Foremost non-stop journey arrival in linear time

Juan Villacis-Llobet, Binh-Minh Bui-Xuan, Maria Potop-Butucaru

Foremost non-stop journey arrival in linear time

Juan Villacis-Llobet[1,2], Binh-Minh Bui-Xuan[1], and Maria Potop-Butucaru[1]

[1] LIP6 (CNRS – Sorbonne Université), [buixuan,maria.potop-butucaru]@lip6.fr
[2] Institut Polytechnique de Paris, juan.villacisllobet@ip-paris.fr

Abstract. A journey in a temporal graph is a sequence of adjacent and dated edges preserving the increasing order of arrival dates to the consecutive edges. When a journey never visits a vertex twice it is also called a temporal path. Given a pair of source and target vertices, a journey connecting them is foremost if the arrival date at the target vertex is the earliest. Like in the static case, there always exists a foremost journey which is also a temporal path because it is useless to circle around an intermediary vertex. It is therefore equivalent to compute the arrival date of a foremost journey or a foremost temporal path.
A non-stop journey is a journey where every pair of consecutive edges must also fulfill a maximum waiting time constraint. Foremost non-stop journeys can achieve strictly earlier arrival date than foremost non-stop temporal paths. We present a linear time algorithm computing the earliest arrival date of such a non-stop journey connecting any two given vertices in a given temporal graph.

Keywords: temporal graph, foremost journey, non-stop journey.

1 Introduction

In a static graph, both ShortestWalk and ShortestPath ask for the same computation, that is, a path joining two given vertices with the least number of edges. There is no need to make a distinction between walks and paths because a shortest walk never visits a vertex twice, hence, is also a path. Moreover, shortest paths fulfill a very convenient local optimisation property called prefix preservation: any prefix of a shortest path is itself a shortest path. Exploiting prefix preservation, popular greedy algorithms such as Dijkstra or Bellman-Ford algorithms can be used to compute shortest path in polynomial time [4].

Generalising to the temporal case, given a temporal graph whose edges are weighted with cost function c and two vertices s and t, a *journey* from s to t is a sequence of dated edges $(d_1, s = v_1, v_2), (d_2, v_2, v_3), \ldots, (d_p, v_p, v_{p+1} = t)$ satisfying some condition of *realizability* over the dates d_i's. Furthermore, when a journey never visits a vertex twice, it is called a *temporal path*. A fundamental realizability constraint we will impose on all journeys appearing in this paper is being *timely increasing*, that is, $d_i + c(d_i, v_i, v_{i+1}) \leq d_{i+1}$ for every $1 < i \leq p$. Finding the earliest arrival date at destination $d_p + c(d_p, v_p, v_{p+1} = t)$ of a journey, resp. temporal path, satisfying the timely increasing property, or outputting

a negative answer when such a date does not exist, is called the ForemostJourneyArrival, resp. ForemostTemporalPathArrival, problem. This helps modelling both ground traffic [5] and TCP/IP transmission [10], where a vehicle or a TCP/IP package need to be at successive checkpoints in increasing arrival dates. Like in the static case, there is no need here to make a distinction between journeys and temporal paths because removing from any foremost journey a cycle around an intermediary vertex does not modify the arrival date at its final destination. Furthermore, prefix preservation can also be retrieved for such journeys after a topological sort over the vertices [1]. From that point, the Dijkstra or Bellman-Ford approach can be extended to compute a foremost journey in polynomial time. Prefix preservation also plays a crucial role in obtaining algorithmic solutions for other path problems in temporal graphs [2,9].

In addition to the timely increasing property, a stronger realizability condition is to also have *non-stop transit*, that is, we also have the inequality the other way around $d_{i+1} \leq d_i + c(d_i, v_i, v_{i+1})$ for every $1 < i \leq p$. Here, $c(d_i, v_i, v_{i+1})$ represents exactly the time it takes for sailing from one vertex to another, where the journey *must continue* without delays. This helps dealing with physical constraints when the traversal is performed by an aircraft or a boat [6]: while a TCP/IP package can be retained at a vertex for an unlimited delay, an aircraft can not perform a stationary flight at a vertex waiting for a better wind condition. In the present paper, we address the slightly more general condition of (α, β)-*transit* which allows for d_{i+1} to depart within a time window, that is, $d_i + c(d_i, v_i, v_{i+1}) + \alpha(v_{i+1}) \leq d_{i+1} \leq d_i + c(d_i, v_i, v_{i+1}) + \beta(v_{i+1})$ for every $1 < i \leq p$. This helps modeling disease spreading where an infection is supposed not to stay on any infected individual v_{i+1} for more than $\beta(v_{i+1}) = 7$ days. With α being constantly equal to 0 and β constantly equal to 7 days, solving the corresponding ForemostJourneyArrival under (α, β)-transit would let us know if the destination vertex would be at risk of contamination whenever the source vertex is infected.

On the theoretical side, not only ForemostJourneyArrival and ForemostTemporalPathArrival strictly differ under (α, β)-transit, but it is also unclear how to retrieve the prefix preservation property, as the topological sort approach does not seem to give satisfying results. It is even more unfortunate that ForemostTemporalPathArrival under (α, β)-transit is NP-hard [3]. The situation is better for ForemostJourneyArrival, where to the best of our knowledge, it can be solved in $O(n+m \log m)$ time under (α, β)-transit, with n being the number of vertices and m the number of dated edges in the input temporal graph following a recent result in Ref. [8]. Therein, the main idea is to slice the input temporal graph into graphs G_d containing arcs of the input temporal graph dated with d, plus some well selected arcs with arrival date equal to d. Then, by sliding the value of d over the time dimension, one can decompose the original problem into a computation of journeys ending before d and what will be remaining. Using a Dijkstra approach to solve the former part, one can achieve a global $O(n + m \log m)$ time solution for ForemostJourneyArrival under (α, β)-transit, along with a larger class of path-like problems [8].

We present a linear time solution for ForemostJourneyArrival under (α, β)-transit. Unlike the previous decomposition approach, we focus in reducing a set of relevant arcs into one arc which achieves the desired foremost journey arrival date. It is divided into four stages which will be called sequentially. In a nutshell, we first compute a representation for the arc set of a large gadget digraph (first two stages), then traverse it (third stage) before a last scan to filter the output (fourth stage). Considering implementation matters, we devise the first two stages and the fourth stage using the filter-map-reduce programming paradigm. Additionally, these stages can easily be batch-performed in a distributed setting. Our third stage computation is a graph traversal and it is very unclear whether this stage can be parallelized. Nevertheless, we leave open the question whether the third stage can also be implemented using functional programming, which would be interesting among other things for reuse matters.

In order to achieve linear time complexity, we cope with the gadget digraph using an implicit representation. Originally, each vertex of the gadget graph is a pair (d, u) denoting that it is possible in the input temporal graph to leave u at d (to any destination). There is an arc from (d, u) to (d', v) if it is possible to leave u at d to arrive to v, *and* leave again v at d' while fulfilling the (α, β)-transit condition at v. We show that the gadget graph allows for encoding every information we need to solve ForemostJourneyArrival. However, its size is very large. For every fixed v, we then exploit the total order of the time dimension, and regroup only relevant values of d' into disjoint-sets using a restricted version of disjoint-set data structure [7]. Finally, we show a constant upper bound for the out-going degree of the leftover implicit representation of the gadget graph, and use it to prove the global linear time complexity.

