A dynamic data structure for temporal reachability with unsorted contact insertions

Luiz F. A. Brito · Marcelo K. Albertini · Arnaud Casteigts · Bruno A. N. Travencolo

Received: 26 February 2021 / Revised: 13 August 2021 / Accepted: 15 August 2021 / Published online: 23 December 2021
© The Author(s), under exclusive licence to Springer-Verlag GmbH Austria, part of Springer Nature 2021

Abstract
Temporal graphs represent interactions between entities over the time. These interactions may be direct (a contact between two nodes at some time instant), or indirect, through sequences of contacts called temporal paths (journeys). Deciding whether an entity can reach another through a journey is useful for various applications in communication networks and epidemiology, among other fields. In this paper, we present a data structure that maintains temporal reachability information under the addition of new contacts. In contrast to previous works, the contacts can be inserted in an arbitrary order—in particular, non-chronologically—which corresponds to systems where the information is collected a posteriori (e.g., when trying to reconstruct contamination chains among people). The main component of our data structure is a generalization of transitive closure called timed transitive closure (TTC), which allows us to maintain reachability information relative to all nested time intervals, without storing all these intervals, nor the journeys themselves. Let $n$ be the number of nodes and $\tau$ be the number of timestamps in the lifetime of the temporal graph. Our data structure answers reachability queries regarding the existence of a journey from a given node to another within a given time interval in time $O(\log \tau)$; it has a worst-case insertion time of $O(n^2 \log \tau)$; and it can reconstruct a valid journey that witnesses reachability in time $O(k \log \tau)$, where $k < n$ is the maximum number of contacts of this journey. Finally, the space complexity of our reachability data structure is $O(n^2 \tau)$, which remains within the worst-case size of the temporal graph itself.

Keywords Temporal graph · Reachability · Dynamic data structures · Transitive closure

1 Introduction
Temporal graphs represent interactions between entities over the time. These interactions often appear in the form of contacts at specific timestamps. Moreover, entities can also interact indirectly with each other by chaining several contacts over time. For example, in a communication network, devices that are physically connected can send new messages or propagate received ones; thus, by first sending a new message and, then, repeatedly propagating messages over time, remote entities can communicate indirectly. Time-respecting paths in temporal graphs are known as temporal paths, or simply journeys, and when a journey exists from one node to another node, we say that the first can reach the second.

In a computational environment, it is often useful to check whether entities can reach each other. Investigations on temporal reachability have been used for characterizing mobile and social networks (Tang et al. 2010); for validating protocols and better understanding communication networks (Cacciari and Rafiq 1996; Whitbeck et al. 2012); for checking the existence of trajectories and improving flow in transportation networks (Wu et al. 2017; Williams and Musolesi 2016; Bedogni et al. 2018); for assessing future states of ecological networks (Martensen et al. 2017); and for making plans for agents using automation networks (Bryce and Kambhampati 2007). Beyond the sole reachability, some applications require the ability to reconstruct a concrete journey if one exists. For example, journey reconstruction...
has been used for finding and visualizing detailed trajectories in transportation networks (Wu et al. 2017; George et al. 2007; Zeng et al. 2014; Hasan et al. 2011); for visualizing system dynamics (Hurter et al. 2014); and for matching temporal patterns in temporal graph databases (Moffitt and Stoyanovich (2016); Latapy et al. 2018).

In standard graphs, the problem of maintaining reachability information under various modifications of the graph is known as dynamic connectivity and has been extensively studied (Agrawal et al. 1989; Wang et al. 2006; Cohen et al. 2003; Zhu et al. 2014; Seufert et al. 2013; Wei et al. 2018). Here, the adjective dynamic does not refer to the temporal nature of the network; it refers to the fact that the computed information is to be updated after the input graph is changed. This maintenance is performed by means of a dynamic data structure, which stores intermediate information to speed up the query time (and its own update time after a change).

Three types of dynamic data structures are classically considered, depending on the type of change allowed, namely incremental (insertion only), decremental (deletion only), and fully-dynamic (both). Typically, the elements to be inserted and removed in the classical version are the edges of the graph.

In the case of temporal graphs, the elements to be inserted are contacts, edges labeled with a timestamp, which indicates a particular moment the two corresponding nodes interacted. An important aspect of data structures that manipulate contacts is whether the order of insertion respects the order of the interactions themselves, i.e., the insertions are chronological. Algorithms for updating reachability information with the assumption that the input is chronological have been proposed (Barjon et al. (2014); Whitbeck et al. 2012). However, this assumption does not capture important use cases where the contacts are collected in an unpredictable order and the reachability information is updated afterward. For instance, during scenarios of epidemics, outdated information containing the interaction details among infected and non-infected individuals are reported in an arbitrary order. Then, this information is periodically queried in order to better understand the dissemination process and then take appropriate measures for reducing reachability and identifying sources of contamination (Xiao et al. xxx, 2018; Enright et al. 2021; Rozenshtein et al. 2016).

Motivated by such scenarios, we investigate the problem of maintaining an incremental data structure for temporal reachability, where the insertions of contacts are made in an arbitrary order. A naive approach is to store and update the temporal graph itself (e.g., as a set of contacts) and then run standard journey computation algorithms like (Xuan et al. 2003) when a query is made. However, the goal of a data structure is to reduce the computational cost of the queries based on pre-computing intermediate information. In fact, data structures typically offer a tradeoff between query time, update time, and space. To the best of our knowledge, the only existing work supporting non-chronological contact insertions and exploiting intermediate representations for speeding up reachability queries in temporal graphs is Wu et al. (2016). The solution in Wu et al. (2016) relies on maintaining a directed acyclic graph (DAG) in which every original vertex is possibly copied up to \( \tau \) times (where \( \tau \) is the number of timestamps) and a journey exists from \( u \) to \( v \) in the interval \([t_1', t_2']\) if and only if vertex \( u \) can reach vertex \( v \), in the DAG. Some paths preprocessing is additionally considered that results in an average speed up for reachability queries. However, the worst-case query time corresponds to a standard path search (e.g., depth-first search) in the DAG, which takes \( \Theta(n^2 \tau) \) time in the case of dense temporal graphs (whose number of contacts is of the same order). The space complexity (size of the DAG) also corresponds essentially to the number of contacts, thus \( \Theta(n^2 \tau) \) in the worst case. Finally, the update time upon insertion is quite efficient, because the DAG representation allows its effect to remain local. If one ignores the cost of paths preprocessing in Wu et al. (2016) (as we focus on worst-case analysis), it only takes \( O(1) \) time to update the DAG if the corresponding nodes are already known, and up to \( \Theta(\tau) \) otherwise, due to the creation of (up to) \( \tau \) copies of the new nodes.

