Algorithmica (2012) 62:416–435
DOI 10.1007/s00453-010-9461-6

Shortest Paths in Time-Dependent FIFO Networks

**Frank Dehne · Masoud T. Omran ·
Jörg-Rüdiger Sack**

Received: 5 April 2010 / Accepted: 6 October 2010 / Published online: 28 October 2010

Abstract In this paper, we study the *time-dependent shortest paths problem* for two types of time-dependent FIFO networks. First, we consider networks where the availability of links, given by a set of disjoint time intervals for each link, changes over time. Here, each interval is assigned a non-negative real value which represents the travel time on the link during the corresponding interval. The resulting shortest path problem is the time-dependent shortest path problem for availability intervals ($\mathcal{TDSP}_{\text{int}}$), which asks to compute all shortest paths to any (or all) destination node(s) d for all possible start times at a given source node s. Second, we study time-dependent networks where the cost of using a link is given by a non-decreasing piece-wise linear function of a real-valued argument. Here, each piece-wise linear function represents the travel time on the link based on the time when the link is used. The resulting shortest paths problem is the time-dependent shortest path problem for piece-wise linear functions ($\mathcal{TDSP}_{\text{lin}}$) which asks to compute, for a given source node s and destination d, the shortest paths from s to d, for all possible starting times.

We present an algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem that runs in time $O((F_d + \gamma)(|E| + |V| \log |V|))$ where F_d is the output size (i.e., number of linear pieces needed to represent the earliest arrival time function to d) and γ is the input size (i.e., number of linear pieces needed to represent the local earliest arrival time functions for all links in the network). We then solve the $\mathcal{TDSP}_{\text{int}}$ problem in

Research supported by NSERC, SUN Microsystems of Canada and HPCVL.

F. Dehne · M.T. Omran · J.-R. Sack (✉)
Carleton University, Ottawa, Canada
e-mail: sack@scs.carleton.ca

F. Dehne
e-mail: dehne@scs.carleton.ca

M.T. Omran
e-mail: mtomran@scs.carleton.ca

$O(\lambda(|E| + |V|\log|V|))$ time by reducing it to an instance of the $\mathcal{TDSP}_{\text{lin}}$ problem. Here, λ denotes the total number of availability intervals in the entire network. Both methods improve significantly on the previously known algorithms.

Keywords Time-dependent networks · Earliest arrival time · Shortest paths · FIFO property · Algorithms

1 Introduction

The *shortest path problem* for a network (directed graph $G = (V, E)$) is a fundamental problem in graph theory with many applications related e.g. to road networks and transportation. It is of both, theoretical and practical interest. Although well-studied conventional *static* shortest path algorithms (see e.g. [7]) play a fundamental role in applications with non-changing nature, many real-world applications and their underlying networks are changing over time (e.g., applications in transportation science, computer networks, robotics, and VLSI design). A more realistic approach for these networks with dynamically changing characteristics is to take *time* into consideration. For example, in a road network, the shortest path from a given source node to a destination node during rush hour is not the same as during low traffic periods.

Depending on which characteristics of the underlying network are changing over time, different models have been applied. For networks where the characteristics of a link-traversal depends on the time when the link is traversed, the most utilized model is the *time-dependent network* (see e.g., [2, 4, 5, 9, 12, 17, 18, 21]) which is also used in this paper. The time-dependent network model includes a wide range of dynamic networks because link-traversal costs can be modeled with arbitrary functions. In [19], it has been shown that the general shortest paths problem on time-dependent networks is NP-Hard. However, there are variants of the problem that have polynomial time solutions and are of interest in real-world applications (including road networks which are the most common applications of the time-dependent shortest paths problem). In this paper, we consider shortest paths problems for two types of time-dependent networks which will be introduced in the next two subsections.

1.1 The $\mathcal{TDSP}_{\text{int}}$ Problem

The first time-dependent shortest paths problem, $\mathcal{TDSP}_{\text{int}}$, is defined for a network $G_i(V, E)$, where the *availability* of links changes over time. Each link $e = (u, v) \in E$ is assigned a set I_e of disjoint time intervals, referred to as *availability intervals*, during which the link is available. Availability intervals can model e.g. situations where some roads or lanes of a road network are only available during certain times. Typical examples include no left-turns during rush hours, border crossings that are closed during the night, and lanes restricted for buses during certain peak periods. During times outside the set I_e of time intervals, the link e is not available and one has to wait until the start of the next availability interval in I_e. For every availability interval $i = [l_e^i, r_e^i] \in I_e$ ($l_e^i, r_e^i \in \mathbf{R}^+$ and $l_e^i \le r_e^i$) on e is defined a non-negative cost $w_e^i \in \mathbf{R}^+$ which represents the time needed to traverse e during availability interval i. More precisely, starting from node u of $e = (u, v)$ at time ST_u, $l_e^i \le ST_i \le r_e^i$, one arrives at node v at time $AT_v = ST_u + w_e^i$. Let $j = [l_e^j, r_e^j] \in I_e$ be the interval immediately following i in ascending time order. Our model captures waiting at nodes in that for $ST_u, r_e^i < ST_u < l_e^j$, one can wait at u until the next interval is available. The start time would then be $ST_u = l_e^j$, resulting in arrival time $AT_v = l_e^j + w_e^j$. Note that for starting times greater than the last interval on the link, e is not available and $AT_v = \infty$. Consider a path $p = \langle v_0, v_1, \dots, v_k, v_{k+1}\rangle$ in a network $G_i(V, E)$. We say p is valid if there exist starting times $ST_{v_0}, \dots, ST_{v_k}$ and arrival times $AT_{v_1}, \dots, AT_{v_{k+1}}$ such that $ST_{v_i} \in I_{(v_i, v_{i+1})}$, $AT_{v_{i+1}} = ST_{v_i} + w^i_{(v_i, v_{i+1})}$, and $AT_{v_{i+1}} \le ST_{v_{i+1}}$ for all $0 \le i \le k$. Here, $ST_{v_i} \in I_{v_i, v_{i+1}}$ denotes that there exists an interval $i \in I_{(v_i, v_{i+1})}$ such that $ST_{v_i} \in i$. The arrival time of a valid path p at v_{k+1} for start time ST_{v_0} at v_0 is $AT_{v_{k+1}}$. If path p is invalid, then $AT_{v_{k+1}} = \infty$.

For the $\mathcal{TDSP}_{\text{int}}$ problem our goal is to find, for all starting times at a given source node $s \in V$, the earliest arrival time at all destination node $d \in V$, considering all valid paths from s to d. A path p that returns the earliest arrival time at destination node d for a given start time t at source node s is called the *shortest path to d for start time t at s*. We use the terms earliest arrival time and shortest path interchangeably. In this paper, we solve the $\mathcal{TDSP}_{\text{int}}$ problem by reducing it to a special instance of $\mathcal{TDSP}_{\text{lin}}$ defined in the next subsection. Note that G_i is assumed to be a FIFO network in that the FIFO property holds for every link of the network. This is the usual assumption in the literature (see e.g. [10, 16, 17]). The FIFO property on links implies that for every link (u, v), a later start time at u results in a later (or equal) arrival time at v. In other words, for every link (u, v) the arrival time function to v for different departure times from u is non-decreasing. Note that, we do not allow unnecessary waiting at a node as it would result in a later arrival time (due to the network's FIFO property). Also note that, using the approach described in [18], any non-FIFO network that allows unrestricted waiting on nodes can be converted to a FIFO network with zero waiting on nodes.

