             Break;
19 for each group $GP$ of $L$ do
20     Sort points $T_i = (LST_i, EAT_i)$ in the ascending order of
       $LST_i$;
21     for each two neighbor points $T_i$ and $T_j$ in the sorted $GP$
       do
22         AddLinearPiece($Arr_d(t), T_i, T_j$);
```

tion of T_i with that of the points in other groups (paths, line 10). Before the comparison, ICSP_Arr sorts each of the other groups GPs in the ascending order of $|LST_i - LST_j|$ for T_i and T_j in GP (line 11), so that the comparison starts from the points close to T_i. It then compares T_i with the points in each sorted GP in ascending order (lines 12-13). In this process, ICSP_Arr obtains the path P from s at time LST_j to d at time EAT_j (line 14). Thereafter, it obtains the weight WT of this route and the compared point $(Arr_p(LST_i), WT)$ (line 15). ICSP_Arr removes T_i if it is dominated by $(Arr_p(LST_i), WT)$ (lines 16-17). Note that once a T_j is removed, ICSP_Arr jumps out of traversing GP and starts for another group of L (line 18), because only one point in each path can be dominated. Finally, ICSP_Arr obtains true O-points from L.

Phase 3: Construct $Arr_d(t)$ (lines 19-22). This phase is easy, since ICSP_Arr has obtained all breakpoints to build $Arr_d(t)$. First, for the breakpoints in each group (path), ICSP_Arr sorts them in the ascending order of LST_i (line 21) such that ICSP_Arr can add a linear piece of $Arr_d(t)$ between two consecutive breakpoints (lines 21-22).

Example 2. *Figure 2(b) illustrates an example of how ICSP_Arr is performed. Recall that Figure 2(a) gives all O-points and I-points of two paths. ICSP_Arr calculates all (LST, EAT) as candidate O-points, which have smaller arrival times than those in Figure 2(a). After validating all (LST, EAT), ICSP_Arr obtains the true O-points of two paths as shown in Figure 2(b), i.e., O_1, O_3, O_5, O_6 of path P_1 and O_8, O_9, O_{11}, O_{13} of path P_2. ICSP_Arr connects the pair of neighboring O-points on the same path and then outputs the approximation function $Arr_d(t)$, i.e., the path formed by O_1, O_3, O_5, O_6 approximates P_1 and the path formed by O_8, O_9, O_{11}, O_{13} approximates P_2. From this example, we observe that after the heuristic method is applied, the approximation function $Arr_d(t)$ has 12 fewer breakpoints (O-points and I-points) in Figure 2(b) than those in Figure 2(a). This result shows that the heuristic method is effective.* □

Algorithm 2: ArrTime

```
Input: $(Arr_d(t), Wgh_d(t))$ of $L(d)$, $Q$
Output: $Dep(s)$, $Arr(d)$
1  Construct a time list $TL$ and a weight list $WL$;
2  for each pair $(Arr_d(t), Wgh_d(t))$ in $L(d)$ do
3      Based on $Wgh_d(t)$, determine constant weights with
       values smaller than $\Delta$ and insert them into $WL$;
4      Based on $WL$, determine the smallest and largest
       time points $[t_a, t_b]$;
5      if $[t_{s1}, t_{s2}] \cap [t_a, t_b] \neq \phi$ then
6          Insert Min($Arr(t_{s1}), Arr(t_a)$) into $TL$;
7  Determine the smallest time point $t_\tau$ in $TL$;
8  $Dep(s) = Arr_d^{-1}(t_\tau)$;
9  $Arr(d) = t_\tau$
10 return $(Dep(s), Arr(d))$;
```

ICSP_Wgh uses similar steps to ICSP_Arr to calculate $Wgh_d(t)$. Thus, we omit the detailed description of ICSP_Wgh.

Theorem 4.

- *The time complexity of ICSP_Arr is $O(K(m + n\log n) + K^2(\log K + m + n))$, where K is the total number of breakpoints of $f_e(t)$ in G_t. The memory complexity of ICSP_Arr is $O(m)$.*
- *The time complexity of ICSP_Wgh is $O(W^2(n+m) + W(m+n\log n) + W\log W)$, where W is the total number of breakpoints of $w_e(t)$ in G_t. The memory complexity of ICSP_wgh is $O(m)$.*

The proof can be found in the full version of this paper [1].

Path Retrieval. Now that we obtain the smallest functions $Arr_d(t)$ and $Wgh_d(t)$, we can compute the earliest arrival time $Arr(d)$ and its related departure time $Dep(s)$, by inputting $Q = (s, d, t_{s1}, t_{s2}, \Delta)$ into $Arr_d(t)$ and $Wgh_d(t)$. Specifically, Algorithm 2 shows how to compute $Arr(d)$ and $Dep(s)$. Note that the input $L(d)$ is the list of pair of functions $(Arr_d(t), Wgh_d(t))$. Based on the fixed departure time $t_s = Dep(s)$, we can use the PCSP query $Q = (s, d, t_s, \Delta)$ to retrieve the path for the ICSP query.