### 1.1 A data structure for unsorted contact insertions

In this paper, we consider the problem of maintaining reachability information through a data structure\(^1\) that supports the following four operations, where, by convention, \( \mathcal{G} \) is a temporal graph, \( u \) and \( v \) are vertices of \( \mathcal{G} \), and \( t, t_1, \) and \( t_2 \) are timestamps:

- add_contact\((u, v, t)\): Update information based on a contact from \( u \) to \( v \) at time \( t \). This operation may be called, for example, every time a new contact is discovered in the considered scenario.
- can_reach\((u, v, t_1, t_2)\): Return true if \( u \) can reach \( v \) within the interval \([t_1, t_2]\).
- is_connected\((t_1, t_2)\): Return true if \( G \) restricted to the interval \([t_1, t_2]\) is temporally connected, i.e., all vertices can reach each other within the interval \([t_1, t_2]\). This operation allows one to test, for example, whether all the nodes of a communication network can reach each other.
- reconstruct_journey\((u, v, t_1, t_2)\): Return a journey (if one exists) from \( u \) to \( v \) occurring within the interval \([t_1, t_2]\). This operation may be called, for example, to learn a potential contamination chain from a person to another in a scenario of epidemics.

\(^1\) We have a simple implementation available at https://github.com/albertiniufu/dynamic_temporal_graph/.
For generality, we consider directed contacts (timed arcs). Furthermore, if \( t_1 \) and \( t_2 \) are omitted in the above operations, then the entire lifetime of \( G \) is considered. The challenge in realizing these operations is to answer queries as fast as possible, while keeping space consumption and update time at reasonable levels. The worst-case complexities are as follows: The queries, \( \text{can}_\text{reach}(u, v, t_1, t_2) \) and \( \text{is}_\text{connected}(t_1, t_2) \), run, respectively, in \( O(\log \tau) \) and \( O(n^2 \log \tau) \) time; the update operation, \( \text{add}_\text{contact}(u, v, \tau) \), runs in \( O(n^2 \log \tau) \) worst-case time; and the retrieval operation, \( \text{reconstruct}_\text{journey}(u, v, t_1, t_2) \), runs in \( O(k \log \tau) \) time, where \( k < n \) is the length of the resulting journey. The worst-case space complexity remains within the worst-case size of the temporal graph itself, namely \( O(n^2 \tau) \). Overall, the space complexity is comparable to that of Wu et al. (2016), while the query time is much faster and the update time is slower.

The core of our data structure is a component called the \textit{timed transitive closure} (TTC), which generalizes the classical notion of a transitive closure (TC). Classical TCs capture reachability information among vertices over the entire lifetime of the network. They are classically encoded as a static directed graph where the existence of an arc from \( u \) to \( v \) implies that there is a journey \( u \rightarrow v \) in the temporal graph. If one is not interested in querying reachability for specific subintervals, and if the contacts are inserted in chronological order, then TCs are actually sufficient for maintaining temporal reachability information (see Barjon et al. (2014)). A generalization of TC has also been considered in Whitbeck et al. (2012), which allows queries to be parametrized by a maximum journey duration; however, basic journey information, such as departure and arrival times, is not known, and the computation of the structure requires the information to be processed at once and chronologically (i.e., subsequent updates are not supported).

In the unsorted (i.e., non-chronological) case, TCs do not provide enough information to decide whether a new contact (possibly occurring at any point in history) can be composed with known journeys. To address this need, we introduce a generalization of TCs called \textit{timed transitive closure} (TTCs), which store information regarding the availability of journeys for a well-chosen set of time intervals, without storing the journeys themselves. We study the general properties of TTCs, and we prove, in particular, that one can restrict the number of intervals considered to \( O(\tau) \) for any pair of nodes (as opposed to \( O(\tau^2) \)), with immediate consequences on the space complexity of a data structure based on TTCs. This information is then exploited by our data structure algorithms.

1.2 Organization of the document

This paper is organized as follows. In Sect. 2, we present basic definitions. In Sect. 3, we introduce timed transitive closures, study their basic properties, and provide a number of low-level primitives for manipulating them. In Sect. 4, we describe the algorithms that perform each operation of our data structure based on TTCs, together with their running time complexities. Finally, Sect. 5 concludes with some remarks and open questions.

2 Definitions

Following the formalism in Casteigts et al. (2012), a temporal graph can be generally represented by a tuple \( G = (V, E, \tau, \rho, \zeta) \), where \( V \) is a set of vertices, \( E \subseteq V \times V \) is a set of edges, \( \tau \) is the time interval over which the network exists (lifetime), \( \rho : E \times \tau \rightarrow \{0, 1\} \) is a presence function that expresses whether a given edge is present at a given time instant, and \( \zeta : E \times \tau \rightarrow \tau \) is a latency function that expresses the duration of an interaction for a given edge at a given time, where \( \tau \) is the time domain (typically \( \mathbb{R} \) or \( \mathbb{N} \)). In this paper, we consider a setting where \( E \) is a set of arcs (directed edges), \( \tau \) is equal to \( \mathbb{N} \) (time is discrete) and \( \tau = [1, \tau] \subseteq \tau \) (the lifetime contains \( \tau \) timestamps). The latency function is constant, namely \( \zeta = \delta \), where \( \delta \) is any fixed positive integer (typically 0 or 1). We call \( (u, v, \tau) \) a \textit{contact} in \( G \) if \( \rho((u, v), \tau) = 1 \). We use a short-hand notation \( G_{[t_1, t_2]} \) when restricting the lifetime of \( G \) to a subinterval \([t_1, t_2] \subseteq \tau \), and call \( G_{[t_1, t_2]} \) a temporal subgraph of \( G \). Finally, the static graph \( G = (V, E) \) is called the \textit{underlying graph} of \( G \), and the static graph containing only the contacts that exist at time \( \tau \) is called the \textit{snapshot} of \( G \) at time \( \tau \), noted \( G_{\tau} \).

Reachability in temporal graphs is defined in a time-respecting way, by requiring that a path travels along increasing times (respectively, non-decreasing times) if \( \delta \geq 1 \) (respectively, if \( \delta = 0 \)). These paths are called temporal paths or journeys, interchangeably.

\textbf{Definition 1} (\textit{Journey}) A journey from \( u \) to \( v \) in \( G \) is a sequence of contacts \( \mathcal{J} = (c_1, c_2, \ldots, c_k) \), whose sequence of underlying arcs form a valid \((u, v)\)-path in the underlying graph \( G \) and for each contact \( c_i = (u_i, v_i, t_i) \), it holds that \( \rho((u_i, v_i), t_i) = 1 \) and \( t_{i+1} \geq t_i + \delta \) for each \( i \in [1, k - 1] \). Additionally, we say that \textit{departure}(\mathcal{J}) = t_1, \textit{arrival}(\mathcal{J}) = t_k + \delta \) and \textit{duration}(\mathcal{J}) = \textit{arrival}(\mathcal{J}) - \textit{departure}(\mathcal{J}).