1.2 The $\mathcal{TDSP}_{\text{lin}}$ Problem

The second problem that we solve in this paper is the $\mathcal{TDSP}_{\text{lin}}$ problem which is a more general version of $\mathcal{TDSP}_{\text{int}}$. Here, we consider a time-dependent network $G_l(V, E)$ in which each link $e = (u, v) \in E$ is assigned a piece-wise linear function $a_e(t)$ denoting the arrival time at v for start time t at u. The notion of arrival time function is also extendable to a path in the network. Suppose $p = \langle v_1, v_2, \dots, v_k, v_{k+1}\rangle$ is a path of length k. Then, starting from v_1 at time t one arrives at v_k at time $a_{(v_k, v_{k+1})}(\dots(a_{(v_2,v_3)}(a_{(v_1,v_2)}(t))))$ which we denote by $a_p(t)$. Piece-wise linear functions can be used as an approximation of non-linear arrival time functions. They model time-dependent networks where costs are changing linearly during specific periods of time. An example is a road network where travel times on some links increase linearly when snowfall piles up on the road. This can also model e.g. snow melting periods and snow removal schedules. In $\mathcal{TDSP}_{\text{lin}}$, our goal is to find for all possible start times t at source node s the earliest arrival time at a destination node d. In other words, considering P to be the set of all possible paths from s to d, we are interested in finding the minimum arrival time function $A_{sd}(t) = \min_{p \in P}(a_p(t))$.

Note that we only consider cases where all arrival time functions are piece-wise linear real-valued functions. Additionally, the $a_e(t)$ functions are non-decreasing for all $e \in E$ so that the FIFO property holds.

The problem of finding the earliest arrival time (or shortest path) between two points for a particular given starting time can be solved efficiently by applying straight-forward modifications to well-known static shortest path methods. For example, a simple modification to Dijkstra's shortest path algorithm [11] using Fredman and Tarjan's implementation [13] finds the earliest arrival time for a given start time in $O(|E|\log\gamma_{max} + |V|\log|V|)$ with the same approach as used in [4]. Here, γ_{max} is the maximum number of linear pieces on link arrival time functions, $a_e(t)$. However, solving the problem for all possible starting times appears to be harder even when the $a_e(t)$ functions are piece-wise linear and the FIFO property holds, since the shortest path to a destination node d from source node s depends on the start time from s. A naive algorithm for solving the $\mathcal{TDSP}_{\text{lin}}$ problem computes the earliest arrival time function for every possible path from s to d and then calculates the lower envelope. This algorithm would need exponential time in many cases because many networks would have an exponential number of possible paths between s and d. In fact, as stated in [19], the problem is NP-hard in its general form (arbitrary arrival time functions).

1.3 Results for the $\mathcal{TDSP}_{\text{lin}}$ Problem

In this paper, we present a novel approach for solving $\mathcal{TDSP}_{\text{lin}}$ which improves on the previous results. Our method is based on the observation that the label-setting and label-correcting algorithms (discussed in Sect. 2.1) compute earliest arrival time functions for all intermediate nodes of the network. We present new, non-trivial, combinatorial properties of arrival time functions. These new insights allow us to focus on finding the earliest arrival time function for the destination node only, and only at crucial time-points. This way, we can discard unnecessary computations. In contrast to previous methods, our algorithms directly compute the earliest arrival time function for every destination node d by tracing time and finding the earliest arrival time only at time instances that may change the earliest arrival time function for d.

Our new algorithm for solving the $\mathcal{TDSP}_{\text{lin}}$ problem runs in time $O((F_d + \gamma)(|E| + |V|\log|V|))$, where F_d is the output size (number of linear pieces needed to represent the earliest arrival time function) and γ is the input size (total number of linear pieces in all link arrival time functions). Our method improves significantly upon the previously known methods. Orda and Rom's label-correcting method [18] had an $O(F_{max}|V||E|)$ time bound where F_{max} is the maximum output size F_d for all possible destination nodes $d \in V$, and Dean [10] had improved that with a label-setting algorithm that has a running time of $O(|E|F^* \log|V|)$, where F^* is the total number of pieces among all output functions. Dean [10] also conjectured that it is possible to have super-polynomial output size for the earliest arrival time function on some nodes of the network. Our method will have super-polynomial time only if the output size is super-polynomial.

1.4 Results for the $\mathcal{TDSP}_{\mathrm{int}}$ Problem

In order to solve the $\mathcal{TDSP}_{\mathrm{int}}$ problem, we convert every instance of $\mathcal{TDSP}_{\mathrm{int}}$ into an instance of the $\mathcal{TDSP}_{\mathrm{lin}}$ where the slope of each linear piece is either 0 or 1. The algorithm determines for every start time s the earliest possible arrival time at a given destination d or all nodes of the network. Our approach solves the $\mathcal{TDSP}_{\mathrm{int}}$ problem with $O(\lambda(|E| + |V|\log|V|))$ time complexity where λ denotes the total number of availability intervals in the entire network. When applied to the $\mathcal{TDSP}_{\mathrm{int}}$ problem, Orda and Rom's algorithm [18] requires time $O(\lambda|V||E|)$ and Dean's algorithm [10] requires time $O(\lambda|E|\log|V|)$. A crucial observation that supports our approach is the fact that there cannot be more than $O(\lambda)$ pieces in the earliest arrival time function for d. This reduces the number of paths and time instances that need to be considered and improves performance. The improved method uses structural properties of earliest arrival functions which are of interest in their own right.

1.5 Possible Improvements for Special Cases

Our method makes extensive use of static shortest paths algorithms as a base to build the earliest arrival functions to each node. We note that, our algorithm is independent of the static shortest paths algorithm used. One may use more efficient static shortest path algorithms for further improvement in special cases. For example, Thorup's linear time algorithm [22] could be used if link costs are positive integers. In case that the underlying network is a planar network, Henzinger et al. [15] proposed a linear time algorithm for the static shortest paths problem that could be applied to our algorithm. Heuristics like A* algorithms [14] could be applied as well.

1.6 Organization of the Paper

The remainder of this paper is organized as follows. Section 2 presents our new algorithm for the $\mathcal{TDSP}_{\mathrm{lin}}$ problem and analyzes its running time and correctness. Section 3 presents a new algorithm for the $\mathcal{TDSP}_{\mathrm{int}}$ problem that is based on our $\mathcal{TDSP}_{\mathrm{lin}}$ solution. Section 4 concludes the paper.

2 The Shortest Paths Problem on a Time-Dependent Network with Piece-Wise Linear Functions ($\mathcal{TDSP}_{\mathrm{lin}}$)

In this section, we consider the shortest paths problem $\mathcal{TDSP}_{\mathrm{lin}}$ on a time-dependent network in which link arrival time functions, $a_e(t)$ $e \in E$, are piece-wise linear real-valued functions of a real argument. Additionally, we assume that $a_e(t)$ functions are non-decreasing for all $e \in E$, so that the FIFO property holds which is the usual assumption in literature (see e.g. [10, 16, 17]). We present and analyze a new algorithm for $\mathcal{TDSP}_{\mathrm{lin}}$ after giving a summery of previous work.

2.1 Previous Algorithms and Results

The problem of finding the shortest paths in a time-dependent network (also referred to as "earliest arrival time" and "minimum travel time" problem in some texts) was first proposed in 1966 by Cooke and Halsey [6]. They considered time to have discrete values. Most current results are obtained for continuous time because it better models real scenarios.

For the $\mathcal{TDSP}_{\text{lin}}$ problem, the goal is to find the earliest arrival time from s to d for all starting times t from s. The arrival time on each path is a function of the start time from the source node and can be obtained by computing the composition of the link arrival time functions along the path. Consequently, the solution of the $\mathcal{TDSP}_{\text{lin}}$ problem is the minimum of all arrival time functions on all paths from s to d. This leads to a naive algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem: compute the arrival time functions of all paths from s to d and compute their lower envelope. For more information on lower envelope algorithms see e.g. [20]. Although this gives the correct solution, such an algorithm is not efficient in that there could be an exponential number of paths from s to d leading to exponential time complexity. In [16] a heuristic algorithm has been presented to solve the $\mathcal{TDSP}_{\text{lin}}$ problem. The algorithm uses an approach similar to A* algorithms [14], but its worst case complexity is unknown.