Our paper is organised as follows. We formalise in Section 2 problem ForemostJourneyArrival under (α, β)-transit. In Section 3 we present the main structures to be computed before solving this problem. We show in Section 4 how to compute all these structures in linear time. In Section 5 we close the paper with concluding remarks and open perspectives for further research.

2 Journey in a temporal (di)graph

In this paper, digraphs are simple loopless directed graphs. This encompasses the case of simple loopless undirected graphs, whose formalism is equivalent to that of symmetric digraphs. We denote $V \otimes V = V \times V \setminus \{(v, v) : v \in V\}$ for any finite set V. A *temporal digraph* is a tuple $G = (\tau, V, A, c)$ where:

- $\tau \in \mathbb{N}$ is an integer called the *timespan* of G. We define interval $T = [\![0, \tau - 1]\!]$ as the set of time instants used in G.
- V is a finite set called the *vertex set* of G.
- $A \subseteq T \times V \otimes V$ is called the *arc set* of G.
- $c : A \to \mathbb{N}$ represents the traversal time of every arc, it is called the *cost function* for G.

For every arc $a = (d, s, t) \in A$, we denote $s(a) = s$ the *source vertex* of the arc, $t(a) = t$ its *target vertex*, and $d(a) = d$ its *departure time*. The traversal of

arc a departs from s towards t at departure time d and arrives to t at arrival time $d + c(a)$.

Remark 1. With this formalism, if $a = (d, s, t)$ belongs to A and $c(a) > 1$, it is still not necessarily the case that $a' = (d + 1, s, t)$ belongs to A. If both a and a' belong to A, the formalism allows for $c(a)$ and $c(a')$ to differ arbitrarily. This helps modeling the routing condition from s to t according to the moment the arc is traversed.

We define journeys with waiting time constraints following the formalism given in [8]. Let $s, t \in V$ be two distinct vertices of G. Let $\alpha, \beta : V \to \mathbb{N}$ be two functions representing the minimum and maximum waiting time at every vertex. An (α, β)-*journey* from s to t is a sequence of arcs $J = (a_1, a_2, \ldots, a_p) \in A^p$, where $s(a_1) = s$, $t(a_p) = t$, and for every $1 \leq i < p$ we have both $t(a_i) = s(a_{i+1})$ and $d(a_i) + c(a_i) + \alpha(t(a_i)) \leq d(a_{i+1}) \leq d(a_i) + c(a_i) + \beta(t(a_i))$. For $1 \leq i < p$, the traversal of arc a_i begins from source vertex $s(a_i)$ at departure time $d(a_i)$, it takes $c(a_i)$ time steps to arrive at target vertex $t(a_i)$, where the journey has to be delayed for at least $\alpha(t(a_i))$ and at most $\beta(t(a_i))$ time steps before pursuing with the traversal of arc a_{i+1}. The *arrival date* of J is defined as $d(a_p) + c(a_p)$. A journey is called *foremost* when its arrival date is minimum.

On input a temporal graph $G = (\tau, V, A, c)$ with transit functions (α, β) and two vertices s, t in G, the problem of computing the minimum value of arrival date $d(a_p) + c(a_p)$ taken over every (α, β)-journey $J = (a_1, a_2, \ldots, a_p)$ from s to t is called the ForemostJourneyArrival under (α, β)-transit problem. When both α and β are constantly equal to 0, such a $(0, 0)$-journey J is called a *non-stop* journey. Figure 1 exemplifies such journeys.

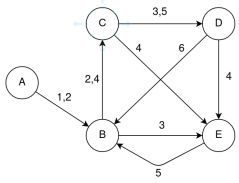

Fig. 1. A temporal digraph, the labels on the arcs denote the time instants where the arcs are active. If the cost function is uniformly unitary, then the following journey from A to E is foremost: $A \xrightarrow{1} B \xrightarrow{3} E$, while the journey $A \xrightarrow{1} B \xrightarrow{2} C \xrightarrow{3} D \xrightarrow{4} E$ is the foremost $(0, 0)$-journey, also known as non-stop.

3 From journey to time set

In this section we prepare the way for solving ForemostJourneyArrival under (α, β)-transit. Lemma 1 below is crucial because it helps us reduce a path-like problem down to a set problem. Then, Lemmas 2 and 4 introduce the structures

that will be used in next Section 4 to solve ForemostJourneyArrival under (α, β)-transit. For the sake of clarity, the time complexity which will be given in both Lemmas 2 and 4 is inefficient. In the next section, we depict optimised replacements for these steps, reducing the time complexity down to linear.

Let $G = (\tau, V, A, c)$ be a temporal digraph. Let $\alpha, \beta : V \to \mathbb{N}$ be two functions representing the minimum and maximum waiting time constraints. For any source vertex $s \in V$, we define the set of (α, β)-*reachable arcs* from s as $R(s) = \{a_p \in A : \exists\ (\alpha, \beta)\text{-journey } J = (a_1, a_2, \ldots, a_p) \in A^p \wedge s(a_1) = s\}$. We show below how to compute, from the supposed knowledge of set $R(s)$, the minimum arrival date of an (α, β)-journey from s to any target vertex t.

Lemma 1 (Reachable arcs). *On input a temporal digraph G, a pair of source and target vertices s, t in G, two constraint functions $\alpha, \beta : V \to \mathbb{N}$, and the above defined (α, β)-reachable arc set $R(s)$, it is possible to output in linear time the minimum arrival date of an (α, β)-journey from s to t. More precisely, the minimum arrival date is equal to $\min\{d(a) + c(a) : a \in R(s) \wedge t(a) = t\}$, which can be reduced from $R(s)$ in linear time. Moreover, the reduction can be done using a functional programming approach as showed in Remark 2 below.*

Proof. We first address the case when there exists an (α, β)-journey from s to t. Let mad be the minimum arrival date of such a journey. Let $m = \min\{d(a) + c(a) : a \in R(s) \wedge t(a) = t\}$. We claim that $m = mad$.

Indeed, let $J = (a_1, a_2, \ldots, a_p) \in A^p$ be an (α, β)-journey from s to t minimising the arrival date. By definition, we firstly have that mad is the arrival date of J, that is, $mad = d(a_p) + c(a_p)$. Besides, it also follows from definition of J that $t(a_p) = t$. Moreover, J is also such that $s(a_1) = s$, therefore, we have from definition of $R(s)$ that $a_p \in R(s)$. Now, we have both $a_p \in R(s) \wedge t(a_p) = t$, therefore, by definition of m we have that $m \leq d(a_p) + c(a_p)$. Combining with earlier proven $mad = d(a_p) + c(a_p)$ we obtain $m \leq mad$.

Conversely, let $a \in R(s)$ such that $t(a) = t$. We claim that $mad \leq d(a) + c(a)$. By definition of $a \in R(s)$, there exists an (α, β)-journey $J = (a_1, a_2, \ldots, a_p) \in A^p$ such that $s(a_1) = s$ and $a_p = a$. Combining with $t(a) = t$ we have that J is exactly an (α, β)-journey from s to t. Since the arrival date of J is $d(a) + c(a)$, we have that $mad \leq d(a) + c(a)$ because mad is the minimal arrival date taken over all (α, β)-journeys from s to t. We have proven that $mad \leq d(a) + c(a)$ for every $a \in R(s) \wedge t(a) = t$. Hence, $mad \leq m$.