A journey is \textit{trivial} if it consists of a single contact.

\textbf{Definition 2} (\textit{Reachability}) A vertex \( u \) can \textit{reach} a vertex \( v \) within time interval \([t_1, t_2]\) iff there exists a journey \( \mathcal{J} \)
from $u$ to $v$ in $G_{[t_1,t_2]}$ (i.e., such that $\text{departure}(\mathcal{J}) \geq t_1$ and $\text{arrival}(\mathcal{J}) \leq t_2$).

The (standard) transitive closure (TC) of a temporal graph $G$ is a static directed graph $G^* = (V,E^*)$ such that $(u,v) \in E^*$ if and only if $u$ can reach $v$ in $G$. This notion is illustrated in Figure 1. As already explained, if the contacts are discovered in a chronological order, then $G^*$ can be updated incrementally to compute the entire reachability information of $G$ (Barjon et al. (2014)). In the unsorted case, however, this notion is not sufficient because it does not allow one to decide if a new contact can be composed with previously-known journeys, which motivates the definition of more powerful objects.

3 Reachability tuples and timed transitive closure

In this section, we describe an extension of the concept of transitive closure called timed transitive closure (TTC). The purpose of TTCs is to encode reachability information among the vertices, parametrized by time intervals, so that one can subsequently decide if a new contact occurring anywhere in history can be composed with existing journeys. The main components of TTCs are called reachability tuples (R-tuples). We introduce a number of operators on R-tuples, such as inclusion and concatenation, and describe their role in the construction and maintenance of a TTC.

3.1 Reachability tuples (R-tuples)

Just as the number of paths in a static graph, the number of journeys in a temporal graph could be too large to be stored explicitly (typically, factorial in $n$). To avoid this problem, R-tuples capture the fact that a node can reach another within a certain time interval without storing the corresponding journeys. Thus, a single R-tuple may capture the reachability information corresponding to many possible journeys. We distinguish between two versions of R-tuples, namely (existential) R-tuples and constructive R-tuples, the latter adding information for reconstructing a journey that witnesses reachability.

3.1.1 Existential R-tuples

The following definitions are given in the context of a temporal graph $G$ whose vertex set is $V$, lifetime is $T = [1, \tau]$, and latency is $\delta$.

Definition 3 (R-tuple) An (existential) R-tuple is a tuple $r = (u,v,t^-,t^+)$, where $u$ and $v$ are vertices in $G$, and $t^-$ and $t^+$ are timestamps in $T$. It encodes the fact that node $u$ can reach node $v$ through a journey $\mathcal{J}$ such that $\text{departure}(\mathcal{J}) = t^-$ and $\text{arrival}(\mathcal{J}) = t^+$. If several such journeys exist, then they are all captured by the same R-tuple.

The set of journeys captured by an R-tuple $r$ is denoted by $\mathcal{J}(r)$, and we say that $r$ represents these journeys. An R-tuple is trivial when it represents a trivial journey (i.e., a single contact). Trivial R-tuples thus have the form $(u,v,t,t+\delta)$ for some $t$. The following relations and operations, compatible with any fixed value of parameter $\delta$, are quite natural to define.

Definition 4 (Precedence $\prec$) An interval $I_1 = [t_1^-,t_1^+]$ precedes an interval $I_2 = [t_2^-,t_2^+]$, noted $I_1 \prec I_2$, if $t_1^+ \leq t_2^-$. Given two R-tuples $r_1 = (u_1,v_1,t_1^-,t_1^+)$ and $r_2 = (u_2,v_2,t_2^-,t_2^+)$, $r_1$ precedes $r_2$, noted $r_1 \prec r_2$ if $t_1^+ \leq t_2^-$ and $u_2 = v_1$.

Intuitively, the precedence relation among R-tuples tells us that the journeys they represent can be composed, leading to another R-tuple as follows:

Definition 5 (Concatenation $\cdot$) Given two R-tuples $r_1 = (u_1,v_1,t_1^-,t_1^+)$ and $r_2 = (u_2,v_2,t_2^-,t_2^+)$ such that
\[ r_1 < r_2, \text{ the concatenation of } r_1 \text{ with } r_2 \text{ is the R-tuple } r_1 \cdot r_2 = (u_1, v_2, t_1^r, t_2^r). \]

The natural inclusion among intervals extends to R-tuples as follows:

**Definition 6** *(Inclusion \( \subseteq \))* Given two R-tuples \( r_1 = (u_1, v_1, t_1^r, t_1^r) \) and \( r_2 = (u_2, v_2, t_2^r, t_2^r) \), \( r_1 \subseteq r_2 \) if and only if \( u_1 = u_2, v_1 = v_2, \) and \( [t_1^r, t_1^r] \subseteq [t_2^r, t_2^r] \) (that is, \( t_2^r \leq t_1^r \leq t_1^r \leq t_2^r \)).

If neither \( r_1 \not\subseteq r_2 \) nor \( r_2 \not\subseteq r_1 \) (or if the vertices are different), then \( r_1 \) and \( r_2 \) are called *incomparable*. Intuitively, if \( r_1 \not\subseteq r_2 \), then any of the journeys represented by \( r_2 \) could be replaced by a (possibly faster) journey represented by \( r_1 \). More precisely:

**Lemma 1** Let \( u \) and \( v \) be two nodes in \( V \). Let \( I_1 = [t_1^r, t_1^r] \) and \( I_2 = [t_2^r, t_2^r] \) be two subintervals of \( T \) such that \( I_1 \subseteq I_2 \). If \( u \) can reach \( v \) within \( I_1 \), then \( u \) can reach \( v \) within \( I_2 \).

**Proof** The proof is straightforward, we give it for completeness. Let \( r \) be the R-tuple \( (u, v, t_1^r, t_1^r) \) and let \( J \) be any of the journeys in \( J(r) \). One can reach \( v \) from \( u \) within \( I_2 \) through the three following steps: (1) wait at \( t_1^r \) to \( t_2^r \), (2) travel from \( u \) to \( v \) using \( J' \), and finally (3) wait at \( v \) from \( t_2^r \) to \( t_1^r \).

The main consequence of Lemma 1 is that if \( r_1 \not\subseteq r_2 \), then \( r_2 \) is redundant for answering reachability queries from \( u \) to \( v \).