In the following, we review exact algorithms for the $\mathcal{TDSP}_{\text{lin}}$ problem. We present two approaches known as *label correcting* and *label setting* in separate sections.

2.1.1 Label-Correcting Algorithms

A slightly modified version of the standard label-correcting algorithms for shortest paths [3] (e.g., Bellman-Ford) solves the $\mathcal{TDSP}_{\text{lin}}$ problem. Here, instead of computing labels for a specific time, one can do this simultaneously for all values of t. In this case, arrival time functions are used instead of scalar arrival times at each node. Orda and Rom [18] proposed such an algorithm. For a FIFO network with piece-wise linear functions, it has time complexity $O(F_{max}|V||E|)$, where F_{max} is the maximum number of linear pieces needed to represent the earliest arrival time function between s and any node in the network. Our algorithm is a significant improvement of this method as well as the methods outlined in the following section.

2.1.2 Label-Setting Algorithms

In a label-setting algorithm the goal is to compute, in small pieces, actual correct values of output functions rather than iteratively revising these functions. This approach is similar to Dijkstra's algorithm for the static shortest path problem [11]. In contrast to label-correcting algorithms, it is not possible to simply replace scalar label values by functions to solve the problem because a minimum element (i.e., one function which is minimum over the entire domain) may not exist. The main idea of these algorithms is to determine the latest time ϕ_v, for each node v, so that the current earliest arrival time function for any time less than ϕ_v gives the actual earliest arrival time at the node. For FIFO networks with piece-wise linear arrival time functions, Dean [10] suggested a label-setting algorithm that performs a single chronological scan through time to establish output functions. The algorithm employs the same approach used for solving parametric shortest path problems. In the worst-case, this algorithm has a running time of $O(|E|F^* \log|V|)$, where F^* is the total number of pieces among all

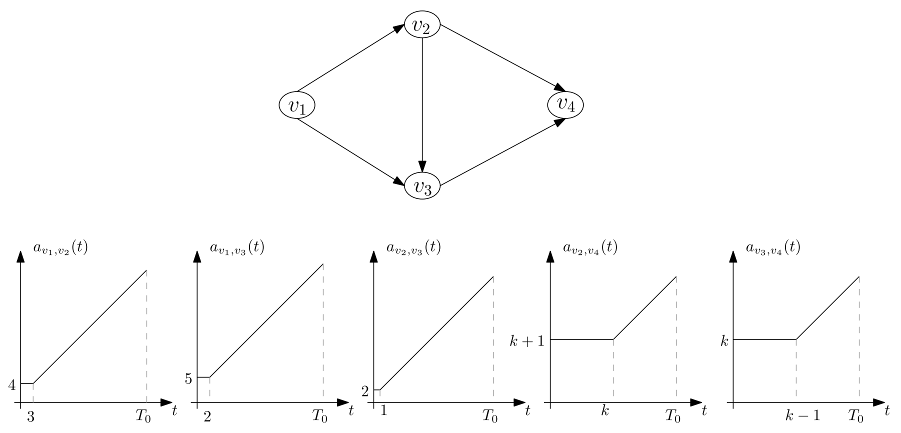


Fig. 1 An example network for which the algorithm by Ding et al. [12] requires $\alpha = k - 3$ iterations where k is the *value* of the arrival time function for (v_3, v_4) at time t_s

output functions. As discussed in [10], this "slightly improves on the worst-case running time in the label-correcting algorithm in the amount of time required per linear piece of the output".

Recently, Ding et al. [12] presented a simpler label-setting algorithm for the $\mathcal{TDSP}_{\text{lin}}$ problem in FIFO networks with piece-wise linear functions. The algorithm scans a sequence of time steps the size of which depends on the values of the arrival time functions. Careful analysis of this algorithm shows that this approach yields a solution with time complexity $O(\alpha(|E| + |V| \log |V|))$ which contains a factor α that depends on the *values* of arrival time functions and can be arbitrarily large, independent of the input size or output size. An example of an instance where α is independent on $|E|, |V|$ and γ, and can be arbitrarily large is depicted in Fig. 1. In the following, we briefly explain how the algorithm by Ding et al. [12] would process the network shown in Fig. 1. For more details on the algorithm, refer to [12]. Consider a network $G(V, E, A)$ as shown in Fig. 1 with starting node $v_s = v_1 \in V$, and destination node $v_e = v_4 \in V$ where V and E are the node and edge sets, and A is the set of piece-wise linear function on links. Observe the values $k + 1$ and k for the arrival time functions of (v_2,v_4) and (v_3,v_4) at time t_s, respectively. Note that, k can be chosen arbitrarily large, without changing the input size or output size. Given a total time interval $T = [t_s, t_e] = [0, T_0]$, $T_0 > k$, the problem is to find the earliest arrival time function at v_e for all starting times in T from v_s. Let $g_i(t)$ be the earliest arrival time function from v_s to v_i for all $v_i \in V$. The algorithm in [12] finds sub-intervals $I_i = [t_s, \tau_i]$ such that for all $t \in I_i$, g_i reflects the correct earliest arrival time function. As the algorithm proceeds, the intervals are extended until they cover T. The algorithm stops when $g_e(t)$ is defined for all $t \in T$ or if v_e is not reachable from v_s. The main contribution of [12] is the way they extend the interval I_i on each node. The algorithm maintains a priority queue which holds $(\tau_i, g_i(t))$, $v_i \in V$ values sorted by ascending order of $g_i(\tau_i)$. At every iteration, it dequeues the top element of the queue in $(\tau_i, g_i(t))$ and assigns the next top element to $(\tau_l, g_l(t))$ without dequeuing. These values are then used to compute the next earliest possible arrival