All in all we have just proved that $m = mad$. Therefore it is sufficient to compute m in order to output the value of mad. Finally, computing m from the input of $R(s)$, vertex t, and cost function c can be done in linear time by any standard streaming process.

When there is no (α, β)-journey from s to t then $mad = \infty$. In this case $R(s)$ contains no arc a such that $t(a) = t$ and therefore m would have the value ∞ after performing the map-reduce (as no arc satisfies both properties needed to be considered a suitable value for m). Therefore the result still holds for this particular case. □

Remark 2. Lemma 1 is proper to temporal digraphs in the sense that we can from input $G = (\tau, V, A, c)$ filter the set A to a smaller subset $R(s) \subseteq A$, then filter further to the set of arcs whose target vertex is t, and finally reduce the stream to find the minimum value $d(a) + c(a)$. As a comparison no shortest path algorithm on static (di)graphs allows for using filter-map-reduce programming in such a straightforward manner.

We now introduce an intermediary step to compute the arc set $R(s)$. We define the set of *valid transit departures* in an (α, β)-journey from s as $D(s) = \{(d, v) \in T \times V : \exists\ (\alpha, \beta)\text{-journey } J = (a_1, a_2, \ldots, a_p) \in A^p,\ \text{such that } s(a_1) = s \wedge s(a_p) = v \wedge d(a_p) = d\}$.

Lemma 2 (Valid transit departures). *On input a temporal digraph G, a pair of source and target vertices s, t in G, two constraint functions $\alpha, \beta : V \to \mathbb{N}$, and the above defined set $D(s)$ of valid transit departures in an (α, β)-journey from s, it is possible to output in polynomial time the set $R(s)$ of (α, β)-reachable arcs from s.*

Proof. A naive way to output $R(s)$ from $D(s)$ is as follows. We initialize a boolean table `R` indexed by the elements of A. For any $a \in A$ with $a = (d, u, v)$, we scan $D(s)$ and check if $(d, u) \in D(s)$. If this is the case we set `R[a]` to `true`. At the end of the process, we scan `R` and output every index a where `R[a]` has value `true`. □

In the sequel we show how to compute in polynomial time the set $D(s)$ from the input of G. We first define a static digraph associated to temporal digraph G, then we perform a graph search on the thus defined static graph.

The (α, β)-*transit departure digraph* of G, that we call $G_D = (V_D, A_D)$, is defined as follows. First, $V_D = \{(d, v) : \exists a \in A, s(a) = v \wedge d(a) = d\}$ is the set of all possible transit departures, including those not necessarily valid w.r.t. any (α, β)-journey from s. In other words, $D(s) \subseteq V_D$, however, V_D could be much larger than $D(s)$. Then, for any pair of vertices $x = (d, u)$ and $y = (d', v)$ of V_D, we define $(x, y) \in A_D$ if and only if we have both that $a = (d, u, v)$ belongs to A and that $d + c(a) + \alpha(v) \leq d' \leq d + c(a) + \beta(v)$. Figure 2 exemplifies the construction of a transit departure digraph.

We capture in the following property our main computational purposes of defining G_D. It gives a reasonable upper bound for both $|V_D|$ and $|A_D|$.

Property 1. Let $G = (\tau, V, A, c)$ be a temporal digraph, $\alpha, \beta : V \to \mathbb{N}$ two functions representing the minimum and maximum waiting time constraints, and $G_D = (V_D, A_D)$ the (α, β)-transit departure digraph of G. Let $\gamma = \max\{\beta(v) - \alpha(v) + 1 : v \in V\}$. Then, $|V_D| \leq |A|$ and $|A_D| \leq \gamma \times |A|$.

Proof. Note that two naive upper bounds for G_D exist: $|V_D| \leq \tau \times |V|$ and $|A_D| = O(|V_D|^2)$. Furthermore, we can also note by definition $V_D = \{(d, v) : \exists a \in A, s(a) = v \wedge d(a) = d\}$ that $|V_D| \leq |A|$ because there will be at most one vertex in V_D for every arc in A. As a side note, it could be the case that

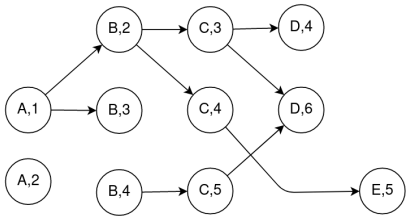

Fig. 2. The corresponding static representation G_D of the temporal graph in Figure 1 taking 0 as minimum waiting time, 2 as the maximum waiting time and a traversal time of 1 for all arcs.

$|V_D| < |A|$: if we have $(d, v, w) \in A$ and $(d, v, w') \in A$ for distinct vertices $w \neq w'$.

Now, let us examine $(x, y) \in A_D$ with $x = (d, u)$ and $y = (d', v)$. By definition, $a = (d, u, v)$ must belong to A, and d' must satisfy the waiting time constraints $d+c(a)+\alpha(v) \leq d' \leq d+c(a)+\beta(v)$. Let us define function $f : A_D \to A \times [\![0, \gamma-1]\!]$ as $f((d, u), (d', v)) = (d, u, v, d' - d - c(a))$. Then, we can check that f is a well-defined injective function, and therefore, deduce that $|A_D| \leq \gamma \times |A|$. □

Essentially, the size of thus defined graph G_D is not far from linear in $|A|$. We now would like to extract from V_D all vertices belonging to $D(s)$. By definition, we have the following closure property: $(x, y) \in A_D \wedge x \in D(s)$ implies $y \in D(s)$. Hence, $D(s)$ encompasses the set of vertices in G_D reachable from any vertex of the set $V_D \cap \{(d, s) : 0 \leq d < \tau\}$. Moreover, we show in the following lemma that $D(s)$ is exactly the latter set.

Lemma 3. *Let $G = (\tau, V, A, c)$ be a temporal digraph, $\alpha, \beta : V \to \mathbb{N}$ two functions representing the minimum and maximum waiting time constraints, $s \in V$, and $D(s)$ the set of valid transit departures in an (α, β)-journey from s. Let $G_D = (V_D, A_D)$ be the (α, β)-transit departure digraph of G. Then, $D(s)$ is exactly the set of vertices in G_D which are reachable from a (directed) path beginning at any vertex of the set $V_D \cap \{(d, s) : 0 \leq d < \tau\}$.*

Proof. We denote by $R_D(s)$ the set of vertices in G_D which are reachable from a (directed) path beginning at any vertex of the set $V_D \cap \{(d, s) : 0 \leq d < \tau\}$. By definition of G_D, we have the following closure property: if $(x, y) \in A_D$ and $x \in D(s)$ then $y \in D(s)$. Besides, whenever $(d, s) \in V_D$ for any $0 \leq d < \tau$, that is, whenever there exists $a \in A$ such that $s(a) = s$ and $d(a) = d$, we also have that $(d, s) \in D(s)$ by using the single-arc (α, β)-journey $J = (a)$ in the definition of $D(s)$. Now, we use the above mentioned closure property in order to deduce that $R_D(s) \subseteq D(s)$. Hence, the only thing left for us to show is that $D(s) \subseteq R_D(s)$.