**Definition 7** *(Redundancy)* Let \( S \) be a set of R-tuples and let \( r \in S \), \( r \) is called redundant in \( S \) if there exists \( r' \in S \) such that \( r' \not\subseteq r \). A set with no redundant R-tuple is called *irredundant*.

An R-tuple that is non-redundant in a set is also called *minimal* (in that set). It is natural to ask what the maximum size of an irredundant set of R-tuples could be, with consequences for the space complexity of a reachability data structure based on R-tuples. It turns out that this number is always significantly smaller than the number of possible R-tuples.

**Lemma 2** The maximum size of an irredundant set of R-tuples for \( G \) is \( \Theta(n^2 \tau) \).

**Proof** First, we prove that the maximum number of pairwise incomparable R-tuples is \( \Theta(n^2 \tau) \). Then, we show that this bound is tight, as some graphs induce \( \Theta(n^2 \tau) \) incomparable R-tuples.

(1) **Upper bound**

There are \( \Theta(n^2) \) ordered pairs of vertices. Thus, it is sufficient to show that for each pair \((u, v)\), the number of incomparable R-tuples whose starting vertex is \( u \) and whose ending vertex is \( v \) is \( \Theta(\tau) \). Let \( S \) be an irredundant set of such R-tuples, and let \( r_1 = (u, v, t_1^r, t_1^r) \) and \( r_2 = (u, v, t_2^r, t_2^r) \) be any two R-tuples in \( S \). If \( t_1^r = t_2^r \), then either \( r_1 \subseteq r_2 \) or \( r_2 \subseteq r_1 \); thus, \( S \) is redundant (contradiction). As a result, all departure timestamps \( t^r \) belonging to the R-tuples in \( S \) are different, which implies that \( |S| \leq \tau \).

(2) **Tightness**

Consider the complete temporal graph \( K_{n, \tau} \) on \( n \) vertices in which every edge is present in all timestamps in \([1, \tau]\). In such a graph, there are consequently \( \Theta(n^2 \tau) \) contacts, each of which is a trivial journey. Now, observe that either these journeys connect different vertices, or their intervals are incomparable (same duration with different starting times); thus, none of them is redundant with the others.

Given a graph \( G \) and a set \( S \) of R-tuples representing all the journeys of \( G \), the subset \( S' \subseteq S \) of all minimal R-tuples is called the representative R-tuples of \( G \), denoted by \( R(G) \). We also write \( R(u, v) \) for those R-tuples in \( R(G) \) whose source is \( u \) and destination is \( v \). From the proof of Lemma 2, we extract:

**Observation 1** Every contact of \( G \) is present in \( R(G) \) in the form of a trivial R-tuple.

Observation 1 implies that \( R(G) \) is a non-lossy representation, as \( G \) itself is contained in it. The downside is that its space complexity is at least as large as the number of contacts in \( G \). Observe that, up to a constant factor, it can, however, not be worse than the worst number of contacts, since there may exist up to \( \Theta(n^2 \tau) \) contacts, and irredundant sets cannot exceed this size (Lemma 2). In other words, in dense temporal graphs, the reachability information offered by R-tuples is essentially free in space.

### 3.1.2 Constructive R-tuples

The data structure considered in this work has four operations, namely add_contact, can_reach, is_connected, and reconstruct_journey. The first three operations can be dealt with using only existential R-tuple. The fourth operation could benefit from storing a small amount of additional information into the R-tuple.

**Definition 8** *(Constructive R-tuple)* A constructive R-tuple is a tuple \( r = (u, v, t^r, t^+, w) \) that contains the same information
as an existential R-tuple, plus a vertex \( w \) such that at least one journey \( J \in J(r) \) starts with the contact \((u, w, r^-)\). Node \( w \) is called the successor of \( u \) in \( r \) (respectively, in \( J \)).

Most of the definitions and lemmas from Sect. 3.1 apply unchanged to constructive R-tuple. In particular, the definition of redundant R-tuples applies without considering the successor field. Indeed, if two constructive R-tuples differ only by the successor node, then they are seen as equivalent and any of the two can be discarded. As for the concatenation of two constructive R-tuples \( r_1 = (u_1, v_1, r^-_1, r^+_1, w_1) \) and \( r_2 = (u_2, v_2, r^-_2, r^+_2, w_2) \), provided \( r_1 \prec r_2 \), we additionally require that the resulting R-tuple adopts the successor of \( r_1 \) as its own successor; that is, \( r_1 \cdot r_2 = (u_1, v_2, r^-_1, r^+_1, w_1) \). For simplicity, whenever constructive R-tuples are not needed, we describe the algorithms using existential R-tuples.

### 3.2 Timed transitive closure

Informally, the timed transitive closure of a temporal graph \( G \) is a multigraph that captures the existence of journeys within all possible time intervals, based on irredundant R-tuples.

**Definition 9 (Timed transitive closure)** Given a graph \( G \), the timed transitive closure of \( G \), noted TTC \((G)\), is a (static) directed multigraph on the same set of vertices, whose arcs correspond to the representative R-tuples of \( G \).

Figure 2 shows two examples of TTCs, both corresponding to the temporal graph of Figure 1 (one for existential R-tuples, the other for constructive R-tuples). Algorithmically, a TTC provides most of the support needed to realize the high-level operations of our data structure. For example, the operation \( \text{can}_\text{reach}(u, v, t_1, t_2) \) amounts to testing if there exists an arc whose associated R-tuple is \((u, v, r^-, r^+)\) with \([r^-, r^+] \subseteq [t_1, t_2] \). The operation \( \text{is}_\text{connected}(t_1, t_2) \) can be realized by performing such a test for every pair of vertices. The operation \( \text{add}_\text{contact}(u, v, t) \) reduces to adding a new arc to the TTC \((G)\) if no smaller interval already captures this information. If the new arc is added, then some other arcs may become redundant and should be removed, some others may also be created by composition. This operation is therefore the most critical. Finally, if constructive R-tuples are used, then an actual journey may be reconstructed quite efficiently from TTC \((G)\) when \( \text{reconstruct}_\text{journey}(u, v, t_1, t_2) \) is called, by retrieving a constructive R-tuple \((u, v, r^-, r^+, w)\) such that \([r^-, r^+] \subseteq [t_1, t_2] \) and unfolding the corresponding journey inductively, by replacing \( u \) with the successor vertex \( w \) and \( r^- \) with \( r^+ + \delta \) in each step.

All algorithms for these operations are described in Sect. 4. Before doing so, we present an explicit encoding of TTCs based on adjacency matrices and binary search trees (BST). In order for the high-level algorithms to remain independent of this particular choice, we define a set of primitives for manipulating the TTC, that are used by the high-level algorithms of Sect. 4.