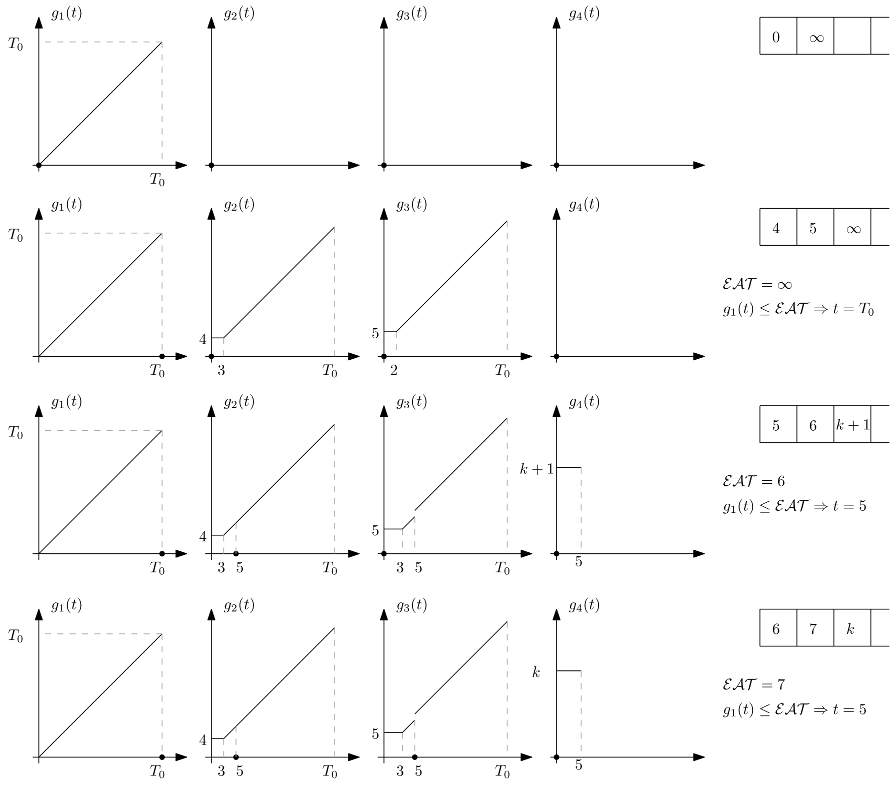


Fig. 2 Illustration of the Ding et al. algorithm for the network shown in Fig. 1

time from v_s to v_i via any edge (v_f, v_i), namely $\mathcal{EAT} = \min\{a_{f,l}(\tau_l),\ (v_f, v_l) \in E\}$ which is then used to extend τ_i. They show that τ_i can be extended to τ_i' which is the latest starting time t that satisfies $g_i(t) \le \mathcal{EAT}$. For any link (v_i, v_j), the new value $g_j'(t) = a_{i,j}(g_i(t)),\ t \in [\tau_i, \tau_i']$ is then compared with $g_j(t)$ and the minimum value is stored. Figure 2 shows a few iterations of the Ding et al. algorithm for our simple network shown in Fig. 1. The first row, shows the starting state of the algorithm where $g_1(t) = t$ and $\tau_i = 0$ for $i \in \{1, 2, 3, 4\}$. The current value of τ_i is indicated by a black dot on the x-axis. The priority queue is shown on the right. All $g_i(\tau_i)$ values in the priority queue are ∞ except for $g_1(0)$ which is 0. In the first iteration (second row in Fig. 1), $(\tau_1, g_1(t))$ is removed from the priority queue and the next element of the priority queue is used to obtain the new value of τ_1. The next top element of the priority queue has value $g_l(\tau_l) = \infty$ and there is no incoming edge to v_s, indicating that τ_1 can be extended to $t_e = T_0$. Then, the $g_2(t)$ and $g_3(t)$ functions will be updated, as well as their corresponding values in the priority queue. In the next iteration, $(\tau_2, g_2(t))$ is removed form the priority queue which has value $g_2(\tau_2) = 4$ and the next element is $g_l(\tau_l) = 5$. For all incoming links to v_2, the smallest arrival time for starting time 5 is $\mathcal{EAT} = 6$. Hence, we can extend τ_2 to 5 and update g_3 and g_4 for interval $[0, 5]$. We also update the priority queue with new values of $g_i(\tau_i)$ for $i \in \{3, 4\}$. In the third iteration, $(\tau_3, g_3(t))$ is removed from the priority queue and the next element would be $g_l(\tau_l) = 6$. For all incoming links to v_3, the smallest arrival time for starting time 6 is $\mathcal{EAT} = 7$. Hence, we can extend τ_3 to 5 and update g_4 for interval $[0, 5]$. We also update the priority queue to reflect the updated value of $g_4(\tau_4)$. We observe that in all subsequent iterations, the elements removed from the priority queue toggle between $g_2(\tau)$ and $g_3(\tau)$ and increase only by value 1 in each iteration until they reach value of k. Hence, the algorithm of Ding et al. [12], applied to the network shown in Fig. 1, requires $\alpha = k - 3$ iteration. We recall that $k + 1$ and k are the *values* for the arrival time functions of (v_2, v_4) and (v_3, v_4) at time $t_s = 0$, respectively. Hence, $\alpha = k - 3$ can be chosen arbitrarily large without changing the network size, input size or output size.

2.2 Structural Properties

Our new algorithm makes extensive use of certain structural properties of the problem discussed in this section.

2.2.1 Structural Properties of the Earliest Arrival Time Function

The earliest arrival time function from s to d, $A_{sd}(t)$, is a piece-wise linear function since all input arrival time functions are assumed to be piece-wise linear functions and the function operators used to compute $A_{sd}(t)$ (*function inverse, linear combination, function compound, min, max*) do not change linearity of the result.

We are interested in the points on the curve $A_{sd}(t)$ that connect its different linear pieces, and will refer to them as *breakpoints*. We differentiate between two types of breakpoints. First, a breakpoint may represent the intersection between two pieces of arrival time functions on different paths. Second, a breakpoint may represent a breakpoint on one of the arrival time functions for a path from s to d. We refer to the first type as X-point and to the second type as V-point. Figure 3 depicts an arrangement of arrival time functions for four paths and identifies X and V-points.

Every V-point corresponds to a breakpoint on the arrival time function, $a_p(t)$, for some path p from s to d. Each breakpoint on the $a_p(t)$ function is the result

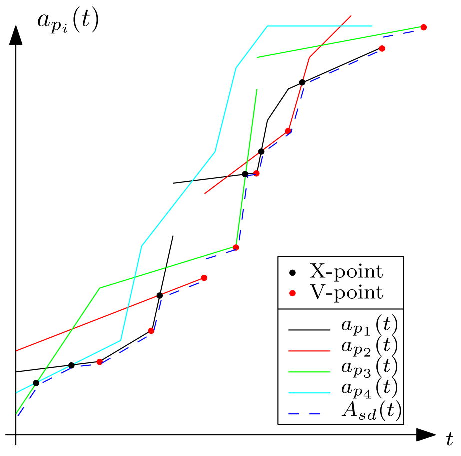


Fig. 3 An illustration of X and V-points. Curves $a_{p_1}(t), \ldots, a_{p_4}(t)$ are arrival time function for four paths $p_1, \ldots, p_4$, and $A_{sd}(t)$ is the final arrival time function from s to d

of a breakpoint between two linear pieces of arrival time functions on a link of p introduced because of a compound operation for computing $a_p(t)$. In the following lemma, we will show that every breakpoint of a link arrival time function can create at most one V-point on $A_{sd}(t)$.

Lemma 1 *Suppose P is the set of all paths that go through link $e = (v, w) \in E$ and $a_e(t)$ is the arrival time function for e and has γ_e breakpoints. Then, all arrival time functions $a_p(t)$, $p \in P$, create, in total, at most γ_e V-points on $A_{sd}(t)$.*

Proof Consider the following representation of the piece-wise linear function $a_e(t)$:

$$a_e(t) = \begin{cases} \alpha^1 t + \beta^1, & 0 \le t \le T^1, \\ \alpha^2 t + \beta^2, & T^1 < t \le T^2, \\ \vdots & \vdots \\ \alpha^{\gamma_e} t + \beta^{\gamma_e}, & T^{\gamma_e - 1} < t \le T^{\gamma_e}, \\ \infty, & T^{\gamma_e} < t. \end{cases}$$

For every breakpoint $T^i, i = 1 \dots \gamma_e$, 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, w), and a path with earliest arrival time (EAT) to d which starts from w at time $\alpha^i T^i + \beta^i$. Because of the definition of p^i, for any path $p \in P$ other than p^i, T^i will create a breakpoint either at (LST, EAT) or to the left and above this point. Since (LST, EAT) is on $a_{p^i}(t)$ and the FIFO property holds, any points that are to the left and above (LST, EAT) are not on $A_{sd}(t)$. Hence, other paths cannot create new breakpoints on $A_{sd}(t)$. This proves that all arrival time functions $a_p(t)$, $p \in P$, create in total at most γ_e V-points on $A_{sd}(t)$. □

Let $\gamma = \sum_{e \in E} \gamma_e$ be the total number of breakpoints on link arrival time functions in the entire network. Since every V-point arises from a breakpoint on some link arrival time function, Lemma 1 implies that there cannot be more than $O(\gamma)$ V-points on $A_{sd}(t)$.

2.2.2 Possibly Super-Polynomial Output Size

In [10], the author conjectured that in a FIFO network with piece-wise linear link arrival time functions, the earliest arrival time function $A_{sd}(t)$ from a source node s to a destination node d may have more than a polynomial number of linear pieces. This means that there possibly exist network structures that result in super-polynomial complexity for earliest arrival time functions to some nodes of the network.

The *super-polynomial structure* could appear as a subnetwork of the actual input network, resulting in earliest arrival time functions with possibly super-polynomial number of pieces (breakpoints) for destination nodes whose shortest path from s passes through the super-polynomial structures. However, the earliest arrival time function from s to d could still be of linear size since the earliest arrival time path may not intersect the super-polynomial structure at all. In this case, F_{max} would be of super-polynomial size and F_d would be of linear size.