4. PARALLEL ICSP QUERY

ICSP queries may be cost-prohibitive over big time dependent graphs G_t. Therefore we develop parallel algorithms for the ICSP queries that guarantee to scale with big G_t.

4.1 Parallel Scalability

To characterize the effectiveness of parallelism, we advocate a notion of parallel scalability following [17]. Consider a problem I posed on a graph G_t. We denote by $t(|I|, |G|)$ the running time of the best sequential algorithm for solving I on G_t. For a parallel algorithm, we denote by $T(|I|, |G|, n_s)$

the time it takes to solve I on G by using n_s machines, taking n_s as a parameter.

Parallel Scalability. An algorithm is parallel scalable if,

$$T(|I|, |G_t|, n_s) = O(\frac{t(|I|, |G_t|)}{n_s}) + (n_s|I|)^{O(1)}$$

. That is, the parallel algorithm achieves a linear reduction in sequential running time, plus a "bookkeeping" cost $O((n_s|I|)^l)$ that is independent of $|G_t|$, for a constant l.

A parallel scalable algorithm guarantees that the more machines that are used, the less time it takes to solve I on G_t. Hence, given a big graph G_t, it is feasible to efficiently process I over G_t by adding machines when needed.

4.2 Parallel Algorithms

To parallelize the ICSP query, we should parallelize the sequel algorithm ICSP_Arr (Algorithm 1). In this section, we propose two parallel algorithms (denoted as T_Arr and F_Arr), each of which works with a master M_c and n_s slaves (machines).

The proposed schemes consist of two parallelisms, *time-parallelism* and *fragment-parallelism*. T_Arr utilizes the following time-parallelism: T_Arr creates a partition scheme of the time period T over multiple slaves once for all, so that it is performed on all partitioned time sub-intervals in parallel. F_Arr utilizes the following fragment-parallelism: F_Arr creates a partition scheme of G_t over multiple slaves once for all, so that it is performed on these fragments in parallel.

We first show how T_Arr is executed, and we then introduce F_Arr.

4.2.1 Time-based Parallel Algorithm T_Arr

We first distribute G_t to n_s slaves in two steps: (1) Each slave maintains a copy of G_t. The copy only contains the node and edge sets of G_t instead of its edge-time and edge-weight functions. (2) T_Arr partitions the time period T of functions into n_s disjointed subintervals, and the edge-time and edge-weight functions of the ith subinterval are distributed to G_t in the ith slave. Note that the length of each subinterval should be equal so that the distribution is balanced.

Based on the partition, T_Arr works as follows: (1) The master M_c posts Q to each slave. (2) Each slave M_i then invokes ICSP_Arr to compute the partial function of the minimum $Arr_d(t)$ and sends the answer to M_c. (3) Once all the slaves have sent their partial functions to M_c, the master computes $Arr_d(t)$ of G_t as the union of all the partial functions.

Note that the slave M_i contains all the nodes and edges of G_t, and every edge e holds its ith partial time-function $f_e^i(t)$. The main idea of ICSP_Arr is to compute the shortest-paths for every breakpoint of $f_e^i(t)$. Therefore, ICSP_Arr outputs O-points originated from the breakpoints of $f_e^i(t)$. In other words, the partial function computed in M_i is originated from $f_e^i(t)$. After unifying all the partial functions, we could obtain the complete $Arr_d(t)$.

For T_Arr, we have the following theorem.

Theorem 5. *T_Arr is parallel scalable for graph G_t taking time $O(\frac{t(Q,G_t)}{n_s} + n_s)$, where $t(Q, G_t)$ is the running time of ICSP_Arr.*

The proof can be found in the full version of this paper [1].

4.2.2 Fragment-based Parallel Algorithm F_Arr

We first partition G_t into n_s fragments and distribute them to the slaves. To maximize parallelism, a partition scheme should guarantee that, (1) each of n_s slaves manages a small fragment of approximately equal size, and (2) a query can be evaluated locally at each fragment without incurring inter-fragment communication. We propose such a scheme.

In (1), a fragment does not only include a substructure of G_t, but also includes the edge-functions (time and weight) associated with the substructure. Thus, the balance should consider both substructures and their edge-functions.

To achieve this goal, we construct a static graph $G_s(V_s, E_s)$ from $G_t(V_t, E_t)$ as follows: $V_s = V_t$ and $E_s = E_t$. Each edge $e_s \in E_s$ has a weight of $A(e_s) = k(e_t) + w(e_t)$, where $e_s = e_t$ for $e_t \in E_t$. Here, $k(e_t)$ and $w(e_t)$ are the numbers of the breakpoints of functions $f_e(t)$ and $w_e(t)$, respectively. Then, we partition G_t into balanced fragments as follows.

Balanced Fragments. Each slave M_i manages a fragment F_i, which contains the subgraph G_i of $G_t(V_t, E_t)$ induced by a set V_i of nodes, such that $\bigcup V_i = V_t$ ($i \in [1, n_s]$) and the size of F_i is bounded by $c \cdot \frac{\sum_{e_s \in E_s} A(e_s)}{n_s}$, for a small constant c.