#### 3.2.1 Encoding the TTC

We encode the TTC by an \( n \times n \) matrix, in which every entry \((i, j)\) points to a self-balanced binary search tree (BST) denoted by \( T(i, j) \). The nodes in this tree contain all the time intervals corresponding to R-tuples in \( R(i, j) \). From Lemma 2, we know that a tree \( T(u, v) \) contains up to \( r \) nodes. In addition, all these intervals are incomparable; thus, one can use any of their boundaries (departure or arrival) as the sorting key of the BST. Note that retrieving \( T(u, v) \) within the matrix takes constant time, as the cells of a matrix are directly accessed. Also recall that finding the largest key below (respectively, the smallest key above) a certain value takes \( O(\log r) \) time. Similarly, inserting a new element (in our case, an interval) takes \( O(\log r) \) time.
that several types of BST (e.g., red–black trees) can self-balance without impacting the asymptotic cost of insertions.

We provide the following low-level operations for manipulating TTCs: (1) find_next \( u, v, t \) returns the node containing the earliest interval \([r^-, r^+]\) in \( T(u, v)\) such that \( r^+ \geq t\), if any, and nil otherwise; symmetrically, (2) find_previous \( u, v, t \) returns the node containing the latest interval \([r^-, r^+]\) in \( T(u, v)\) such that \( r^+ \leq t\), if any, and nil otherwise; finally, (3) insert \( u, v, r^-, r^+ \) inserts a new node containing the interval \([r^-, r^+]\) in \( T(u, v)\) and performs some operations for maintaining the property that all intervals in \( T(u, v)\) are minimal.

Let us now describe the algorithms that perform these operations, along with their time complexities. The algorithm for find_next \( u, v, t \) searches \( T(u, v)\) recursively, by comparing \( t\) with the departure \( t\) of the current node interval \([r^-, r^+]\). If \( t^+\) is equal to or greater than \( t\), then the current node is a candidate answer. The algorithm then compares the current node candidate and the previous one and keeps the one containing the smallest (earliest) \( t\); then, it descends the left child. Otherwise, if \( t\) is smaller than \( t\), it simply descends the right child. As soon as a leaf is reached (and visited), the algorithm returns the current candidate as the answer. The algorithm for find_previous \( u, v, t \) works symmetrically. The time complexities of both algorithms correspond to the depth of the tree, which is \( O(\log \tau)\).

The algorithm for insert \( u, v, r^-, r^+\) finds and removes any potential node with interval \( \mathcal{I}_i\) such that \([r^-, r^+] \subseteq \mathcal{I}_i\); then, it inserts a new node containing \([r^-, r^+]\) using a standard BST insertion. Figure 3 shows a linear representation of the intervals in \( T(u, v)\) while performing this operation. A naive implementation of this operation would consist of searching and removing each corresponding node independently. However, this would lead to a complexity of \( O(d \log \tau)\) time, where \( d\) is the number of nodes removed, that is up to \( O(\tau)\). We use a non-standard approach that makes it feasible in \( O(\log \tau)\) time only. The strategy is to first identify in \( T(u, v)\) the nodes containing the boundary intervals \( \mathcal{I}_1\) and \( \mathcal{I}_2\) that correspond to the first and last nodes to be removed. The boundary interval \( \mathcal{I}_1\) is found by calling \( \mathcal{I}_1 = \text{find_previous}\ u, v, r^+\), then checking whether \( \mathcal{I}_1\) itself must be removed or not. If the arrival time of \( \mathcal{I}_1\) is smaller than \( r^+\), then either \([r^-, r^+]\) is redundant, and therefore the algorithm stops, or \( \mathcal{I}_1\) should not be removed, and therefore \( \mathcal{I}_1\) is replaced by the next node with greater key in \( T(u, v)\). Similarly, \( \mathcal{I}_2\) is found by first calling \( \mathcal{I}_2 = \text{find_next}\ u, v, r^-\), then, if the departure time of \( \mathcal{I}_2\) is greater than \( r^-\), \( \mathcal{I}_2\) is replaced by the previous node with smaller keys in \( T(u, v)\).

Note that the parameters passed to find_previous and find_next are indeed \( r^+\) and \( r^-\), not the reverse. Then, every node containing intervals in this range is removed using the technique outlined in the proof of Lemma 3.

**Lemma 3** In the worst case, the cost of the insert operation is \( O(\log \tau)\).

**Proof** The range of intervals to be removed is characterized by two boundary intervals \( \mathcal{I}_1\) and \( \mathcal{I}_2\), which can be found by calling both find_next and find_previous a single time, which takes \( O(\log \tau)\) time. The final insertion of the input interval in the BST also takes \( O(\log \tau)\). The difficult part is thus the removal of redundant intervals prior to this insertion (illustrated abstractly in Fig. 3). Let split\( T\) be the operation that splits a balanced BST into two balanced BSTs \( T_{< \mathcal{I}}\) and \( T_{\geq \mathcal{I}}\), where the first contains the intervals earlier than \( \mathcal{I}\) and the second the intervals later or equal to \( \mathcal{I}\). Let join\( T_1, T_2\) be the operation that receives as input two balanced BSTs and joins them into a single balanced BST. We proceed as follows. First, we split \( T(u, v)\) into two trees \( T_{< \mathcal{I}}\) and \( T_{\geq \mathcal{I}}\). Then, we split \( T_{\geq \mathcal{I}}\) into two trees \( T_{\geq \mathcal{I}_1, < \mathcal{I}_2}\) and \( T_{\geq \mathcal{I}_2}\) and remove the smallest node of \( T_{\geq \mathcal{I}_2}\) (by standard operations) to obtain \( T_{> \mathcal{I}_2}\). The join of \( T_{< \mathcal{I}}\) and \( T_{> \mathcal{I}_2}\) is considered to be the new \( T(u, v)\). Indeed, this tree consists of all the original intervals except the ones in the range to be removed. Both the split and join operations are known to be feasible at cost \( O(\log \tau)\) on typical self-balanced trees, such as red–black trees (see Blelloch et al. 2016 for details).

Additionally, we define the following basic operations:

- **\( N_{out}(u)\)**: Returns the set of vertices \( \{v_1, v_2, \ldots, v_k\}\) such that there exists at least one arc from \( u\) to \( v_i\) in the TTC.
Both operations can be realized in $O(n)$ time, through traversing the corresponding row (respectively, column) of the matrix and testing if the corresponding tree is empty.

