Applied Network Science

RESEARCH Open Access

Temporal network compression via network hashing

Rémi Vaudaine[1], Pierre Borgnat[2], Paulo Gonçalves[1], Rémi Gribonval[1] and Márton Karsai[3,4*]

*Correspondence:
karsaim@ceu.edu

[1] Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, 69342 Lyon, France
[2] CNRS, Univ de Lyon, ENS de Lyon, Laboratoire de Physique, 69342 Lyon, France
[3] Department of Network and Data Science, Central European University, 1100 Vienna, Austria
[4] National Laboratory of Health Security, Rényi Institute of Mathematics, 1053 Budapest, Hungary

Abstract

Pairwise temporal interactions between entities can be represented as temporal networks, which code the propagation of processes such as epidemic spreading or information cascades, evolving on top of them. The largest outcome of these processes is directly linked to the structure of the underlying network. Indeed, a node of a network at a given time cannot affect more nodes in the future than it can reach via time-respecting paths. This set of nodes reachable from a source defines an out-component, which identification is costly. In this paper, we propose an efficient matrix algorithm to tackle this issue and show that it outperforms other state-of-the-art methods. Secondly, we propose a hashing framework to coarsen large temporal networks into smaller proxies on which out-components are more easily estimated, and then recombined to obtain the initial components. Our graph hashing solution has implications in privacy respecting representation of temporal networks.

Keywords: Temporal networks, Out-component calculation, Streaming matrix algorithms, Graph hashing

Introduction

While temporal networks represent the sequence of time-evolving interactions between entities, they also code the connected structure that lays behind many dynamical processes like the spreading of an epidemic or an information cascade or the collective adoption of behavioural norms or products. In static networks, connectivity is conventionally defined between two nodes if they are connected via a direct edge, or via a path building up from a sequence of adjacent edges that (pair-wise) share at least one node (Newman 2018). In temporal networks, however, connectedness is coded by *temporal* paths that are constructed from adjacent *temporal interactions*, which are not simultaneous yet structurally adjacent, and respect the causal time order. They determine the set of reachable nodes that can be influenced in the future with information held by a given node at a given time (Badie-Modiri et al. 2020; Holme and Saramäki 2012). The set of reachable nodes by a node at a given time, also called its influence set, is the node's temporal *out-component*, whose structure and size are important indicators of any ongoing dynamical processes. Indeed, no ongoing process can exhibit a larger collective pattern than the largest connected out-component in the underlying temporal network. However, the

characterisation of connected components in temporal networks is a computationally challenging task, as the temporal ordering of interactions introduces a degree of complexity to detect time-respecting paths in an effective way. Here, we address this challenge by defining a component matrix that codes the in- and out-component size of any node in a temporal network. Using this matrix we apply network compression and reconstruction techniques via graph hashing, to estimate the distribution of the size of connected components of nodes. The proposed algorithm provides advancements in the computation efficiency of the largest node components compared to the state-of-the-art, specifically for temporal networks with a large number of interactions.

Calculation of the largest out-component Considering all nodes and timed interactions in a temporal network, the most important component to characterise is, among other components, the largest out-component that ever emerged in the structure. Its identification can be approached using different ideas. A simple one would be to simulate a deterministic *Susceptible-Infected (SI) process* starting from every node at their first interaction time. In a deterministic SI process, nodes are either in a susceptible (S) or infected (I) state and a susceptible node certainly becomes infected when interacting with an infected one. It is a conventional model to describe the fastest spreading process in a network, where starting from a single seed node at its first appearance, the downstream set of infected nodes determines its maximum out-component. Using this method, in a temporal network of n nodes and m events, the computation of the out-component of a spreading seeded from a single source node, at its first appearance time would have $O(n)$ space and $O(m)$ time complexity (in terms of memory usage and computation time). This results in $O(n^2)$ space and $O(nm)$ time complexity when considering every node.

A more efficient method relies on a temporal *Event Graph* (EG), a higher-order representation of temporal networks (Mellor 2017; Kivelä et al. 2018; Badie-Modiri et al. 2020). An EG is a static and lossless representation of a temporal network in the form of a weighted and directed acyclic graph (DAG). In this structure, temporal interactions are associated to nodes that are linked if their corresponding events are adjacent. For a more precise definition see Sect. "Methods". Computing a single traversal of this static event graph (in reversed time order) yields the out-component of any node at any time, with an evidently smaller computational complexity as compared to a direct computation on a temporal network. However, EGs appear with considerably larger size (having as many nodes as events in the original temporal network) and higher link density (by connecting any events to all future adjacent others) that leads to increased memory complexity. In order to reduce memory complexity, a link reduction method has been proposed that eliminates path redundancy in the EG (Mellor 2017; Kivelä et al. 2018), leaving the connectedness of the DAG intact. Relying on the reduced EG, the use of the approximate HyperLogLog (HLL) counting algorithm can further reduce the time complexity of the out-component detection to $O(m \log(m) + \eta)$, where η is the number of edges of the EG. However, this method provides only an estimate of the size of out-components, without giving any information about their detailed structure.