Intuitively, each balanced fragment F_i of G_t includes almost the same sized subgraph as well as the same number of breakpoints. Based on this definition, we can use an existing balanced graph partition strategy (e.g., [31]) to perform on G_s and obtain the static F_i. Then, F_i is associated with edge functions. Specifically, [31] uses multilevel label propagation to iteratively coarsen a graph until the coarsened graph is small enough, and then uses a high quality off-the-shelf partitioning algorithm to generate the final partitioning on the coarsened graph.

To achieve (2), F_Arr should be parallel scalable. Unfortunately, we have a negative theorem for this goal.

Theorem 6. *Any exact sequential algorithm for the ICSP query over continuous time-dependent graphs cannot have a fragment-based parallel scalable version.*

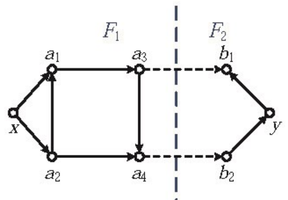
The proof can be found in the full version of this paper [1].

We cannot have a parallel scalable algorithm from an exact algorithm. We however develop a parallel scalable algorithm by resorting to heuristic techniques. Specifically, we add *more heuristics* to ICSP_Arr to make it parallel scalable. We first show which steps of ICSP_Arr should be parallelized.

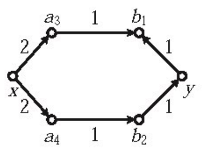
In ICSP_Arr, steps 5 and 6 take $O(m + n \log n)$ time by running the Dijkstra-based algorithms, and other steps take $O(1)$ time when K are considered as constants. Thus, F_Arr parallelizes the most expensive steps 5 and 6 of ICSP_Arr. Based on the proof of Theorem 6, we know that there is no parallel scalable algorithm for steps 5 or 6 of ICSP_Arr.

We solve the problem by adding heuristics to steps 5 and 6. We first define some key concepts before giving the parallel scalable algorithm (F_Arr).

For an edge $e \in G_t$, we define $U(e)$ as the upper bound of $f_e(t)$ and $L(e)$ as the lower bound of $f_e(t)$. For a time-dependent graph $G_t = (V_t, E_t, f_e(t))$, we also define the 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 of 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 of the time-function of $e_t \in G_t$.



(a) A partitioned G_u



(b) A graph S_u

Figure 4: A partitioned graph G_u and its summary graph S_u.

We partition G_u (resp. G_l) to obtain balanced fragments over slaves. For two fragments F_i and F_j ($i \neq j$), we refer a node $u \in F_i$ to a *border node* of F_i if u has a neighbor v in F_j. The edge between two border nodes is a *crossing edge*. F_Arr is performed over the partitioned G_u and G_l.

Figure 4 gives a graph G_u partitioned into two fragments F_1 and F_2. Node a_3 is a border node of F_1, since a_3 has a neighbor b_1 in F_2. Edge (a_3, b_1) is a crossing edge from F_1 and F_2.

```
Algorithm 3: F_Arr
   Input: $G_t$, $Q$, master $M_c$, $n_s$ slaves $M_1$,...,$M_{n_s}$
   Output: $\{Arr_d(t)\}$ of $d$
1  /*executed at master $M_c$*/
2  Perform steps 1-4 of ICSP_Arr;
3  $\widehat{LST}$ =Shortest_Path($u, s, t_i$);
4  $\widehat{EAT}$ =Shortest_Path($v, d, f_e(t_i)$);
5  Perform steps 7-23 of ICSP_Arr;
```

Now, we show the detailed steps of F_Arr in Algorithm 2 which works as follows. At master M_c, F_Arr performs all the steps of ICSP_Arr except for steps 5 and 6, which are parallelized over the slaves. Either step 5 or 6 is parallelized through the procedure Shortest_Path that approximates the true values of LST and EAT (lines 3 and 4). Based on the approximations ($\widehat{LST}$ and $\widehat{EAT}$), we perform ICSP_Arr to approximate $Arr_d(t)$. In the following, we first show how Shortest_Path is executed, after which we analyze the complete process of F_Arr.

Procedure Shortest_Path. The main idea of Shortest_Path is as follows. Shortest_Path parallelizes the Dijkstra-based algorithm over the static graphs G_u and G_l instead of the time-dependent graph G_t. Intuitively, G_u and G_l maintain the upper and lower bounds of the travel time over G_t, and thus Shortest_Path returns the upper and lower bounds of the true value returned by step 5 (or step 6) in ICSP_Arr. Shortest_Path then uses the bounds to approximate the true value. We prove that Shortest_Path is parallel scalable.