Let $(d, v) \in D(s)$. We would like to prove that $(d, v) \in R_D(s)$. By definition of $D(s)$, there exists an (α, β)-journey $J = (a_1, a_2, \ldots, a_p) \in A^p$ such that $s(a_1) = s$, $s(a_p) = v$, and $d(a_p) = d$. Let us consider $J_q = (a_1, a_2, \ldots, a_q)$, for any

$1 \leq q \leq p$. For convenience, we denote $d_q = d(a_q)$, $v_q = s(a_q)$, and $x_q = (d_q, v_q)$. Since $a_q \in A$ we have from definition of V_D that $x_q \in V_D$, for any $1 \leq q \leq p$. We claim that $(x_1, x_2, \ldots, x_p)$ is a directed walk in the static digraph G_D, with $x_1 \in V_D \cap \{(d, s) : 0 \leq d < \tau\}$ and $x_p = (d, v)$.

Indeed, by definition of $D(s)$ we have for any $1 \leq q \leq p$ that $(d_q, v_q) \in D(s)$. When $q = 1$, this implies $v_1 = s$, and therefore $x_1 = (d_1, v_1) = (d_1, s)$ belongs to $V_D \cap \{(d, s) : 0 \leq d < \tau\}$. Since the original J is an (α, β)-journey, it must satisfy the waiting time constraints, that is, we have $d_q + c(a_q) + \alpha(t(a_q)) \leq d_{q+1} \leq d_q + c(a_q) + \beta(t(a_q))$, for every $1 \leq q < p$. Besides, since $t(a_q) = s(a_{q+1}) = v_{q+1}$, we have both $(d_q, v_q, v_{q+1}) = a_q \in A$ and $d_q + c(a_q) + \alpha(v_{q+1}) \leq d_{q+1} \leq d_q + c(a_q) + \beta(v_{q+1})$. This implies (x_q, x_{q+1}) belongs to A_D for every $1 \leq q < p$. In other words, $(x_1, x_2, \ldots, x_p)$ is a directed walk in G_D. Since $d_p = d(a_p) = d$ and $s_p = s(a_p) = v$, we also have $x_p = (d, v)$. We have shown a directed walk in G_D beginning from vertex $x_1 \in V_D \cap \{(d, s) : 0 \leq d < \tau\}$, and ending at vertex $x_p = (d, v)$. This also implies there exists a directed path in G_D from x_1 to $x_p = (d, v)$. Hence, $(d, v) \in R_D(s)$. We have proved for every $(d, v) \in D(s)$ that $(d, v) \in R_D(s)$. In other words, $D(s) \subseteq R_D(s)$. □

Lemma 4. *On input a temporal digraph G, a source vertex s in G, two constraint functions $\alpha, \beta : V \to \mathbb{N}$, it is possible to output in polynomial time the set $D(s)$ of valid transit departures in an (α, β)-journey from s.*

Proof. Let $G = (\tau, V, A, c)$, and $G_D = (V_D, A_D)$ its (α, β)-transit departure digraph. By Lemma 3, $D(s)$ can be computed by a graph search on G_D, that is, in $O(|V_D| + |A_D|)$ time from the knowledge of G_D. From Property 1, the size of V_D and A_D is polynomial in $|A|$, α, and β. Hence, it is straightforward to construct V_D in $O(\tau \times |V| \times |A|)$, then A_D in $O(|V_D|^2 \times |A|)$. □

All in all, we presented in Lemmas 1, 2 and 4 polynomial procedures for computing the minimum arrival date of an (α, β)-journey from s to t. Whereas the procedure presented in Lemma 1 requires linear time, the other two might take more time to terminate. The total time complexity is significantly worse than the recently known $O(|V| + |A| \log |A|)$ algorithm presented in [8]. In the next section we present an improvement in the way we construct the graph G_D and traverse it in linear time in $|A|$.

4 Foremost non-stop journey arrival in linear time

In this section we show how to solve in linear time ForemostJourneyArrival under (α, β)-transit from a source vertex s to a target vertex t in a temporal digraph $G = (\tau, V, A, c)$. This encompass the case of foremost non-stop journeys when both α and β are constantly equal to 0. We do this by an implicit traversal of the (α, β)-transit departure digraph $G_D = (V_D, A_D)$ as defined in the previous section. We suppose the three functions c, α, β are given as tables, so that the cost for accessing $c(a)$ for every $a \in A$, and the cost for accessing $\alpha(v)$ and $\beta(v)$ for every $v \in V$ are constant.

Our algorithm is composed of four main stages, each one terminates in $O(\tau + |V| + |A|)$ time: first we construct V_D; then we construct a subset of A_D of representative arcs whose number is bounded by $2 \times |A|$; in a third stage we use the previously constructed structures to compute an implicit representation of the set $D(s)$ defined in the previous section; finally, we use this information and construct the set $R(s)$ defined in Lemma 1, and result as a byproduct in the earliest arrival date of an (α, β)-journey from s to t.

For use in Algorithm 1, we perform two linear bucket sorting processes (a.k.a. radix sorting) as follows. We first initialize two arrays containing τ buckets each. Each bucket is to contain a list of arcs initially empty. Then, we stream through A, where for every arc $a \in A$ we: first append a to the list present in the bucket numbered $d(a)$ in the first array of buckets; second append a to the list present in the bucket numbered $d(a) + c(a)$ in the second array of buckets. For later use, we also keep a variable counting the number of elements in every bucket. After the streaming process, we have filled two arrays of τ buckets each, where every bucket contains a list of arcs, as well as the number of arcs in the bucket. We now iterate over the buckets by increasing order and concatenate all the two times τ lists consecutively, resulting in a list of arcs sorted in increasing departure time from source vertex, and a list of arcs sorted in increasing arrival time to target vertex. Since the number of elements in each list is known, each concatenation can be done in constant time. The whole procedure is hence in $O(\tau + |A|)$.

From now on we suppose A is given twice: sorted by increasing departure time from source vertex, and sorted by increasing arrival time to target vertex.

Algorithm 1 Construction of V_D, the vertex set of G_D.

procedure GenerateVertices($G = (\tau, V, A, c), \alpha, \beta$)
  `Departures`$\leftarrow \emptyset$ ▷ Set containing all the new vertices in G_D
  Initialize table `VDep` with $|V|$ entries ▷ Same set, fast track for later use
  `Arrivals`$\leftarrow \emptyset$
  Initialize table `VArr` with $|V|$ entries
  for each vertex $v \in V$ **do**
    `VDep[`v`]`$\leftarrow \emptyset$
    `VArr[`v`]`$\leftarrow \emptyset$
  for each arc $a \in A$ in increasing departure time **do**
    Append $(d(a), s(a))$ to `Departures` if not already present
    Append $d(a)$ to `VDep[`$s(a)$`]` if not already present
  for each arc $a \in A$ in increasing arrival time **do**
    Append $(d(a) + c(a), t(a))$ to `Arrivals` if not already present
    Append $d(a) + c(a)$ to `VArr[`$t(a)$`]` if not already present
  Either output `VDep` or `Departures` as the vertex set V_D
  For later use in Algorithm 2, also output `VArr`.