4 The four operations

In this section, we describe the algorithms that perform the four operations of our data structure, whose contact is discussed in Sect. 1.1. These operations are $\text{can\_reach}(u,v,t_1,t_2)$, $\text{is\_connected}(t_1,t_2)$, $\text{add\_contact}(u,v,t)$, and (optionally) $\text{reconstruct\_journey}(u,v,t_1,t_2)$. For simplicity, the first three algorithms are presented using existential R-tuples only. (However, they are straightforwardly adaptable to constructive R-tuples.) All the algorithms rely on the primitives defined in Sect. 3.2.1 for manipulating the TTC in an abstract way. Then, we provide some experimental results regarding the average-case behavior of our data structure over contact insertions, and we formulate some open questions related to it.

4.1 Reachability and connectivity queries

The algorithm for $\text{can\_reach}(u,v,t_1,t_2)$ is straightforward. It consists of testing whether $T(u,v)$ contains at least one interval that is included in $[t_1,t_2]$. This can be done by retrieving $[r^-,r^+] = \text{FIND\_NEXT}(u,v,t_1)$ and checking that $t^+ \leq t_2$. Therefore, the cost of this algorithm reduces essentially to that of the operation $\text{FIND\_NEXT}$, which takes $O(\log \tau)$ time. We note that, if $[t_1,t_2] = T$, then it is sufficient to verify (in constant time) that $T(u,v)$ is not empty. Regarding the operation $\text{is\_connected}(t_1,t_2)$, a simple way of answering it is to call $\text{can\_reach}(u,v,t_1,t_2)$ for every pair of vertices, with a resulting time complexity of $O(n^2 \log \tau)$. It seems plausible that this strategy is not optimal and could be improved in the future.

4.2 Update operation

The algorithm for $\text{add\_contact}(u,v,t)$ manages the insertion of a new contact $(u,v,t)$ in the data structure, where $(u,v) \in E$ and $t \in T$. To start, the interval corresponding to the trivial journey from $u$ to $v$ over $[t,t+\delta]$ is inserted in $T(u,v)$ using the $\text{INSERT}$ primitive. (Recall that this primitive encapsulates the removal of redundant intervals in $T(u,v)$, if any.) Then, the core of the algorithm consists of computing the indirect consequences of this insertion for the other vertices. Namely, if a vertex $w$ could reach $u$ before time $t$ with latest departure $t$ and $v$ could reach another vertex $w^+$ after time $t + \delta$ with earliest arrival $r^+$, it follows that $w^-$ can now reach $w^+$ over interval $[r^-,r^+]$. Our algorithm consists of enumerating these compositions and inserting them in the TTC. Interestingly, for each predecessor $w$ of $u$, only the latest interval ending before $t$ in $T(w^-,u)$ needs to be considered. The reason is that in order to compose an earlier journey $J'$ with the new contact, we need to wait at $u$ until time $t$. Thus, even though some other journey started earlier, it would have to wait at $u$ and it would thus eventually arrive at the same time (based on a non-minimal interval). Based on this property, our algorithm only searches for the latest interval preceding $t$ for each predecessor of $u$ and the earliest interval exceeding $t + \delta$ for each successor of $v$.

The details are given in Algorithm 1, whose behavior is as follows. In Line 1, the algorithm inserts the interval $[t,t+\delta]$ into $T(u,v)$, which corresponds to the trivial journey induced by the new contact. In Lines 2 to 7, for every vertex $w^- \in N^*_w(u)$, it finds the latest interval $[r^-,\_\_\_]$ in $T(w^-,u)$ that arrives before time $t$ (inclusive) and inserts the composition $[r^-,t+\delta]$ into $T(w^-,v)$. For the same reasons as above, the algorithm only needs to consider inserting $[r^-,t+\delta]$ because every other possible composition would contain it as a subinterval. In Lines 8 to 11, for every vertex $w^+ \in N^*_w(v)$, the algorithm finds the earliest interval $[\_\_,r^+]$ in $T(v,w^+)$ that leaves $v$ after time $t + \delta$ (inclusive), and inserts the composition $[\_\_,r^+]$ into $T(w^-,v)$. In the same way, every other possible composition would contain $[t,r^+]$ as a subinterval. Finally, in Lines 12 to 14, for all $w^- \in N^*_w(u)$ and $w^+ \in N^*_w(v)$, it inserts the composition $[r^-,r^+]$ into $T(w^-,w^+)$. In order to optimize this last step, the algorithm only considers the subset of $N^*_w$ whose reachability to $v$ has been impacted by the new contact, thanks to a dedicated storage $D$ computed in Line 7.
Theorem 1 The update operation has worst-case time complexity $O(n^2 \log \tau)$.

Proof An insert operation is performed in Line 1. The loop from Line 3 to 7 iterates over $O(n)$ vertices and makes at most one insertion for each. The loop from Line 8 to 14 iterates over $O(n)$ vertices, and for each one, iterates in a nested way over $O(n)$ vertices. For each resulting pair, it performs at most one insert operation. The latter clearly dominates the overall cost of the algorithm, with a cost of $O(n^2)$ times the cost of the insert operation, the latter being of time $O(\log \tau)$ (Lemma 3). □

4.3 Journey reconstruction

The algorithm for the operation reconstruct_journey$(u, v, t_1, t_2)$ reconstructs a journey from vertex $u$ to vertex $v$ whose contact timestamps must be contained in $[t_1, t_2]$. As explained in Sect. 3.1.2, existential R-tuples can be augmented by a successor field that indicates which vertex comes next in (at least one of) the journeys represented by the R-tuple. This information is very useful for reconstruction and has a negligible cost (asymptotically speaking). Concretely, one can make the nodes of the BST store the successor field in addition to the interval. The low-level operations for manipulating the TTC (see Sect. 3.2.1) are unaffected, neither are the query and update algorithms in a significant way. The only subtlety is that when two intervals (nodes) are composed, the successor field of the resulting node corresponds to the successor field of the first node (this was already discussed in terms of R-tuples in Sect. 3).

The goal of the algorithm is thus to reconstruct a journey by unfolding the intervals and successor fields. Details are given in Algorithm 2.
The first step (Lines 1 to 3) is to retrieve a node in \( T(u, v) \) whose interval is contained within \([t_1, t_2]\) if one exists. If several choices exist, the earliest is selected (through calling the \( \text{find} \_\text{next} \) primitive). Then, the algorithm iteratively replaces \( u \) with the successor and searches for the next interval until the successor is \( v \) itself (Lines 5 to 9), adding gradually the corresponding contacts to a journey \( J \) (Line 4 and Line 7), which is ultimately returned in Line 10.

**Theorem 2** Algorithm 2 has time complexity \( O(k \log r) \), where \( k \) is the length of the resulting journey.

**Proof** The algorithm calls \( \text{find} \_\text{next} \) in Line 1. After that, it is known whether a journey can be reconstructed. If so, a journey prefix \( J \) is initialized with the first contact of the reconstructed journey (indeed, such a contact must exist due to the minimality of the interval). Then, in the loop from Line 5 to Line 9, the algorithm extends \( J \) by one contact for each call to \( \text{find} \_\text{next} \) until \( J \) contains the entire journey. Overall, \( \text{find} \_\text{next} \) is thus called as many times as the length of the reconstructed journey, which corresponds to \( O(|J| \log r) \) time. The costs of the other operations are clearly dominated by this cost.