Shortest_Path inputs a query Q_{sp} which consists of two nodes (x, y) of G_t and a starting time t_x from x. Shortest_Path also inputs static graphs G_u and G_l. Shortest_Path outputs arrival time Arr_y at y. Shortest_Path is executed over the master M_c and n_s slaves M_i ($i \in [1, n_s]$). Algorithm 3 shows the detailed steps of Shortest_Path.

Over M_c, Shortest_Path works as follows. (1) M_c posts Q_{sp} to each slave M_i (line 2). (2) After M_c receives answers from every slave (line 3), M_c constructs two summary graphs

```
Algorithm 4: Shortest_Path
    Input: $Q_{sp} = (x, y, t_x)$, partitioned $G_u$ and $G_l$
    Output: $Arr_y$
1   /*executed at master $M_c$*/
2   Post $Q_{sp}$ to each slave $M_i$;
3   if every slave $M_i$ returns the answer then
4       Construct two summary graphs $SG_u$ and $SG_l$ based
        on all the answers;
5       Over $SG_u$ (resp. $SG_l$), compute the
        shortest-distance $SD_u$ (resp. $SD_l$) from $x$ to $y$;
6       $UP = t_x + SD_u$, $LB = t_x + SD_l$;
7       return $\frac{UP+LB}{2}$;
8   /*executed at each slave in parallel over $G_u$ (resp.
    $G_l$)*/
9   if $M_i$ contains $x$ then
10      Compute the shortest-distance $SD_b$ from $x$ to each
        border node $b$ of $M_i$;
11      Send the set $\{(b, SD_b)\}$ to $M_c$;
12  else if $M_i$ contains $y$ then
13      Compute the shortest-distance $SD_b$ from $y$ to each
        border node $b$ of $M_i$;
14      Send the set $\{(b, SD_b)\}$ to $M_c$;
15  else
16      Compute the shortest-distance $SD_{a,b}$ between each
        pair of border nodes $a$ and $b$ of $M_i$;
17      Send the set $\{(a, b, SD_{a,b})\}$ to $M_c$;
```

$S_u = (V_1, E_1, W_1)$ and $S_l = (V_2, E_2, W_2)$ based on all the answers (line 4). Note that x and y are nodes of S_u (resp. S_l). (3) Over S_u (resp. S_l), M_c computes the shortest-distance SD_u (resp. SD_l) from x to y (line 5). (4) Finally, M_c calculates upper and lower bounds of Arr_y as $UP = t_x + SD_u$, $LB = t_x + SD_l$ and outputs the average $\frac{UP+LB}{2}$ to approximate Arr_y (lines 6 and 7).

Therefore, we know that graphs $S_u = (V_l, E_1, W_1)$ and $S_l = (V_2, E_2, W_2)$ are important to Shortest_Path. Below, we formally define S_u. S_l can be defined similarly as S_u.

$S_u = (V_1, E_1, W_1)$ is defined from the partitioned G_u as follows: (a) The node set V_1 consists of x, y and the border nodes of G_u. (b) The edge set E_1 consists of four types of edge sets E_x, E_y, E_a and E_b, i.e., $E_1 = E_x \cup E_y \cup E_a \cup E_b$. (b1) For every edge $e_x = (x, u_x) \in E_x$, x and u_x are in the same fragment F_i of G_u, and u_x is a border node of F_i. Its weight $W_1(e_x)$ is the shortest-distance from x to u_x within F_i. (b2) E_y and its weight function are defined similarly as E_x. (b3) For every edge $e_a = (u_a, v_a) \in E_a$, u_a and v_a are border nodes of the same fragment F_j and neither x nor y are in F_j. Its weight $W_1(e_a)$ is the shortest-distance from u_a to v_a within F_i. (b4) For every edge $e_b \in E_b$, e_b is a crossing edge $e_c \in G_u$. Its weight $W_1(e_b)$ is $F_u(e_c)$.

Simply speaking, the summary graph is constructed from the partitioned graph, only maintaining the source node, the destination node and the border nodes of each fragment.

For example, Figure 4(b) shows the summary graph S_u of the partitioned G_u in Figure 4(a). S_u is constructed from G_u as follows: x has edges to border nodes a_3 and a_4 of F_1 with edge weights 2 and 2; S_u keeps the crossing edges (a_3, b_1) and (a_4, b_2) of G_u; y also connects to the border nodes b_1 and b_2 of F_2. Note that S_u does not contain nodes a_1 and a_2, because they are not the mentioned nodes above.

Subsequently, over n_s slaves, Shortest_Path computes the node set V_1, the edge set E_1 and the edge-weight set W_1 of S_u (resp. S_l) in parallel, which works as follows. (1) When slave M_i contains x (line 9), we perform line 10 and obtain E_x (line 11). (2) When slave M_i contains y (line 12), we perform line 13 and obtain E_y (line 14). (3) When slave M_i contains neither x nor y (line 15), we perform line 16 and obtain E_a (line 17). After the three phases, we also obtain E_b and can construct S_u (resp. S_l) in the master.

Theorem 7. *F_Arr is parallel scalable (with running time $O(\frac{t(Q,G_t)}{n_s} + n_s)$), where $t(Q, G_t)$ is the running time of ICSP_Arr.*

The proof can be found in the full version of this paper [1].