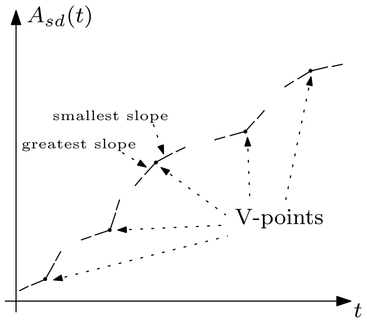
2.3 A New Algorithm for Instances with Polynomial Size Output

Our new algorithm is based on the idea that instead of building all earliest arrival time functions to all nodes in the network, we calculate only the earliest arrival time function to destination node d. An outline of our method is given in Algorithm 1. In the remainder, we discuss our method in more detail.

The main idea is to find all starting times for which the earliest arrival time function from s to d, $A_{sd}(t)$, changes from one linear piece to another as well as linear functions on the left and right of these breakpoints. In Sect. 2.2, we introduced two types of breakpoints in $A_{sd}(t)$: V-points and X-points. We also showed that at most $O(\gamma)$ V-points exist on $A_{sd}(t)$, where γ is the total number of pieces in all input arrival time functions. All V-points that can potentially be on $A_{sd}(t)$ could be captured by computing, for every breakpoint at time T on the arrival time function $a_e(t)$ of each link $e=(v,w)$, the latest departure time (LDT) at s to arrive at v at time T and the earliest arrival time (EAT) at d for departure time $a_e(T)$ at w. Point (LDT, EAT) is a potential V-point on $A_{sd}(t)$. Note that the earliest arrival time at a destination node could be obtained by running a modified version of Dijkstra's static shortest path algorithm for a given start time at the source node. Similarly, a modified version of reverse Dijkstra's static shortest path algorithm could be used to obtain the latest departure time at the source node corresponding to a given arrival time at the destination node. Regarding the reversibility of the $\mathcal{TDSP}_{\text{lin}}$ problem see [8].

In order to make sure that (LST, EAT) is on the final solution, we calculate the earliest arrival time to d starting from s at time LST. If the result arrival time is the same as EAT, then (LST, EAT) is indeed a V-point on $A_{sd}(t)$. In this case, we also compute the linear pieces to the left and right of each V-point. These are pieces with the earliest arrival time and the smallest (greatest) slope on the right (left) vicinity of each V-point. Figure 4 shows a sample $A_{sd}(t)$ function after all V-points have been detected. Lines 2 through 11 of Algorithm 1 state the pseudo-code to find all V-points along with their left and right linear functions. Note that, given a time t_0, the adjacent linear pieces with smallest (greatest) slope can be computed while computing the earliest arrival time to d for starting time t_0. This is accomplished by keeping the product of slopes for each node in the shortest path tree as a secondary key when Dijkstra's algorithm finds two or more entries of the heap that have the same arrival time value. In this case, selecting the entry with smallest (greatest) slope leads to the

Fig. 4 A sample $A_{sd}(t)$ function with all V-points and their adjacent linear functions



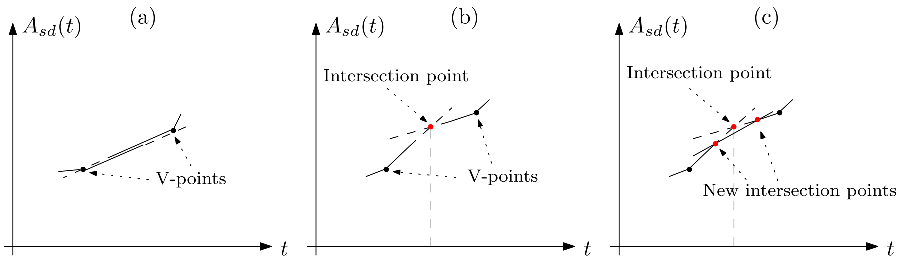


Fig. 5 (**a**) Overlapping pieces. (**b**) Intersection point on $A_{sd}(t)$. (**c**) Intersection point hidden by another linear piece

adjacent linear pieces with smallest (greatest) slope. To show the correctness of this approach consider any two paths from s to d starting at time t_0 with equal arrival times but different slopes on their arrival time functions. The first time where they have equal arrival time values is either at d or at some earlier node d'. In the latter case, they will share the same path from d' to d. In either case, selecting the smallest (greatest) product of slope values from the heap, among equal arrival time values, maintains the smallest (greatest) slope.

Thus far we have determined all V-points and the slope of $A_{sd}(t)$ in their vicinity. We build the remaining part of $A_{sd}(t)$ by adding all X-points and missing pieces between every pair of consecutive V-points on $A_{sd}(t)$. Due to the linearity of the input arrival functions, the X-points between two consecutive V-points are in concave position (seen from below). Consider two consecutive V-points, V_l and V_r, found in the previous step together with the linear pieces in their vicinity. Two cases arise for the linear pieces to the right of V_l and to the left of V_r. They either overlap, or they intersect in some point $I = (x_I, y_I)$. If they overlap, the piece connecting the two V-points is part of the solution (Fig. 5(a)). In case of an intersection, two cases are possible. First, if calculating the earliest arrival time for starting time x_I at s returns the same arrival time y_I at d as for the intersection point, the pieces v_l to I and I to v_r are part of the solution (Fig. 5(b)). Second, if calculating the earliest arrival time for starting time x_I at s returns a value less than y_I at d, there exists a new linear piece that hides the intersection point $I = (x_I, y_I)$ (Fig. 5(c)). The linear pieces to the right of V_l and to the left of V_r intersect the new piece, and we recurse for new intersections. See Theorem 1 for further details. Lines 12 through 35 of Algorithm 1 are the pseudo-code for finding X-points and adding pieces to solution function $A_{sd}(t)$.

Theorem 1 *Given a source node s and destination node d, the $\mathcal{TDSP}_{\text{lin}}$ algorithm (Algorithm 1) outlined above correctly determines $A_{sd}(t)$ for all $t \in [0, \infty)$.*