4.3.1 Properties of the reconstructed journeys

In general, several journeys may exist that satisfy the query parameters. We observe that the specific choices made in Algorithm 2 imply additional properties.

**Lemma 4** The journey \( J \) which is returned by Algorithm 2 is a foremost journey in the requested interval (i.e., it arrives at the earliest possible time at \( v \)). Furthermore, among all the possible foremost journeys, it is also a fastest journey (i.e., the difference between departure time and arrival time is minimized).

**Proof** The fact that \( J \) is a foremost journey follows from the call to \( \text{find} \_\text{next} \) in Line 1. Indeed, the interval returned by this call corresponds to the earliest departure from \( u \), which happens to also correspond to the earliest arrival at \( v \) because the stored intervals are incomparable. \( J \) thus achieves the earliest possible arrival time at \( v \) in the given interval. And since all the stored intervals are minimal (i.e., they do not contain smaller reachability intervals), it also follows that \( \text{departure}(J) \) is as late as possible among all the journeys arriving in \( v \) at time \( \text{arrival}(J) \), which means \( J \) is as fast as possible among all foremost journeys.

Let us insist that Lemma 4 does not imply that \( J \) is both foremost and fastest in the requested interval. It only states that \( J \) is a foremost journey and a fastest one among the possible foremost journeys. Even faster journeys might exist in the requested interval, arriving later at \( v \). The above property is, however, already convenient, e.g., in communication networks, where a message would arrive at the destination as early as possible, while (secondarily) traveling for as little time as possible.

4.4 Evolution of the number of R-tuples over the insertions

As explained in Lemma 2, the worst-case asymptotic number of R-tuples in the data structure cannot exceed the maximum number of contacts. The data structure is worst-case optimal in this particular sense. However, many typical scenarios involve contacts between only a small fraction of the possible pairs of nodes, and only at some specific times. In order to understand, more globally, how the data structure behaves at various densities of contacts, we investigated the evolution of the number of R-tuples, as new contacts are inserted, using a simple randomized model of contact insertions. Namely, we start with an empty data structure, and
then, the contacts are drawn uniformly at random, without
replacement, among a large set of pairs of nodes \((n = 100)\)
and of time steps \((T = 100)\). The choice for these particular
values is somewhat arbitrary, but we chose sufficiently large
values to exhibit general phenomena that are discussed next.
The experiments were performed for two possible values of
the latency \(\delta\), namely 0 and 1, which account for the distinc-
tion between strict and non-strict journeys. The results, aver-
aged over 100 runs (in both cases), are shown in Figure 4 at
different scales: (a) early evolution, (b) intermediate evolution,
and (c) entire evolution.

Looking at the early evolution (Fig. 4a), one can see that,
at least for the first few insertions, the number of R-tuples
seems to grow linearly with the number of contacts. This
phenomenon can be explained by a simple argument: the
first few contacts are independent from each other, in the
sense that they share no vertices. As a result, these contacts
do not combine into non-trivial journeys. This regime will
start to change when the next contacts begin to interact with
the previous ones. Since every contact involves two vertices
and every vertex is picked at random from a set of \(n\) vertices,
the first non-trivial journey will be formed as soon as a ver-
tex is picked twice. By the birthday paradox (Birthday
2021), this will happen in expectation when
\[
f(n) = 1 + \sum_{k=1}^{n} \frac{n^k}{(n-k)!k!} \text{ vertices are drawn, which corre-
sponds to } f(n)/2 \text{ contacts. Then, as further contacts are being}
inserted, the contacts start combining with each other to
form many more non-trivial journeys, as illustrated in
Fig. 4b. When the number of inserted contacts reaches the
theoretical maximum (i.e., the temporal graph becomes satu-
rated), one can see in Fig. 4c that the number of R-tuples
converges to the number of contacts itself, as predicted (by
Lemma 2).

Between the last two regimes, the behavior is different
depending on whether \(\delta = 0\) or \(\delta = 1\). Interestingly, in the
case that \(\delta = 1\), the number of R-tuples does not increase
monotonically (see again Fig. 4c). A plausible explanation
for this phenomenon is as follows. When the contacts start
combining into non-trivial journeys, at first, almost none
of these journeys improve upon existing ones (i.e., the new
R-tuples do not replace existing ones, they just keep add-
ing to the data structure). Then, as the number of existing
R-tuples becomes huge, the insertion of a single new contact
may induce a trivial R-tuple that will replace many redu-
dant R-tuples (which are removed consequently). In the case
that \(\delta = 0\), the number of R-tuples reaches the theoretical
maximum much faster, which can be explained by the fact
that, as soon as a snapshot \(G_t\) becomes connected, then all
the pairs of nodes in the graph now have a minimal R-tuple
between them relative to time \(t\). The minimality of these
R-tuples also implies that these R-tuples will not be replaced
when direct contacts appear between these pairs.

In summary, the evolution of the number of R-tuples
seems to obey at least the following regimes, which are (1)
linear; (2) superlinear; (3) decreasing (for \(\delta = 1\)); and (4)
converging to saturation. There seems to be another phe-
nomenon occurring between regimes (2) and (3) in the case
of \(\delta = 1\), which would be interesting to investigate. In fact,
we formulate three open questions to guide further investiga-
tion of these phenomena, which we think are of independ-
ent interest as they pertain to general aspects of temporal
reachability.

**Open question 1** What is the worst-case theoretical growth
of the second regime, depicted in Fig. 4b? In particular, is
it polynomial?

**Open question 2** Is our explanation for the non-monotonic-
ity of the number of R-tuples sufficient? Otherwise, what
other mechanism explains it?

**Open question 3** What phenomenon explains the mild
inflexion that occurs in the case of \(\delta = 1\) (around 100000
insertions in Fig. 4c)?

5 Concluding remarks

We presented in this paper an incremental data structure to
solve the dynamic connectivity problem in temporal graphs.
Our data structure places a high priority on the query time,
by answering reachability questions in time \(O(\log \tau)\). Based
on the ability to retrieve reachability information for particu-
lar time intervals, it supports the insertion of contacts in a
non-chronological order in \(O(n^2 \log \tau)\) worst-case time and
makes it possible to reconstruct efficiently foremost journeys
within a given time interval, i.e., in time \(O(k \log \tau)\), where
\(k\) is the size of the resulting journey. Our algorithms exploit
the special features of non-redundant (minimal) reachabil-
ity information, which we represent through the concept of
R-tuples. The core of our data structure, namely the timed
transitive closure (TTC), is itself essentially a collection of
irredundant R-tuples, whose size (and that of the data struc-
ture itself) cannot exceed \(O(n^2 \tau)\).