5. PERFORMANCE EVALUATION

Specifically, we evaluate the scalability and parallel scalability of ICSP queries over continuous time-dependent graphs. For *scalability*, we use one machine that has 2 Intel Xeon E5345 CPUs, 32GB memory, and runs CentOS Linux 5.6. For *parallel scalability*, we use a cluster of 21 machines in a high-speed kilomega network, where one machine is selected as the master and the remaining 20 machines are selected as slaves. Each slave has the same configuration as the one in the scalability. All programs are coded in Java.

5.1 Experimental Settings

Datasets. We employ the following real time-dependent road networks.

CDU: We use a real taxi trajectory dataset collected by Didi Chuxing [3] in Chengdu, China, which is published through its GAIA initiative [4]. Each taxi trajectory of this dataset is represented by a sequence of time-stamped points, each of which contains the information of latitude, longitude and altitude. The taxi trajectories were recorded by different taxi GPS loggers, and have a variety of sampling rates, i.e., every 2-4 seconds per point. The dataset is collected in a period of over two months (from October 2016 to Novermber 2016).

To map the trajectories to road network, we use the latest city boundaries [2] and extract its road network out of the national road network of China from Geofabrik via Osmconvert [5]. The road network is represented as an undirected graph with 214,440 nodes and 466,330 edges. For each edge e of the road network, we obtain its time-function $f_e(t)$ from the real time-stamped points associated with it.

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

We generate time-dependent graphs using W-US dataset 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. To simulate a real traffic case, we compute the betweenness centrality for every edge in G_t and sort all the edges in descending order of betweenness. The time domain is set as $T = [0, 2, 000]$, i.e., the departure time t can be selected from [0, 2,000] for any node in a graph. Here, 2,000 means 2,000 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 and then it is 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 maximum value of $f_e(t)$. Within each subinterval $[t_{x-1}, t_x]$ $(1 \le x \le k)$, $f_e(t)$ 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 $\triangle t_x = t_x - t_{x-1}$. Then, the time function $f_e(t)$ is guaranteed to be non-negative and FIFO.

Table 1: Approximation ratios of SICSP

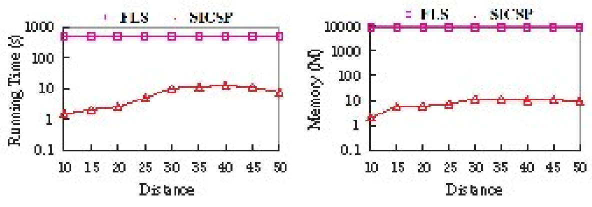
Distance (d_s)	10	15	20	25	30
Approximation ratio	1.2	1.5	2.1	2.8	3.5
Time interval (l_s)	400	600	800	1,000	1,200
Approximation ratio	1.3	1.7	2.1	2.8	3.2
No. of segments of $f_e(t)$ (n_f)	5	10	15	20	25
Approximation ratio	1.5	1.8	2.1	2.8	3.3
No. of segments of $w_e(t)$ (n_w)	5	10	15	20	25
Approximation ratio	1.6	1.8	2.1	2.5	2.5

Algorithms. We evaluate the proposed algorithms for ICSP queries over time-dependent graphs. Specifically, we evaluate the sequential algorithm (SICSP) and compare it with the forward label setting (FLS) algorithm in [30] We evaluate the time-parallelism based algorithm (PICSP-T) and the fragment-parallelism based algorithm (PICSP-F).

Metrics. We are interested in the following aspects for evaluating the performances of SICSP, PICSP-T and PICSP-F: (1) the impact of the number of nodes ($|V_t|$); (2) the impact of the number of edges ($|E_t|$); (3) the impact of distances (d_s) between the 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 $w_e(t)$. For PICSP-T and PICSP-F, we also study the impact of the number of slaves (n_s). The parameters requiring evaluation are: (1) querying time; (2) memory overhead; and (3) the approximation ratio of the heuristic method (i.e., SICSP, PICSP-T and PICSP-F).

The approximation ratio is computed as follows. FLS returns the exact function $Arr_d(t)$, and a heuristic method returns an approximation function $Arr_d^*(t)$. We randomly select w time points t_i $(1 \le i \le w)$ from the starting time interval $[t_{s1}, t_{s2}]$ and we then obtain w function values for $Arr_d(t)$ (resp. $Arr_d^*(t)$): $Arr_d(t_i)$ for $1 \le i \le w$ (resp. $Arr_d^*(t_i)$ for $1 \le i \le w$). Based on the function values, we compute the approximation ratio as $(\sum_{i=1}^{w} \frac{Arr_d^*(t_i)}{Arr_d(t_i)})/w$. We set $w = 40$ in the following experiments.

Experimental Results.