Stage 1: Construction of V_D. We stream through every element $a \in A$ in increasing departure time and append $(d(a), s(a))$ to a `Departures` list. Simi-

larly, we stream through every element $a \in A$ in increasing arrival time and append $(d(a) + c(a), t(a))$ to an `Arrivals` list. By definition, the vertex set V_D contains exactly the elements present in the `Departures` list. Furthermore, we will organise V_D in the following manner, for later use in Stage 2. Let $V = \{v_1, v_2, \ldots, V_{|V|}\}$. We create $|V|$ buckets numbered by these v_i's, each bucket is to contain a list of departure times initially empty. When streaming through every element $a \in A$, we also append $d(a)$ to the list present in the bucket numbered $s(a)$. After the streaming process, we keep the $|V|$ lists in a table named `VDep`, indexed by the v_i's. Thus, for every vertex $v \in V$, reading `VDep[`v`]` gives a quick access to all the departure times d associated to that vertex, *i.e.* where $(d, v) \in V_D$. Since A is sorted by increasing departure time, it is also the case with list `Departures`, as well as with list `VDep[`v`]`, for every $v \in V$. For later use in Stage 2, we also organise `Arrivals` into a table named `VArr`, in a similar way. In reality, we do not use `Departures` and `Arrivals` in the rest of the manuscript. However, we keep them in the discussion for more clarity. We capture the pseudo-code in Algorithm 1, and result in the following lemma.

Lemma 5. *On input a temporal digraph $G = (\tau, V, A, c)$ and two constraint functions $\alpha, \beta : V \to \mathbb{N}$, Algorithm 1 correctly generates in time $O(|V| + |A|)$ all the vertices of G_D, the (α, β)-transit departure graph of G.*

Proof. The algorithm's correctness follows from definition. Lines 3,5,6-8 take $O(|V|)$ times while lines 9-14 take $O(|A|)$ time. □

Stage 2: Implicit representation of A_D in $O(|A|)$ space. If $\beta(v) - \alpha(v) = 1$ for every $v \in V$, then a similar argument as in the proof of Property 1 implies $|A_D| \leq 2 \times |A|$. Indeed, every arc $(d, u, v) = a \in A$ gives rise to at most two arcs in G_D: one from (d, u) to (d', v) with $d' = d + c(a) + \alpha(v)$ if the latter vertex (d', v) belongs to V_D; one from (d, u) to (d', v) with $d' = d + c(a) + \beta(v)$ if the latter vertex (d', v) belongs to V_D.

Now if $\beta(v) - \alpha(v)$ is an arbitrary integer, we remark the following organisation of A_D. Consider set $D' = \{d' : ((d, u), (d', v)) \in A_D\}$, then if both $d'_1 \leq d'_3$ belong to D' and d'_2 is such that we have both $d'_1 \leq d'_2 \leq d'_3$ and $(d'_2, v) \in V_D$, then d'_2 belong to D'. Moreover, in the (ordered) list `VDep[`v`]`, the elements of D' appear consecutively. Therefore, in order to represent all arcs of A_D of the form $((d, u), (d', v))$, for every given $(d, u, v) = a \in A$, we only need to store the arc from (d, u) to (d'_{min}, v) and the arc from (d, u) to (d'_{max}, v), where $d'_{min} = \min D'$ and $d'_{max} = \max D'$. All the other arcs of the form $((d, u), (d', v))$ can be obtained by enumerating from `VDep[`v`]` all d' such that $d'_{min} \leq d' \leq d'_{max}$.

In order to implement this idea, we define `DPmin` and `DPmax` to be two tables, indexed by the elements of A. For every $(d, u, v) = a \in A$, we define `DPmin[`(d, u, v)`]`$= \min\{d' : ((d, u), (d', v)) \in A_D\}$ and `DPmax[`(d, u, v)`]`$= \max\{d' : ((d, u), (d', v)) \in A_D\}$. Then, we will use in Stage 3 and Stage 4 the input of `VDep`, `DPmin`, and `DPmax` as an implicit representation of $G_D = (V_D, A_D)$.

In order to avoid computing `DPmin` and `DPmax` in quadratic time in $|A|$, our main trick is to break down the total transit time cost into two parts: the traversal time represented by function $c : A \to \mathbb{N}$, and the delay time represented

by functions $\alpha, \beta : V \to \mathbb{N}$. For this, we build auxiliary tables `FirstTransit` and `LastTransit` with the help of Algorithm 1 table `VArr`. We capture the pseudo-code for first computing the auxiliary tables, then `DPmin` and `DPmax` in Algorithm 2, and result in the following lemma.

Algorithm 2 Implicit representation of $G_D = (V_D, A_D)$ by `VDep`, `DPmin`, and `DPmax`.

procedure GenerateTables($G = (\tau, V, A, c), \alpha, \beta$)
  Call Algorithm 1 GenerateVertices(G, α, β) and obtain `VDep` and `VArr`
  Initialize table `FirstTransit` with $|V|$ entries
  Initialize table `LastTransit` with $|V|$ entries
  for each vertex $v \in V$ **do** ▷ Find for each arrival to v the first/last departure
    $d_{arr} \leftarrow$ first element of `VArr`$[v]$
    Initialize table `FirstTransit`$[v]$ with $|$`VArr`$[v]|$ entries
    $d'_{first} \leftarrow$ first element of `VDep`$[v]$
    while d_{arr} is still an element of `VArr`$[v]$ **do**
      while $\neg(d_{arr} + \alpha(v) \leq d'_{first})$ **do**
        $d'_{first} \leftarrow$ next element after d'_{first} in `VDep`$[v]$
      `FirstTransit`$[v]$`[`d_{arr}`]`$\leftarrow d'_{first}$
      $d_{arr} \leftarrow$ next element after d_{arr} in `VArr`$[v]$
    $d_{arr} \leftarrow$ last element of `VArr`$[v]$
    Initialize table `LastTransit`$[v]$ with $|$`VArr`$[v]|$ entries
    $d'_{last} \leftarrow$ last element of `VDep`$[v]$
    while d_{arr} is still an element of `VArr`$[v]$ **do**
      while $\neg(d'_{last} \leq d_{arr} + \beta(v))$ **do**
        $d'_{last} \leftarrow$ previous element before d'_{last} in `VDep`$[v]$
      `LastTransit`$[v]$`[`d_{arr}`]`$\leftarrow d'_{last}$
      $d_{arr} \leftarrow$ previous element before d_{arr} in `VArr`$[v]$
  Initialize table `DPmin` with $|A|$ entries
  Initialize table `DPmax` with $|A|$ entries
  for each arc $(d, u, v) = a \in A$ **do** ▷ Take c into account: $d_{arr} = d + c(a)$
    `DPmin`$[(d, u, v)]\leftarrow$ `FirstTransit`$[v]$`[`$d + c(a)$`]`
    `DPmax`$[(d, u, v)]\leftarrow$ `LastTransit`$[v]$`[`$d + c(a)$`]`
  Output `VDep`, `DPmin` and `DPmax` as implicit representation of G_D.

Lemma 6. *On input a temporal digraph $G = (\tau, V, A, c)$ and two constraint functions $\alpha, \beta : V \to \mathbb{N}$, Algorithm 2 outputs in time $O(|V| + |A|)$ three tables called* `VDep`*,* `DPmin`*, and* `DPmax`*. From these tables one can generate G_D, the (α, β)-transit departure graph of G, in linear time (in the size of the input G and the output G_D).*

Proof. It is a standard exercise to prove that Algorithm 2 correctly computes `DPmin`$[(d, u, v)] = \min\{d' : ((d, u), (d', v)) \in A_D\}$ and `DPmax`$[(d, u, v)] = \max\{d' : ((d, u), (d', v)) \in A_D\}$, *e.g.* by induction on the total size of `VArr`.