Proof The algorithm first finds all V-points on $A_{sd}(t)$ along with linear pieces to the left and right of each V-point. From the proof of Lemma 1, if follows that no V-points other than those considered by Algorithm 1 can be on $A_{sd}(t)$. The algorithm picks every two consecutive V-points to compute all X-points between them. Let $v_l = (x_l, y_l)$ and $v_r = (x_r, y_r)$ be two consecutive V-points on $A_{sd}(t)$. Suppose

```
begin
    $A_{sd}(t) \leftarrow NULL$
    for every link $e = (v, w) \in E$ do
        for $i = 0$ to $\lambda_e$ do
            $LST \leftarrow ReverseDijkstra^*(v, s, T_e^i)$ /* Returns the latest departure time at
            $s$ corresponding to arrival time $T_e^i$ at $v$. $T_e^i$ is the time of $i^{th}$
            breakpoint on $a_e(t)$. */
            $EAT \leftarrow Dijkstra^*(w, d, a_e(T_e^i))$ /* Returns the earliest arrival time at $d$
            for departure time $a_e(T_e^i)$ at $w$. */
            $TMP \leftarrow Dijkstra^*(s, d, LST)$
            if $EAT = TMP$ then
                $f_l \leftarrow LeftFunction(s, d, LST)$ /* Returns the linear function on $A_{sd}(t)$
                which is on the right of V-point at time $LST$. */
                $f_r \leftarrow RightFunction(s, d, LST)$      /* Returns the linear function on
                $A_{sd}(t)$ which is on the left of V-point at time $LST$. */
                $InsertToList(L, \{LST, EAT, f_l, f_r\})$ /* Adds the information of V-point
                at time $LST$ to list $L$. */
            end
        end
    end
    $Sort(L)$           /* Sorts list $L$ by the ascending order of $LST$ values. */
    $\{LST_1, EAT_1, LF_1, RF_1\} \leftarrow RemoveItem(L)$          /* Returns and removes the first
    element of $L$. Extracts the information for the left V-point. */
    while $NotEmpty(L)$ do
        $\{LST_2, EAT_2, LF_2, RF_2\} \leftarrow RemoveItem(L)$      /* Returns and removes the first
        element of $L$. Extracts the information for the right V-point */
        if $Overlap(RF_1, LF_2)$ then
            /* If the function on the right of left V-point has the same
            slope as the one on the left of right V-point.                */
            $AddLinearPiece(A_{sd}(t), RF_1, LST_1, LST_2)$
        else
            /* If the function on the right of left V-point has different
            slope from the one on the left of right V-point.              */
            $(PX_1, PY_1) \leftarrow (LST_1, EAT_1)$
            $(PX_2, PY_2) \leftarrow IntersectionPoint(RF_1, LF_2)$ /* Finds the intersection point
            of the function on the right of left V-point and the one on the
            left of right V-point. */
            $Push(S, (PX_2, PY_2, RF_1, LF_2))$          /* Inserts the information of the
            intersection point to stack $S$. */
            while $NotEmpty(S)$ do
                $(PX_2, PY_2, f_l, f_r) \leftarrow Pop(S)$   /* Returns and removes an element from
                the top of stack $S$. */
                $TMP \leftarrow Dijkstra^*(s, d, PX_2)$
                if $TMP = PY_2$ then
                    $AddLinearPiece(A_{sd}(t), f_l, PX_1, PX_2)$     /* Adds linear function $f_l$
                    to $A_{sd}(t)$ from time $PX_1$ to $PX_2$.   */
                    $PX_1 \leftarrow PX_2$
                else
                    $f_m \leftarrow LeftFunction(s, d, PX_2)$
                    $(IX_1, IY_1) \leftarrow IntersectionPoint(f_l, f_m)$
                    $(IX_2, IY_2) \leftarrow IntersectionPoint(f_m, f_r)$
                    $Push(S, (IX_2, IY_2, f_m, f_r))$
                    $Push(S, (IX_1, IY_1, f_l, f_m))$
                end
            end
            $AddLinearPiece(A_{sd}(t), f_r, PX_2, LST_2)$ ;   /* Adds the linear piece between
            the last intersection point and right V-point. */
        end
        $\{LST_1, EAT_1, LF_1, RF_1\} \leftarrow \{LST_2, EAT_2, LF_2, RF_2\}$
    end
    if $EAT_1 \neq \infty$ then  $AddLinearPiece(A_{sd}(t), RF_1, LST_1, \infty)$
    return $(A_{sd}(t))$
end
```

Algorithm 1: $\mathcal{TDSP}_{\text{lin}}(G, V, E, s, d)$

that RF and LF are the linear pieces to the right of v_l and to the left of v_r, respectively. Either RF and LF overlap or they intersect. If they overlap, the linear piece on RF (or LF) from x_l to x_r is part of the solution function since no other V-points are possible between v_l and v_r. On the other hand, if the two functions intersect in some point $I = (x_I, y_I)$ and the intersection is on $A_{sd}(t)$, then the linear piece on RF from x_l to x_I is on $A_{sd}(t)$ since no other V-points are possible between v_l and v_r. If I is not on $A_{sd}(t)$, then there must be another linear piece preventing it from being on the solution. The algorithm determines such a piece with maximum slope. The extension of the linear piece must intersect both RF and LF since otherwise there must be another V-point between v_l and v_r. Let I_l and I_r be the two intersection points. The algorithm now recursively performs what has been done for I, first for I_l and for then I_r. Starting from v_l, it adds linear pieces to the solution function once I_l is found to be on $A_{sd}(t)$. Then, it moves to the next intersection. As a last step, the algorithm adds to the solution function the last piece on LF between the last intersection and x_r. Since every X-point is verified to be on $A_{sd}(t)$ and no more V-points are possible between two consecutive V-points, the algorithm finds all X-points. Since V-points and X-points are the only breakpoints on $A_{sd}(t)$, algorithm $\mathcal{TDSP}_{\text{lin}}$ correctly finds all linear pieces of $A_{sd}(t)$. □

Theorem 2 *The time complexity of algorithm $\mathcal{TDSP}_{\text{lin}}$ (Algorithm 1) is $O((F_d + \gamma)(|E| + |V| \log |V|))$.*

Proof The algorithm first executes a slightly modified version of both standard and reverse Dijkstra's shortest path algorithm for each breakpoint in every link arrival time function in order to find all possible V-points. It then executes another modified version of Dijkstra's algorithm to find the greatest and smallest slope pieces close to each V-point. For every link (v, w) and starting time at v we compute the arrival time at w in $O(1)$ time using an *amortized analysis*. This follows from the fact that breakpoints are sorted in time and we can therefore execute Dijkstra's shortest path algorithm incrementally. For a given starting time at s, the earliest arrival time at d is computed in the same time as Dijkstra's algorithm, that is $O(|E| + |V| \log |V|)$. Consequently, the time-complexity for computing all V-points is $O(\gamma(|E| + |V| \log |V|))$ where γ is the total number of linear pieces in all link arrival time functions. Then, for computing all X-points, the algorithm executes a modified Dijkstra's algorithm as many time as we find intersection points. At each intersection point found, we determine the linear piece with greatest slope that hides the intersection point. This guarantees that every time we run a modified Dijkstra's algorithm at an intersection point we obtain a new linear piece that is part of the solution $A_{sd}(t)$. As a result, we will execute the modified Dijkstra's algorithm at most as many times as there are X-points on $A_{sd}(t)$. With F_d defined as the number of linear pieces on $A_{sd}(t)$, computing all X-points requires time $O(F_d(|E| + |V| \log |V|))$. Hence, the total time complexity of Algorithm $\mathcal{TDSP}_{\text{lin}}$ is $O(F_d + \gamma)(|E| + |V| \log |V|))$. □

3 The Shortest Paths Problem on Time-Dependent Networks with Availability Intervals ($\mathcal{TDSP}_{\text{int}}$)

3.1 A Reduction from $\mathcal{TDSP}_{\text{int}}$ to $\mathcal{TDSP}_{\text{lin}}$

In the following, we discuss how every instance of a time-dependent network with availability intervals for the $\mathcal{TDSP}_{\text{int}}$ problem can be converted into an equivalent time-dependent network with piece-wise linear functions by replacing intervals and weights with arrival time functions for every link in the network.

Let w_e^m and $[l_e^m, r_e^m]$ be the m^{th} weight and interval on link $e = (i, j) \in E$ in a time-dependent network with intervals (see Fig. 6(a)). By definition of the $\mathcal{TDSP}_{\text{int}}$ problem, for all departure times t from i between l_e^m and r_e^m, the arrival time is $a_e(t) = t + w_e^m$. Additionally, for all times t not in any intervals for that link, the earliest departure time is the smallest available time greater than t, say l_e^s. In this case, the earliest arrival time at j is $a_e(t) = l_e^s + w_e^s$. Finally, for all times t greater than the closing time of the last interval of the link, ($[l_e^{M_e}, r_e^{M_e}]$), the link is not available and the earliest arrival time is ∞. Figure 6(b), (c) depicts the result of this conversion and piece-wise linear function $a_e(t)$ for a link $e = (i, j)$.

Note that the resulting time-dependent network is indeed a FIFO network since arrival time functions are non-decreasing. Additionally, for each link e, $a_e(t)$ is a continuous piece-wise linear function. Consequently, in the following we focus on algorithms that solve the $\mathcal{TDSP}_{\text{lin}}$ problem applied to the time-dependent network obtained by our conversion process.

3.2 Structural Properties

3.2.1 O or 1 Slopes on the Output Function

It follows from our conversion process discussed in Sect. 3.1 that all link arrival time functions are piece-wise linear functions with FIFO property. Moreover, all slopes are either 0 or 1 since for any point in time, either a link is available for use or it is unavailable. As described earlier, on a path p from a source node s to destination d, the earliest arrival time for all times t, denoted by $a_p(t)$, is obtained by a sequential composition of the link arrival time functions for all links of the path. Similarly, the earliest arrival time function from s to d for all times t, denoted by $A_{sd}(t)$, is the minimum of the arrival time functions on all paths from s to d. Both operations, composition and minimum, do not change the slopes to any values other than 0 or 1. Consequently, all linear pieces on both $a_p(t)$ and $A_{sd}(t)$ have slope either 0 or 1.