Exp-1: Approximation Ratio of SICSP. We first evaluate the approximation ratios of SICSP on the CDU graph by varying d_s from 10 to 30, l_s from 400 to 1,200, and n_f and n_w from 5 to 25. Table 1 reports the results from which we find the following. (1) The approximation ratios increase when d_s and n_f increase, and the largest approximation ratios are 3.5 and 3.3. (2) In contrast, the approximation ratios do not increase strictly as l_s and n_w grow, and the



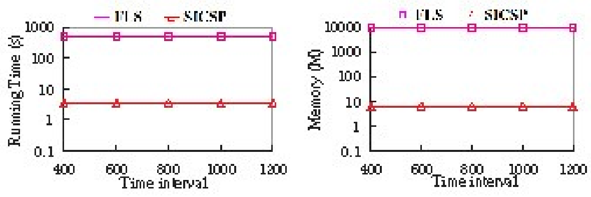
(a) Query time (b) Memory overhead

Figure 5: Impact of distance between source and destination.

approximation ratios are both around 3. (3) The findings in (1) and (2) show that SICSP returns very small (constant) approximation ratios in practice for the real traffics, despite that the algorithm is heuristic.

Exp-2: Scalability of **SICSP**. This set of experiments evaluate the scalability of SICSP, compared to FLS.

Varying d_s. We vary d_s from 10 to 30 on the CDU graph. In this test, we set $l_s = 800$, $n_f = 15$ and $n_w = 15$. To generate the distance of l , we fix a source and perform a BFS search in l hops to obtain a set of destinations. As shown in Figure 5, (1) SICSP is very efficient (e.g., 3.3s on average) and consumes little memory overhead (e.g., 6MB on average). In contrast, FLS takes 500s, and even worse yet, consumes nearly 10GB of memory overhead. (2) SICSP consumes more time and memory overhead as d_s increases, since a longer distance needs more computations in SICSP. FLS is not affected by changes in d_s because FLS computes the arrival and weight functions from s to all other nodes.



(a) Query time (b) Memory overhead

Figure 6: Impact of the length of time interval $[t_{s1}, t_{s2}]$.

Varying l_s. We vary l_s from 400 to 1,200 on the CDU graph. In this test, we set $d_s = 20$, $n_f = 15$ and $n_w = 15$. As shown in Figure 6, (1) SICSP is 200 times faster than FLS and consumes 1/1,000 the memory overhead of FLS. (2) Both the querying time and memory overhead of SICSP and FLS are not affected, since both $f_e(t)$ and $w_e(t)$ do not change.

Varying n_f and n_w. We vary both n_f and n_w from 5 to 25 on the CDU graph. In this test, we set $d_s = 20$ and $l_s = 800$. As shown in Figure 7, (1) the querying time and memory overhead of SICSP increase along with the number of segments (i.e., n_f and n_w). This is because SICSP runs more Dijkstra-based algorithms when n_f and n_w increase. (2) ICSP-T increases more rapidly than ICSP-W, since the change in n_f has a bigger impact on $Arr_d(t)$ than the change in n_w.

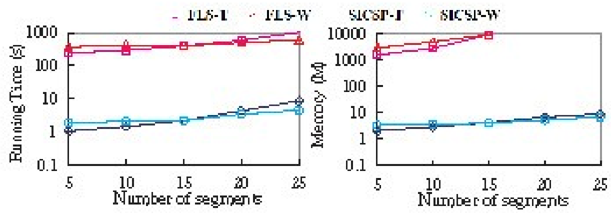
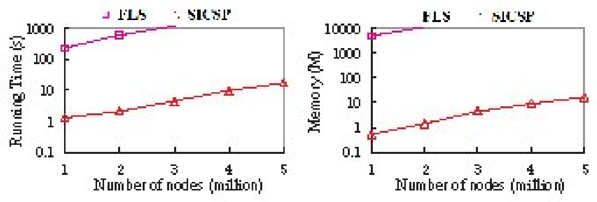
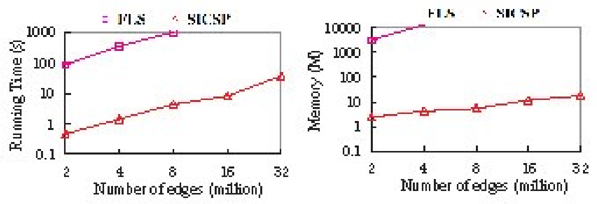


Figure 7: Impact of number of segments of edge-time and edge-weight functions.



(a) Query time (b) Memory overhead

Figure 8: Impact of node size.



(a) Query time (b) Memory overhead

Figure 9: Impact of edge size.

Varying $|V_t|$. We vary $|V_t|$ from 1 million to 5 million, where graphs with 1 million to 5 million are generated from the W-US dataset. In this test, we set $d_s = 20$, $l_s = 800$, $n_f = 15$ and $n_w = 15$. As shown in Figure 8, (1) the querying time and memory overhead of SICSP are always less than those of FLS. SICSP is nearly 300 times faster than FLS. The memory overhead of SICSP is nearly 5,000 times less than that of FLS. (2) The querying time and memory overhead of SICSP increase marginally when the number of nodes increases.