The theory of R-tuples, initiated in this paper, poses a
number of further questions, some of which are of independ-
ent interest, some leading to possible improvements of the
presented algorithms. For example, do R-tuples involving
different pairs of vertices possess further interdependence
that may reduce the space needed to maintain TTCs? More
generally, how restricted are TTCs intrinsically? On the
practical side, can we improve the insertion time for new
contacts by using another low-level structure than a bal-
anced BST? Could the notion of contacts be generalized to
contacts of arbitrary duration? Finally, designing efficient
data structures for the decremental and the fully dynamic versions of this problem, with unsorted contact insertion and deletion, seems to represent both a significant challenge and a natural extension of the present work, one that would certainly develop further our common understanding of temporal reachability.

Acknowledgements This study was financed in part by Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG) and Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES)—Finance Code 001*—under the “CAPES PrInt program” awarded to the Computer Science Post-graduate Program of the Federal University of Uberlândia, as well as the Agence Nationale de la Recherche through ANR project ESTATE (ANR-16-CE25-0009-03).

References

Agrawal R, Borgida A, Jagadish HV (1989). Efficient management of transitive relationships in large data and knowledge bases, vol ’89. Association for Computing Machinery, New York, NY, USA, pp 253–262

Birthday problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Birthday_problem

Barjon M, Casteigts A, Chaumette S, Johnen C, Neggaz YM (arxiv: 1404.7634 (2014).) Testing temporal connectivity in sparse dynamic graphs. CoRR

Bedogni L, Fiore M, Glacet C (2018) Temporal reachability in vehicular networks. pp 81–89

Bleichloch GE, Ferizovic D, Sun Y (2016). Just join for parallel ordered sets, vol ’16. Association for Computing Machinery, New York, NY, USA, pp 253–264

Bryce D, Kambhampati S (2007) A tutorial on planning graph based reachability heuristics. AI Magaz 28(1):47

Cacciari, L., Rafiq, O.: A temporal reachability analysis. In: P. Dembusińska, M. Średzińska (eds.) Protocol Specification, Testing and Verification XV: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995, pp. 35–49. Springer US, Boston, MA (1996)

Casteigts A, Flocchini P, Quattrociocchi W, Santoro N (2012) Time-varying graphs and dynamic networks. Int J Parallel, Emergent Distr Syst 27(5):387–408

Cohen E, Halperin E, Kaplan H, Zwick U (2003) Reachability and distance queries via 2-hop labels. SIAM J Comput 32(5):1338–1355. https://doi.org/10.1137/S0097539702403098

Enright J, Meeks K, Mertzios GB, Zamarrae V (2021) Deleting edges to restrict the size of an epidemic in temporal networks. J Computer Syst Sci 119:60–77

George B, Kim S, Shekhar S (2007) Spatio-temporal network databases and routing algorithms: a summary of results. In: Papadias D, Zhang D, Kollios G (eds) Advances in Spatial and Temporal Databases. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 460–477

Haixun Wang, Hao He, Jun Yang, Yu, P.S., Yu, J.X.: Dual labeling: answering graph reachability queries in constant time. In: 22nd International Conference on Data Engineering (ICDE’06), pp. 75–75 (2006)

Hasan, K.T., Noori, S.R.H., Salam, A., Kabir, M.A.: Making sense of time: timeline visualization for public transport schedule. In: Symposium on Human-Computer Interaction and Information Retrieval (HCIR 2011), Washington (2011)

Hurter C, Ersoy O, Fabrikant SI, Klein TR, Telea AC (2014) Bundled visualization of dynamicgraph and trail data. IEEE Trans Visual Computer Gr 20(8):1141–1157

Latafy M, Viard T, Magnien C (2018) Stream graphs and link streams for the modeling of interactions over time. Soc Netw Anal Minig 8(1):1–29

Martenssen AC, Saura S, Fortin MJ (2017) Spatio-temporal connectivity: assessing the amount of reachable habitat in dynamic landscapes. Methods Ecol Evol 8(10):1253–1264

Mofitt VZ, Stoyanovich J (2016). arxiv:1602.00773.) Portal: A Query Language for Evolving Graphs. CoRR

Rozenstein P, Gionis A, Prakash BA, Vreeken J (2016). Reconstructing an epidemic over time, vol KDD ’16. Association for Computing Machinery, New York, NY, USA, pp 1835–1844

Seufert S, Anand A, Bedathur S, Weikum, G.: Ferrari, (2013) Flexible and efficient reachability range assignment for graph indexing. pp 1009–1020

Tang J, Musolesi M, Mascolo C, Latora V (2010) Characterising temporal distance and reachability in mobile and online social networks. SIGCOMM Comput Commun Rev 40(1):118–124

Wei H, Yu JX, Lu C, Jin R (2018) Reachability querying: an independent permutation labeling approach. VLDB J 27(1):1–26

Whitbeck, J., Dias de Amorim, M., Conan, V., Guillaume, J.L.: Temporal reachability graphs. In: Proceedings of the 18th Annual International Conference on Mobile Computing and Networking, Mobicom ’12, p. 377–388. Association for Computing Machinery, New York, NY, USA, 2012

Williams MJ, Musolesi M (2016) Spatio-temporal networks: reachability, centrality and robustness. R Soc Open Sci 3(6):160196

Wu G, Ding Y, Li Y, Bao J, Zheng Y, Luo J (2017) Mining spatio-temporal reachable regions over massive trajectory data. pp 1283–1294

Wu H, Huang Y, Cheng J, Li J, Ke Y (2016) Reachability and time-based path queries in temporal graphs. pp 145–156

Xiao, H. Aslay C, Gionis A (2018) Robust cascade reconstruction by Steiner tree sampling. pp 637–646

Xiao, H., Rozenstein, P., Tatti, N., Gionis, A.: Reconstructing a cascade from temporal observations, pp. 666–674

Xuan BB, Ferreira A, Jarry A (2003) Computing shortest, fastest, and time-varying graphs and dynamic networks. Int J Parallel, Emergent Distr Syst 27(5):387–408

Zeng W, Fu C, Arisona SM, Erath A, Qu H (2014) Visualizing mobility and efficiency: assessing the amount of reachable habitat in dynamic landscapes. Methods Ecol Evol 5(10):1253–1264

Zhu AD, Lin W, Wang S, Xiao X (2014). Reachability queries on large temporal graphs. pp 145–156

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.