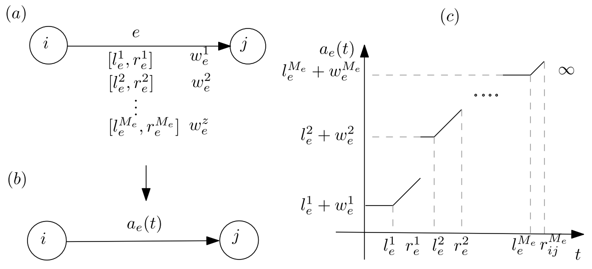


Fig. 6 (**a**) A link in the time-dependent network with intervals. (**b**) A converted link in the new time-dependent network with piece-wise linear functions. (**c**) The arrival time function on the converted link

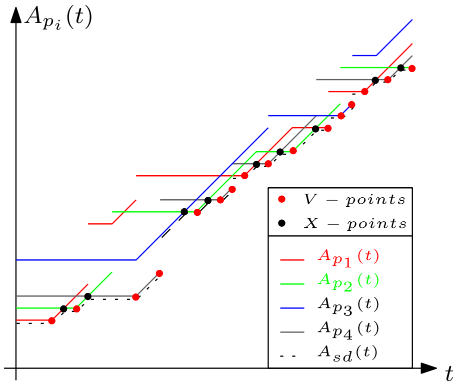


Fig. 7 An example showing V-points and X-points

3.2.2 $O(\kappa)$ Output Size

As discussed in Sect. 2.3, it follows from Lemma 1 that there are at most κ V-points on the solution function $A_{sd}(t)$ of the $\mathcal{TDSP}_{\text{int}}$ problem. Here, κ denotes the input size which is the total number of linear pieces in all link arrival time functions.

Lemma 2 *There are at most $O(\kappa)$ X-points on the solution function $A_{sd}(t)$.*

Proof An X-point is the intersection of slope 0 and slope 1 linear pieces. For two consecutive V-points on $A_{sd}(t)$ it follows from the fact that all slopes are 0 and 1 there cannot be more than one X-point between these two consecutive V-points. By Lemma 1, there are $O(\kappa)$ V-points on $A_{sd}(t)$. Hence, there are at most $O(\kappa)$ X-points on $A_{sd}(t)$. □

Figure 7 shows V-points and X-points on an arrangement of four path arrival time functions.

Theorem 3 *There are at most $O(\kappa)$ breakpoints on the solution function of the $\mathcal{TDSP}_{\text{int}}$ problem, $A_{sd}(t)$.*

Proof The result follows from Lemma 1 and Lemma 2. □

Figure 8 shows a time-dependent network with availability intervals. There are $\theta(|E|)$ breakpoints on $A_{sd}(t)$ for the case of $O(1)$ intervals on each link. Note that the example could be generalized to cases of multiple intervals (translates to κ_e linear pieces on $a_e(t)$ in the time-dependent network) if we build the i^{th} interval on a link by adding some constant value to the $(i-1)^{st}$ interval so that they do not overlap. This will result in a network with $\theta(\kappa) = \theta(\lambda)$ breakpoints on $A_{sd}(t)$, where λ is the total number of intervals in the network.

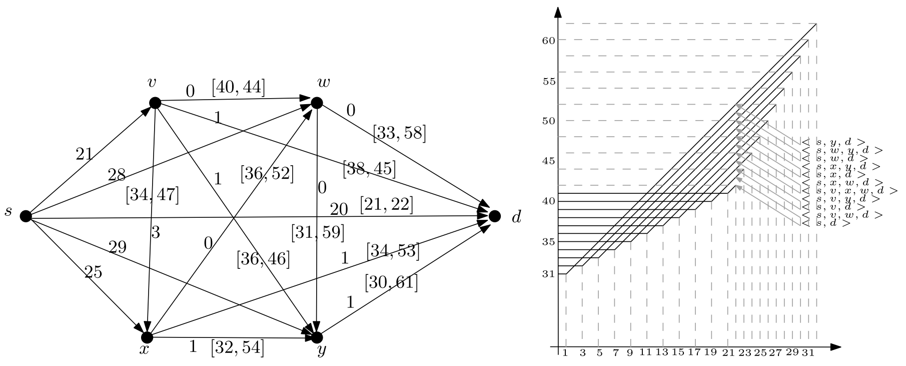


Fig. 8 An example of a network with $O(E)$ pieces on the EAT function to d

3.3 Solving the $\mathcal{TDSP}_{\text{lin}}$ Instances Created for Solving $\mathcal{TDSP}_{\text{int}}$ Problems

3.3.1 Applying Lower Envelope Algorithms

Suppose P is the set of all possible paths from s to d, and $a_p(t)$ is the arrival time function of path $p \in P$. Then, the earliest arrival time function from s to d for all times t, $A_{sd}(t) = \min_{p \in P}(a_p(t))$ is the lower envelope of all possible earliest arrival time functions. A naive algorithm for solving the $\mathcal{TDSP}_{\text{int}}$ problem could be to use well-studied lower envelope algorithms (see e.g. [1, 20]). Unfortunately, the number of such paths and therefore the number of arrival time functions could be exponential which makes this approach inefficient.

3.3.2 Applying Label-Correcting Algorithms

For general time-dependent networks for which the FIFO property holds and have piece-wise linear functions, Orda and Rom [18] proposed an algorithm that has a time bound of $O(F_{max}|V||E|)$, where F_{max} is the maximum number of pieces in a given output function $A_{si}(t)$ for source node s to any node i. As shown, in the $\mathcal{TDSP}_{\text{int}}$ problem there are at most $O(\lambda)$ pieces in the output function for each node which results in a total running time of $O(\lambda|E||V|)$.

3.3.3 Applying Label-Setting Algorithms

For time-dependent networks with FIFO property and piece-wise linear arrival time function on links, Dean [10] suggested a label-setting algorithm. This algorithm has a running time of $O(|E|F^* \log|V|)$ where F^* is the total number of pieces among all output functions. When applied to the $\mathcal{TDSP}_{\text{int}}$ problem, this approach results in a running time of $O(\lambda|E|\log|V|)$.

3.3.4 Applying our Improved $\mathcal{TDSP}_{\text{lin}}$ Algorithm to Instances Created for Solving $\mathcal{TDSP}_{\text{int}}$ Problems