Varying $|E_t|$. We vary $|E_t|$ from 2M to 32M by fixing $|V_t|$ =1M on the W-US dataset. In this test, we set $d_s = 20$, $l_s = 800$, $n_f = 15$ and $n_w = 15$. As shown in Figure 9, (1) the querying time and memory overhead of SICSP increase slightly and are 5.8s and 5.3MB at the graph with 8 million edges. (2) The querying time and memory overhead of FLS grow exponentially, and are beyond 1,000s and 10GB at the graph with 8 million edges.

The results in Exp-1 and Exp-2 justify that SICSP is very efficient and lightweight, and scales well with all metrics. Furthermore, ICSP has very small approximation ratios, despite that SICSP is a heuristic algorithm.

Exp-3: Scalability of **PICSP-T** and **PICSP-F**. This set of experiments evaluates the scalability of parallel algorithm-

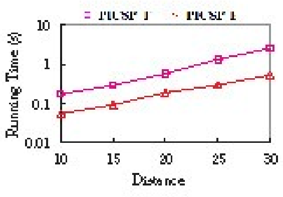


Figure 10: Impact of distance between source and destination.

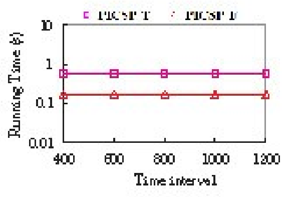


Figure 11: Impact of the length of time interval $[t_{s1}, t_{s2}]$.

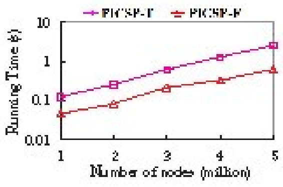


Figure 12: Impact of node size.

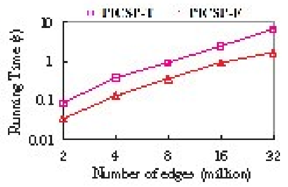


Figure 13: Impact of edge size.

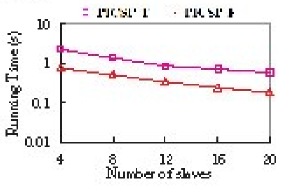


Figure 14: Impact of slave size on the CDU dataset.

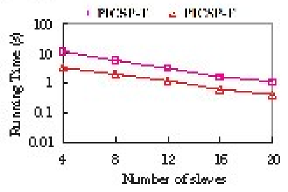


Figure 15: Impact of slave size on the W-US dataset.

s, PICSP-T and PICSP-F. In particular, we only report the querying time because neither PICSP-T nor PICSP-F incurs inter-fragment communications.

Varying d_s and l_s. We vary d_s from 10 to 30 and l_s from 400 to 1,200 on the CDU graph. In this setting, we set $n_f = 15$ and $n_w = 15$, and the number of slaves is set as $n_s = 12$. As shown in Figures 10 and 11, (1) the querying time of PICSP-F is always less than that of PICSP-T, when d_s and l_s take in all values. This is because PICSP-F only computes shortest-distances between the border nodes of slaves, while PICSP-T still calculates shortest-distance for every pair of nodes of G_t. (2) The querying times of PICSP-F and PICSP-T increase when d_s increases, while the querying time does not change as l_s grows.

Varying $|V_t|$ and $|E_t|$. We vary $|V_t|$ from 1M to 5M and $|E_t|$ from 2M to 32M on the W-US dataset. In this setting, we set $d_s = 20$, $l_s = 800$, $n_f = 15$, $n_w = 15$, and the number of slaves is set as $n_s = 12$. As shown in Figures 12 and 13, (1) the querying time of both PICSP-T and PICSP-F scales well with $|V_t|$ and $|E_t|$, but PICSP-F increases more smoothly than PICSP-T. (2) On $|V_t|$, PICSP-T and PICSP-F outperform SICSP by 7.3 times and 22 times on average, respectively, when 12 slaves are used to accelerate the process.

Varying number of slaves. We vary the number of slaves from 4 to 20 on the CDU and W-US datasets. In this setting, we set $d_s = 20$, $l_s = 800$, $n_f = 15$, $n_w = 15$. As shown in Figure 14 on the CDU dataset, (1) PICSP-T and PICSP-F scale well with the increase of slaves: for PICSP-T, the improvement is 4.3 (resp. 3.9) times when the number of slaves increases from 4 to 20. (2) As shown in Figure 15 on the W-US dataset, (1) PICSP-T and PICSP-F also scale well when the number of slaves increases: for PICSP-T, the improvement is 4.3 (resp. 4.6) times when the number of slaves increases from 4 to 20. The two results verify Theorems 5 and 7.