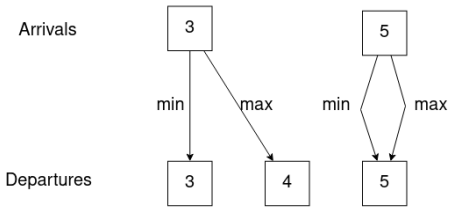

Fig. 3. Data structure used in Algorithm 2 for each vertex $v \in V$, where we suppose $\alpha(v) = 0$ and $\beta(v) = 2$. In this case it is done for the node c in the temporal graph of Figure 1. List `VArr[`v`]` is represented by the "Arrivals" row on top, it contains all times at which a path arrives to node c. List `VDep[`v`]` is represented by the "Departures" row on the bottom, it contains all times at which a path departs from node c. For every element d_{arr} in `VArr[`v`]`, its associated value `FirstTransit[`v`][`d_{arr}`]` in `VDep[`v`]` is pointed to by the *min* arrow departing from d_{arr}, while its associated value `LastTransit[`v`][`d_{arr}`]` in `VDep[`v`]` is pointed to by the *max* arrow.

For complexity issues, the main point lies in the chasing *while* loops which happens twice, once at lines 9-13, and the other time at lines 17-21. Here, vertex $v \in V$ is already fixed. In the first case, lines 9-13, variables d_{arr} and d'_{first} can only move forward in lists `VArr[`v`]` and `VDep[`v`]`, respectively. Accordingly, at the end of the process lines 9-13, every entry in `VArr[`v`]` and `VDep[`v`]` are visited exactly once. The case with lines 17-21 is similar, where variables d_{arr} and d'_{last} can only move backward. Figure 3 exemplifies for every entry of table `VArr[`v`][`d_{arr}`]` the position of `FirstTransit[`v`][`d_{arr}`]` and `LastTransit[`v`][`d_{arr}`]` in `VDep[`v`]`.

Summing up over every vertex $v \in V$, at the end of the process lines 5-21, every elements in `VArr` and `VDep` are visited exactly twice. Since the total size of `VArr` is at most $|A|$, and so is the total size of `VDep`, *cf.* Property 1, Lemma 5 and Algorithm 1, we deduce that the total contribution of lines 9-13 and 17-21 to the *for* loop lines 5-21 is $O(|A|)$. The contribution of lines 7 and 15 is $O(|A|)$, and that of lines 6,8,14,16 is $O(|V|)$. To complete the complexity analysis for Algorithm 2, we note that line 2 takes $O(|V| + |A|)$ time (Lemma 5), lines 3-4 take $O(|V|)$ time, and lines 23-26 take $O(|A|)$ times.

In order to generate V_D, we scan over all elements of `VDep`. The enumeration of A_D can proceed as follows. For every arc $(d, u, v) = a \in A$, we scan from the element `DPmin[`(d, u, v)`]` of `VDep[`v`]` to its element equal to `DPmax[`(d, u, v)`]` and enumerate all d' in between, generating an arc $((d, u), (d', v))$ for each of them. By definition of `DPmin` and `DPmax`, we have $A_D = \{((d, u), (d', v)) : (d, u, v) \in A \wedge (d', v) \in V_D \wedge$ `DPmin[`(d, u, v)`]` $\leq d' \leq$ `DPmax[`(d, u, v)`]`$\}$. □

Stage 3: Marking all vertices in V_D whether it belongs to $D(s)$. After the first two stages, every vertex of $G_D = (V_D, A_D)$ is constructed, as well as an implicit representation of the arcs of G_D. We now need to visit the vertices of G_D and mark them according to Lemma 3. Because the representation does not include all arcs of G_D, solving this step using standard graph searches such as

Breadth-First Search seems difficult. More precisely, our implicit representation only stores in entries `DPmin[`(a)`]` and `DPmax[`(a)`]` the earliest and the latest departure dates for a valid transit at vertex $v = t(a)$ w.r.t. the (α, β)-transit condition. Let us assume $(d, u) \in D(s)$ for $d = d(a)$ and $u = s(a)$, and let us consider $v = t(a)$: if we do test whether (d', v) belongs to $D(s)$ by naively scanning all potential values of d' s.t. `DPmin[`(a)`]` $\leq d' \leq$ `DPmax[`(a)`]`, then the overall complexity will have a factor depending on $\max_{v \in V} |$`VArr[`v`]`$|$, which is a sharp upper bound for `DPmax[`(a)`]` $-$ `DPmin[`(a)`]`. This could lead to a non-linear complexity in the worst case.

Algorithm 3 Traverse the graph G_D

procedure TRAVERSE(G, α, β, s)
  Call Algorithm 2 GENERATETABLES(G, α, β) and obtain `VDep`, `DPmin` and `DPmax`
    ▷ We refer to V_D as the concatenation of all entries in `VDep`
  $Q \leftarrow$ a queue initially with all vertices in `VDep[`s`]`
  $Traversed \leftarrow$ a list that stores all vertices that are visited during the traversal
  for each vertex $u \in V$ **do**
    Make disjoint-set D_u ▷ With $.join(d, d')$ for union and $.comp(d)$ for find
    for each vertex $(d, u) \in V_D$ **do**
      $B_u[d] = -1$ ▷ Values in B_u will represent the date when vertex
        ▷ $(d, u) \in V_D$ was visited before, -1 represents `false`
  while Q is not empty **do**
    $(d, u) \leftarrow Q$.pop()
    for each $a = (d, u, v) \in A$ and $d' =$ `DPmin[`(a)`]` **do** ▷ Implicit iteration
      ▷ over $((d, u), (d', v)) \in A_D$
      $d'_{max} \leftarrow$ `DPmax[`(d, u, v)`]`
      $d'_{now} \leftarrow d'$
      $d'_{prev} \leftarrow d'$
      while $d'_{now} \leq d'_{max}$ **do**
        $visited = B_v[D_v.comp(d'_{now})]$
        $temp \leftarrow max(B_v[D_v.comp(d'_{prev})], B_v[D_v.comp(d'_{now})])$
        $D_v.join(d'_{now}, d'_{prev})$
        $B_v[D_v.comp(d'_{now})] \leftarrow temp$
        if $visited \neq -1$ **then** ▷ Value in B_v states if the vertex was visited
          $d'_{now} \leftarrow B_v[D_v.comp(d'_{now})]$ ▷ Jump to the vertex with the
            ▷ largest time in that component
        else
          $Q.push((d'_{now}, v))$
          $Traversed.append((d'_{now}, v))$
        $B_v[D_v.comp(d'_{now})] = max(d'_{now}, B_v[D_v.comp(d'_{now})])$
        $d'_{prev} \leftarrow d'_{now}$
        $(d'_{now}, v) \leftarrow$ next vertex after (d'_{now}, v) in `VArr[`v`]`

To achieve linear runtime, we use two main ideas. First, we use disjoint-set data structure to dynamically join the possible values of d' whenever we do the test whether (d', v) belongs to $D(s)$, for every $v \in V$. Generally, the use of

disjoint-set data structure leads to a quasi-linear time complexity. In our case, note that the possible values of d' for every $v \in V$ are in reality values in $\texttt{VArr}[v]$, Furthermore, already after calling Algorithm 1 the original values present in $\texttt{VArr}[v]$ are both known and totally ordered. Therefore, a faster case of disjoint-set data structure [7] can be used, with $O(1)$ amortised cost per operation. Hence, there is only a total contribution in time $O(\sum_{v \in V} |\texttt{VArr}[v]|) = O(|A|)$ for disjoint-set operations. The second idea is captured in the proof of Lemma 9. Roughly, we show with Lemma 9 that after joining dynamically the values of d' while testing whether (d', v) belongs to $D(s)$, we can control the out-degree of what is leftover in our implicit representation of G_D.