In Sect. 2, we presented a new algorithm that solves the $\mathcal{TDSP}_{\text{lin}}$ problem in time $O((F_d + \lambda)(|E| + |V|\log|V|))$ where F_d is the output size (number of linear pieces on the earliest arrival time function to d). For instances created for solving $\mathcal{TDSP}_{\text{int}}$problems, the output size is $O(\lambda)$. Hence, the time complexity would be $O(\lambda(|E| + |V| \log |V|))$. Note that $\mathcal{TDSP}_{\text{lin}}$ considers only one destination node whereas $\mathcal{TDSP}_{\text{int}}$ requires shortest paths from s to *all* nodes in the network. Algorithm 2 computes, for a given source node s, the earliest arrival time functions to all nodes in the network within the same time-complexity $O(\lambda(|E| + |V| \log |V|))$.

```
begin
    for $i = 1$ to $|V|$ do $A[i] \leftarrow NULL$
    for every link $e = (v, w) \in E$ do
        for $i = 0$ to $\kappa_e$ do
            $LST \leftarrow ReverseDijkstra^*(v, s, T_e^i)$
            $EAT \leftarrow DijkstraAllDestinations^*(v, T_e^i)$
            $TMP \leftarrow DijkstraAllDestinations^*(s, LST)$
            $f_l \leftarrow LeftFunction(s, LST)$
            $f_r \leftarrow RightFunction(s, LST)$
            for $i = 1$ to $|V|$ do
                if $EAT[i] = TMP[i]$ then
                    $InsertToList(L[i], \{LST, EAT[i], f_l[i], f_r[i]\})$
                end
            end
        end
    end
    for $i = 1$ to $|V|$ do $Sort(L[i])$
    for $i = 1$ to $|V|$ do
        $\{LST_1, EAT_1, LF_1, RF_1\} \leftarrow RemoveItem(L[i])$
        while $NotEmpty(L[i])$ do
            $\{LST_2, EAT_2, LF_2, RF_2\} \leftarrow RemoveItem(L[i])$
            if $Overlap(RF_1, LF_2)$ then
                $AddLinearPiece(A[i](t), RF_1, LST_1, LST_2)$
            else
                $(IX, IY) \leftarrow IntersectionPoint(RF_1, LF_2)$
                $AddLinearPiece(A[i](t), RF_1, LST_1, IX)$
                $AddLinearPiece(A[i](t), LF_2, IX, LST_2)$
            end
            $\{LST_1, EAT_1, LF_1, RF_1\} \leftarrow \{LST_2, EAT_2, LF_2, RF_2\}$
        end
        if $EAT_1 \neq \infty$ then $AddLinearPiece(A[i](t), RF_1, LST_1, \infty)$
    end
    return $(A)$
end
```

Algorithm 2: $\mathcal{TDSP}_{\text{int}}(G, V, E, s)$

Function $A[i](t)$ represents the earliest arrival time function from s to every node i of the network. In Line 2 of Algorithm 2, this function has been initialized to ∞ for every node of the network. We will gradually build these functions by adding linear pieces to them. For all nodes i, Lines 3 through 12 find all V-points on $A[i](t)$ and add them to list $L[i]$ for later use. Here, *ReverseDijkstra*(v, s, T_e^i) is the same as in Algorithm 1. It returns the latest possible starting time from s to arrive at v at time T_e^i. Similarly, *DijkstraAllDestinations*(v, T_e^i) is a slightly modified version of the static shortest path algorithm from v to every node i starting at breakpoint time T_e^i. It returns the earliest possible arrival time at *all* nodes if one starts from v at time T_e^i. Note that the result of this function is an array holding the earliest arrival time to node i in its i^{th} position. For every node i, functions *LeftFunction*(s, LST) and *RightFunction*(s, LST) return the linear function on the left and on the right of the V-point that occurs at time LST. Line 11 checks whether a V-point is on the solution function or not, and adds it to the list accordingly. The algorithm builds $A[i](t)$ for each node i by adding linear pieces between every two consecutive V-points. We first sort the list of V-points for each node (Line 13). Then, in Lines 14 through 25, the algorithm adds one or two linear pieces depending on whether linear pieces between two consecutive V-points overlap or intersect, respectively.

As a further extension, consider the all pair version of $\mathcal{TDSP}_{\text{int}}$ where shortest paths are calculated between all pairs of nodes. We observe that the reverse shortest path algorithm reports at each breakpoint not only the latest start time from source node s, but also from all nodes in the network. As a result, we can compute the earliest arrival time function from all source nodes to all destinations using a slight modification of Algorithm 2. Thus, all $|V|^2$ earliest arrival time functions for the all pairs version of $\mathcal{TDSP}_{\text{int}}$ can be computed in time $O(\lambda|V|^2)$.

4 Conclusion

In this paper, we presented new algorithms for shortest path problems on two types of time-dependent networks with FIFO property: networks where edges have time-dependent availability intervals ($\mathcal{TDSP}_{\text{int}}$), and networks where edges have piece-wise linear arrival time functions ($\mathcal{TDSP}_{\text{lin}}$). We solved the $\mathcal{TDSP}_{\text{int}}$ problem by reducing to a special case of the second problem, $\mathcal{TDSP}_{\text{lin}}$ for which we presented a novel solution based on new, non-trivial, combinatorial properties of arrival time functions. These new insights allow us to focus on finding the earliest arrival time function for the destination node only, and only at crucial time-points. This way, we can discard unnecessary computations. In contrast to previous methods, our algorithms directly compute the earliest arrival time function for every destination node d by tracing time and finding the earliest arrival time only at time instances that may change the earliest arrival time function for d. The algorithms presented in this paper improve significantly upon the previously known methods for the $\mathcal{TDSP}_{\text{int}}$ and $\mathcal{TDSP}_{\text{lin}}$ problems.

Both of our methods make extensive use of static shortest paths algorithms. One may use more efficient static shortest path algorithms for further improvement in special cases. For example, in planar networks, applying linear time shortest path algorithms will further improve our results. In many practical networks, heuristics such as A^* could also be applied to improve the practical performance of our methods.

Acknowledgements We would like to thank Lyudmil Aleksandrov, Rolf Klein, and Florian Berger for their helpful discussions, and Hassan Abbas for help with the implementation work.

References

1. Agarwal, P.K., Schwarzkopf, O., Sharir, M.: The overlay of lower envelopes and its applications. Discrete Comput. Geom. **15**(1), 1–13 (1996)
2. Ahuja, R.K., Orlin, J.B., Pallottino, S., Scutellà, M.G.: Minimum time and minimum cost-path problems in street networks with periodic traffic lights. Transp. Sci. **36**(3), 326–336 (2002)
3. Bertsekas, D.P.: A simple and fast label correcting algorithm for shortest paths. Networks **23**(7), 703–709 (1993)

4. Brodal, G.S., Jacob, R.: Time-dependent networks as models to achieve fast exact time-table queries. Electron. Notes Theor. Comput. Sci. **92**, 3–15 (2004)
5. Chon, H.D., Agrawal, D., El Abbadi, A.: Fates: finding a time dependent shortest path. In: MDM '03: Proceedings of the 4th International Conference on Mobile Data Management, pp. 165–180. Springer, London (2003)
6. Cooke, K.L., Halsey, E.: The shortest route through a network with time-dependent internodal transit times. J. Math. Anal. Appl. **14**(3), 493–498 (1966)
7. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, New York (2001)
8. Daganzo, C.F.: Reversibility of the time-dependent shortest path problem. Transp. Res., Part B, Methodol. **36**(7), 665–668 (2002)
9. Dean, B.C.: Continuous-time dynamic shortest path algorithms. Master's thesis, MIT Department of Computer Science (1999)
10. Dean, B.C.: Shortest paths in FIFO time-dependent networks: theory and algorithms. Technical report, MIT Department of Computer Science (2004)
11. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
12. Ding, B., Yu, J.X., Qin, L.: Finding time-dependent shortest paths over large graphs. In *EDBT*, pp. 205–216 (2008)
13. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
14. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
15. Henzinger, M.R., Klein, P.N., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. J. Comput. Syst. Sci. **55**(1), 3–23 (1997)
16. Kanoulas, E., Du, Y., Xia, T., Zhang, D.: Finding fastest paths on a road network with speed patterns. In: ICDE '06: Proceedings of the 22nd International Conference on Data Engineering, p. 10. IEEE Computer Society, Washington (2006)
17. Nachtigall, K.: Time depending shortest-path problems with applications to railway networks. Eur. J. Oper. Res. **83**(1), 154–166 (1995)
18. Orda, A., Rom, R.: Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. J. ACM **37**(3), 607–625 (1990)
19. Orda, A., Rom, R.: Minimum weight paths in time-dependent networks. Networks **21**, 295–319 (1991)
20. Sharir, M., Agarwal, P.K.: Davenport-Schinzel Sequences and Their Geometric Applications. Cambridge University Press, New York (1996)
21. Sung, K., Bell, M.G.H., Seong, M., Park, S.: Shortest paths in a network with time-dependent flow speeds. Eur. J. Oper. Res. **121**(1), 32–39 (2000)
22. Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. J. ACM **46**(3), 362–394 (1999)