Table 2: Approximation ratios of **PICSP-F**

Distance (d_s)	10	15	20	25	30
Approximation ratio	1.7	2.6	3.5	4.8	5.6
Time interval (l_s)	400	600	800	1,000	1,200
Approximation ratio	2.1	2.8	3.5	3.9	5.3
No. of segments of $f_e(t)$ (n_f)	5	10	15	20	25
Approximation ratio	1.9	2.5	3.5	4.2	5.5
No. of segments of $w_e(t)$ (n_w)	5	10	15	20	25
Approximation ratio	2.1	2.8	3.5	4.1	4.9

Exp-4: Approximation Ratio of **PICSP-F**. Finally, we evaluate the approximation ratios of PICSP-F on the CDU graph by varying d_s from 10 to 30, l_s from 400 to 1,200, n_f and n_w from 5 to 25, since we add ICSP heuristics to make PICSP-F parallel scalable. Table 2 reports the results from which we find the following. (1) The approximation ratios increase when d_s, l_s, n_f and n_w increase, and the average approximation ratios of these parameters are all 3.5. (2) The approximation ratios of PICSP-F are 1.3 times as those of SICSP on average, which explains why it is worthwhile to add heuristics to make PICSP-F parallel scalable. The results in Exp-3 show that PICSP-F always outperforms PICSP-T. Therefore, we only need PICSP-F in practice.

The results in Exp-3 and Exp-4 justify the parallel scalability of PICSP-F and PICSP-T.

6. CONCLUSION

We have proposed a dynamic CSP query by extending the traditional static constrained shortest path to the time-dependent graphs. We have also studied important issues in connection with the dynamic CSP query, from complexity to algorithms to applications. The novelty of this work lies in its adaption of static algorithms to solving new problems, heuristic techniques (plugging structural properties of time and weight functions into the shortest path algorithm), and parallel scalable algorithms to cope with big time-dependent graphs. Our experimental study has verified the feasibility of our proposed algorithms in real-life graphs.

7. 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).

8. REFERENCES

[1] http://faculty.neu.edu.cn/ise/yuanye/vldb2019full.pdf.

[2] Boundary of chengdu. https://www.openstreetmap.org/node/244077729.

[3] Didi chuxing. http://www.didichuxing.com/.

[4] Gaia. http-s://outreach.didichuxing.com/research/opendata/.

[5] Osmconvert. https://wiki.openstreetmap.org/wiki/osmconvert.

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

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

[8] X. Cai, T. Kloks, and C. K. Wong. Time-varying shortest path problems. *Networks*, 29(3):141–150, 2015.

[9] K. L. Cooke and E. Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis & Applications*, 14(3):493–498, 1997.

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

[11] D. Delling. Time-dependent sharc-routing. In *European Symposium on Algorithms*, pages 332–343, 2008.

[12] B. Ding, J. X. Yu, and L. Qin. Finding time-dependent shortest paths over large graphs. In *EDBT*, pages 205–216, 2008.

[13] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* W.H.Freeman, 1979.

[14] G. Y. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

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

[16] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1):36–42, 1992.

[17] C. P. Kruskal, L. Rudolph, and M. Snir. *A complexity theory of efficient parallel algorithms.* Springer Berlin Heidelberg, 1988.

[18] 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*, pages 197–208, 2006.

[19] 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. *PVLDB*, 10(11):1274–1285, 2017.

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

[21] D. H. Lorenz and D. Raz. A simple efficient approximation scheme for the restricted shortest path problem. *Operations Research Letters*, 28(5):213–219, 2001.

[22] K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In *European Symposium on Algorithms*, pages 326–337, 2000.

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

[24] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the Acm*, 37(3):607–625, 1990.

[25] C. S. Peskina. Numerical analysis of blood flow in the heart. *Journal of Computational Physics*, 25(3):220–252, 2015.

[26] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics*, 12(1):2.4, 2008.

[27] F. Schulz, D. Wagner, and K. Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. *Journal of Experimental Algorithmics*, 5(2):12, 2000.

[28] O. J. Smith, N. Boland, and H. Waterer. Solving shortest path problems with a weight constraint and replenishment arcs. *Computers & Operations Research*, 39(5):964–984, 2012.

[29] G. Tsaggouris and C. Zaroliagis. Multiobjective optimization: Improved fptas for shortest paths and non-linear objectives with applications. *Theory of Computing Systems*, 45(1):162–186, 2009.

[30] A. Veneti, C. Konstantopoulos, and G. Pantziou. Continuous and discrete time label setting algorithms for the time dependent bi-criteria shortest path problem. In *14th INFORMS Computing Society Conference*, 2014.

[31] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *ICDE*, pages 568–579, 2014.

[32] 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*, pages 967–982, 2015.

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

[34] 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*, pages 145–156, 2016.

[35] Y. Yang, H. Gao, J. X. Yu, and J. Li. Finding the cost-optimal path with time constraint over time-dependent graphs. *PVLDB*, 7(9):673–684, 2014.

[36] Y. Yuan, X. Lian, G. Wang, L. Chen, Y. Ma, and Y. Wang. Weight-constrained route planning over time-dependent graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 914–925, 2019.