We formalize Stage 3 of our computation in Algorithm 3. Roughly, we initialize a visiting queue with all vertices in V_D associated to s, that is, of the form (d, s) for any d. At each iteration we remove the top element (d, u) of the queue, which is considered as a vertex of G_D. In this iteration, we would like to follow all arcs that the vertex (d, u) might have in G_D. For this purpose we make an implicit iteration: by iterating over every arc $a = (d, u, v) \in A$ and every possible value of d' s.t. $\texttt{DPmin}[(a)] \leq d' \leq \texttt{DPmax}[(a)]$, as explained in Algorithm 2 and Lemma 6. However, to control the global runtime, we only start with $d'_{now} \leftarrow d' = \texttt{DPmin}[(a)]$, and our plan is to check whether (d'_{now}, v) belongs to $D(s)$ while dynamically joining all previously visited $D_v.comp(d'_{now})$, for every possible value of d'_{now} between $d' = \texttt{DPmin}[(a)]$ and $d'_{max} = \texttt{DPmax}[(a)]$. These values are read from $\texttt{VArr}[v]$.

More precisely, for each unvisited vertex visited in this process the algorithm marks it as visited, adds it to the queue, joins the components for it and the previous vertex (itself in case it is the first one) in the disjoint-set D_v, sets as d'_{max} the reference to the largest visited vertex for its component and continue to the next vertex in increasing order of time in $\texttt{VArr}[v]$. When the algorithm reaches a vertex previously visited it goes to D_v and gets the identifier of the component it belongs to, jumps to the vertex with the maximum time that belongs to the component and continues to the next vertex. In total, the algorithm will visit the number of unvisited arcs in between d' and d'_{max} plus at most 3 previously visited vertices. The previous value 3 comes from the fact that when the algorithm gets for the first time to a visited vertex it will follow the link to the visited vertex with the largest time value in that component, then it continues iterating vertices until it arrives once more to a visited vertex and follows the link again. The link of the second visited vertex necessarily has to lead to a vertex with a time component at least as large as d'_{max}, and then the visiting process for this particular arc has finished. This is proven in Lemma 9. From the previous analysis it follows that over the whole traversal the algorithm will make at most $3|A| + |A|$ steps, where the first term comes from the bound obtained earlier and the second term is the maximum number of vertices in the graph.

Figure 4 exemplifies Algorithm 3. Therein, the upper and lower arrows indicate the links towards d' and d'_{max} at each time. Eventually, all vertices in between become part of the same component, which is represented by the same color, with the vertex in darker shade indicates the component with the largest

time. As we move from left to right we can see the progression of the components as they are visited. On the leftmost column of the picture we can see the component derived from a path arriving to b whose minimum departure time is 3 and whose maximum departure time is 6 (denoted by the top and bottom arrows pointing to these nodes). As there were no previous components on the disjoint-set data structure at this iteration we create a component that contains all nodes within those times. On the second column we see the resulting components after a path with minimum departure time 8 and maximum departure time 9. As this times do not intersect with those of the previous components we get two disjoint components. The same process can be seen in the third column, where the red component is expanded to include the node $(7, b)$. On the last column we get the result of processing a path that has minimum departure time from b equal to 5 and maximum departure time from b equal to 8. The nodes comprised between these times contain elements from both sets in the disjoint-set data structure. Therefore they are merged, resulting in a single set of all vertices of the form (d', b) in V_D.

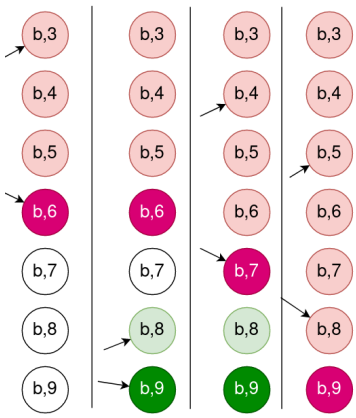

Fig. 4. The process of visiting all vertices in V_D associated with a single vertex b in V.

We recall that for each arc $a = (d, u, v) \in A$ a vertex $(d, u) \in V_D$ will have arcs towards all vertices in G_D associated to vertex v between time d' and d'_{max}. However, our representation of G_D does not include all these arcs: only (d', v) and (d'_{max}, v) are explicit. We will call all of these vertices associated to the same vertex $v \in V$ to which (d, u) has an arc in G_D cousins with respect to (d, u). This is defined formally in the following definition.

Definition 1. *Two vertices (d'_1, v) and $(d'_2, v) \in V_D$ associated to the same vertex $v \in V$ are cousins with respect to a vertex $(d, u) \in V_D$ if arc $(d, u, v) \in A$ exists, $d + c((d, u, v)) + \alpha(v) \leq d'_1 \leq d + c((d, u, v)) + \beta(v)$ and $d + c((d, u, v)) + \alpha(v) \leq d'_2 \leq d + c((d, u, v)) + \beta(v)$.*

Lemma 7. *There does not exist a vertex $(d'_2, v) \in V_D$ such that there exist two vertices (d'_1, v) and $(d'_3, v) \in V_D$ both associated to the same vertex $v \in V$ and $D_v.comp(d'_1) = D_v.comp(d'_3) \neq D_v.comp(d'_2)$ and $d'_1 < d'_2 < d'_3$.*

Proof. The only place where components are joined in the disjoint-set in Algorithm 3 is in line 21. As can be seen in this line of the algorithm, only components that are next to each other in the order given by `VArr[`v`]` can be merged. Lets suppose that d'_1 and d'_3 are part of the same component. As at the beginning all vertices belong to distinct components there should have been a moment when the components of d'_1 and d'_3 were merged. This implies that the components were next to each other and therefore there cannot be a value d'_2 in between these two components because otherwise the merge operation would not have been possible. This leads to the fact that all members of the same component are sequential in the order given by `VArr[`v`]` and proves the lemma. □

Lemma 8. *All vertices that are cousins with respect to a vertex (d, u) will be in the same component after (d, u) has been popped from the queue and its arcs traversed in Algorithm 3.*

Proof. Let C denote the component of all vertices that are cousins with respect to vertex (d, u). From the definition of the cousin relationship it is known that all vertices in C are sequential. After (d, u) has been popped from Q and the processing of all vertices begins through the *while* loop located in line 18 the algorithm starts processing the vertex with the smallest time d' and continues all the way until the vertex with the largest time d'_{max} each time going to the next vertex after the current component. The following will analyze each case the algorithm might encounter

- Reaches a previously unvisited vertex: The algorithm merges it with the previous component, by Lemma 7 all vertices in this new component are sequential.
- Reaches a previously visited vertex: The algorithm merges this component with the previous one and moves to the last visited vertex of this component. By Lemma 7 both merged components are sequential and therefore their merge produces a component with no gap.

Therefore all merged vertices will be sequential and because the process visits all components between d' and d'_{max} all previously existing components in that range will become one after the end of the process. □

Lemma 9. *For each arc $(d, u, v) \in A$ Algorithm 3 will visit $e_{(d,u,v)} + 4$ vertices where $\sum_{a \in A} e_a = O(|A|)$.*

Proof. From Algorithm 2, each arc in $(d, u, v) \in A$ leads to two explicit arcs $((d, u), (d', v))$ and $((d, u), (d'_{max}, v))$ of G_D, along with several implicit arcs of G_D, where $d' =$ `DPmin[`(d, u, v)`]` and $d'_{max} =$ `DPmax[`(d, u, v)`]`. After following one explicit arc it is possible to visit several other vertices by moving through the ordered list of vertices `VArr` created in Algorithm 1. From the inner *while*

loop in line 18 of the algorithm it can be seen that the number of steps that Algorithm 3 takes is bounded by the number of vertices visited by each explicit arc in G_D (by the *for* in line 13). What follows will analyze how many vertices will be visited in the traversal process, as this will be the number of steps that Algorithm 3 will perform. This can be seen in lines 18-31 of Algorithm 3 where for each iteration of the *while* loop there is a vertex that is processed. Therefore it is of interest to count the number of these aforementioned loops. The algorithm will loop until the value of d'_{now} is greater than that of d'_{max}. In lines 15-17 d'_{now} receives as initial value the minimum time d' such that the path that contains (d, u, v) can continue through an outgoing arc from v at time d', while d'_{max} receives as value the maximum at which the journey could continue through an outgoing arc from v. Both values where calculated in Algorithm 2. When the algorithm visits a vertex (d'_{now}, v) there are 2 possibilities, depending on whether (d'_{now}, v) has been visited before or not. What follows will analyze both cases. If the vertex has not been previously visited then it is marked as visited (line 22), its component is joined with that of the previous visited vertex (line 21) and the algorithm moves to the successor vertex. In this case the value of $e_{(d,u,v)}$ is increased by one. In this case vertex (d'_{now}, v) becomes the largest visited vertex of the component containing all vertices associated with v with times between d' and d'_{now} as can be deduced from the analysis of Lemma 8.

If the vertex has been visited previously then in line 24 we get the reference to the largest visited vertex of the component and move to the successor vertex. In what follows the argument that this can happen at most two times is presented. This is the same as showing that at most two previously visited components will be visited. By the definition of disjoint-set it is deduced that vertices in the first and second components of visited vertices cannot intersect because otherwise they would be part of the same set. By Lemma 7 it is also known that the vertex of the second component with the minimum time (let it be denoted by d'') should have a time bigger than that of d' because even if d' is the last vertex from the first visited set, vertex (d'', v) is at least the successor of the last vertex from the first visited set. Now following the link to the latest vertex of this component would lead to at least time `DPmax[`(d'', u, v)`]` given that from Lemma 8 it follows that if this is the first vertex in the component it will have at least enough vertices to lead to the said time, which because $d'' > d'$ should be at least as big as `DPmax[`(d', u, v)`]` and as such we have at least reached time d'_{max} and we finish the loop. □

Stage 4: Computing $R(s)$ and a foremost (α, β)-journey from s to t. Now that we have described the 3 main stages of the algorithm, we present them together in the form of the full procedure in Algorithm 4 and show that it is linear in Theorem 1.

Theorem 1. *On input a temporal digraph G, a pair of source and target vertices s, t in G, and two constraint functions $\alpha, \beta : V \to \mathbb{N}$, Algorithm 4 computes the arrival date of a foremost (α, β)-journey from s to t in linear time.*

Algorithm 4 Foremost (α, β)-journey arrival date in linear time

```
1: procedure FOREMOST(G, α, β, s, t)
2:     GENERATEVERTICES(G, α, β)                          ▷ Algorithm 1
3:     GENERATETABLES(G, α, β)                            ▷ Algorithm 2
4:     TRAVERSE(G, α, β, s)                               ▷ Algorithm 3
5:     ArrivalDate = ∞
6:     for each arc (d, u, v) in A do
7:         if v = t and (d, u) in traversed then
8:             ArrivalDate = min(ArrivalDate, d + c((d, u, v)))
9:     return ArrivalDate          ▷ the time of the foremost journey from s to t
```

Proof. From Lemmas 5 and 6 it is known that Algorithms 1 and 2 have linear complexity. All that is left to do is show that Algorithm 3 is linear. From Lemma 9 it is known that for each unvisited vertex the value of $e_{(d,u,v)}$ is increased by one, and because there are at most $2|A|$ vertices in V then $\sum_{a \in A} e_a \leq 2|A|$. Because each arc in A produces at most one iteration of the *for* loop in line 13 of Algorithm 3 then from Lemma 9 it is deduced that the number of visited vertices over all Algorithm 3 is smaller than $|A| + 4|A|$ and therefore the number of visited vertices is linear in the size of $|A|$ for Algorithm 3. Now that the complexity of Algorithms 1, 2 and 3 are linear it follows that Algorithm 4's complexity is linear as it is a combination of the three aforementioned algorithms plus an iteration over all arcs in A. □

5 Conclusion and perspectives

We addressed the problem of computing in a temporal graph the arrival date of a foremost journey under non-stop transit constraints. It is a polynomial time problem, contrary to the computation of a temporal path under the same constraints which is NP-hard. We then depict a linear time solution for finding the minimum arrival date of a foremost non-stop journey.

As for perspectives, it turns out that most our algorithmic steps follow the filter-map-reduce programming paradigm. In particular, we make intensive use of bucket sorting (a.k.a. radix sort) which can very naturally be implemented by lambdas. We believe that bucket sorting is important for processing historic data in general, and temporal graphs in particular. In this sense, we raise the open question whether our algorithm can be rewritten using a fully functional programming approach.

Acknowledgements We are grateful to the anonymous reviewers for their helpful comments which greatly improved the paper.

References

1. Bui-Xuan, B.M., Ferreira, A., Jarry, A.: Computing shortest, fastest, and foremost journeys in dynamic networks. International Journal of Foundations of Computer Science **14**(2), 267–285 (2003)
2. Casteigts, A., Flocchini, P., Mans, B., Santoro, N.: Shortest, fastest, and foremost broadcast in dynamic networks. International Journal of Foundations of Computer Science **26**(4), 499–522 (2015)
3. Casteigts, A., Himmel, A.S., Molter, H., Zschoche, P.: Finding temporal paths under waiting time constraints. Algorithmica **83**, 2754–2802 (2021)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press (1989)
5. Dibbelt, J., Pajor, T., Strasser, B., Wagner, D.: Connection scan algorithm. ACM Journal of Experimental Algorithmics **23** (2018)
6. Dupuy, M., d'Ambrosio, C., Liberti, L.: Optimal paths on the ocean (2021), https://hal.archives-ouvertes.fr/hal-03404586
7. Gabow, H., Tarjan, R.: A linear-time algorithm for a special case of disjoint set union. Journal of Computer and System Sciences **30**, 209–221 (1985)
8. Himmel, A.S., Bentert, M., Nichterlein, A., Niedermeier, R.: Efficient computation of optimal temporal walks under waiting-time constraints. In: 8th International Conference on Complex Networks and Their Applications. SCI, vol. 882, pp. 494–506 (2019)
9. Rymar, M., Molter, H., Nichterlein, A., Niedermeier, R.: Towards classifying the polynomial-time solvability of temporal betweenness centrality. In: 47th International Workshop on Graph-Theoretic Concepts in Computer Science. LNCS, vol. 12911, pp. 219–231 (2021)
10. Saramäki, J., Kivelä, M., Karsai, M.: Weighted temporal event graphs. Temporal Network Theory pp. 107–128 (2019)