Graph compression for component inference Contrary to earlier solutions, our idea is to use graph compression methods to compute the out-component size distribution of a temporal network, with a reduced computational complexity. The compressibility of

static networks has been studied recently (Lynn and Bassett 2021), and has been shown to depend on the structure of the graph. This notion can be extended for temporal networks by interpreting them as a sequence of time-aggregated static network snapshots. Then compression can be formulated as finding a smaller diffusion-equivalent representation (Adhikari et al. 2017). Also, consecutive snapshots can be compressed depending on their chronological importance (Allen et al. 2022). Moreover, as pointed out by Li and Sharpnack (2017), in spatio-temporal networks, nodes can be compressed via local clustering, while reducing time instants to change-points. Compression can be formulated using Minimum Description Length to also reduce the size of the graph (Liu et al. 2018). Another compression approach has been proposed using information theory considerations, aiming to reduce the number of bytes required to describe a temporal network (Liakos et al. 2022; Caro et al. 2016; Bernardo et al. 2013). Reducing the size of the network via coarsening to compute spectral properties of a graph has also been studied (Loukas and Vandergheynst 2018). Sampling techniques have been largely used to reduce the complexity of computation over large graphs (Yousuf and Kim 2020).

Despite these numerous compression techniques proposed for temporal networks, none of them reduces effectively the number of nodes in a series of events. This reduction has a huge impact on the computational complexity of any of these algorithms, especially when they are characterised by quadratic complexity in the number of nodes. Thus, our central question remains: *how to design an efficient compression scheme that reduces the number of nodes while keeping enough information about the network itself to reconstruct the statistics of its connected components?*

To reduce the computational complexity of the out-component size distribution calculation, we first propose a *online streaming matrix algorithm* that scans through the series of events only once, while it can also consider new events added later on, without re-starting the computation. In addition, we define a general purpose temporal network compression scheme using a graph hashing approach. This compression method reduces the total number of nodes, yet it requires a decompression scheme too, which provides only an approximate solution. The compression method can be used in conjunction with the matrix algorithm and, more generally, it can be applied on any temporal network algorithm.

To present our contributions, we organised the paper as follows. First, we formalise the problem of out-components computation in Sect. "Methods". We present the proposed novel streaming matrix algorithm to compute the distribution of the size of out-components in Sect. "Streaming matrix algorithm for out-component size calculations", including some numerical experiments. Then, we describe the hashing framework in Sect. "Hashing the temporal network", and we report also on the numerical studies carried out to evaluate its ability to estimate the ground-truth out-components' distributions in Sect. "Experimental evaluation". Finally, we discuss the proposed methods and the results.

Methods

The aim of the present work is to effectively compute the distribution of the maximum out-component size for all nodes in a temporal network. To establish our approach, we introduce first the definitions that are necessary to ground our methodology.

Problem definition

We define a temporal network $\mathcal{G} := (\mathcal{V}, \mathcal{E}, \mathcal{T})$ as a series of temporal events $e = (u, v, t) \in \mathcal{E}$ that record interactions between nodes $u, v \in \mathcal{V}$ at time steps t sampled[1] from a time period $\mathcal{T}$ of length T. The network $\mathcal{G}$ is characterised by its number of nodes $n = |\mathcal{V}|$ and its number of events $m = |\mathcal{E}|$. In $\mathcal{G}$ we call two events $e_i \in \mathcal{E}, e_j \in \mathcal{E}$ adjacent if they share at least one node ($\{u_i, v_i\} \cap \{u_j, v_j\} \neq \emptyset$) and their inter-event time is $\Delta t = t_j - t_i > 0$, i.e. the two events are not simultaneous. Furthermore, we call two events to be δt-adjacent if they are adjacent and their inter-event time is $\Delta t \leq \delta t$. A sequence of adjacent events defines a time respecting path between nodes u and v starting at time t, if the first event of the path starts from node u at time t, the last ends at node v, and each consecutive events in the sequence are pairwise adjacent (Holme and Saramäki 2012). The set of nodes that can be reached by any path starting from node u at time t defines the out-component. The size of the out-component of a node u at a given time t is measured as the number of unique nodes that can be reached by valid time respecting paths. Actually, it determines the largest possible phenonenon (e.g., largest epidemic or information cascade) that was initiated from that source node and evolved in the future. The computation of out-components is computationally challenging as it requires the tracking of each time-respecting path starting from each node at each time. However, an effective approximate solution has been proposed lately (Badie-Modiri et al. 2020) to solve a partial challenge, to estimate only the size of all out-components without keeping track of the involved nodes.

Event graphs and the HyperLogLog algorithm

The proposed solution builds on the Event Graph (EG) representation (Kivelä et al. 2018; Mellor 2017) of temporal networks. An event graph $G := (V, E, \Delta t)$ is defined as a static weighted directed acyclic graph (DAG) representation of a temporal network $\mathcal{G}$, where temporal events are associated to nodes in G (i.e., $V = \mathcal{E}$); directed edges in G correspond to Δt-adjacent event pairs in the original temporal network, with direction indicating their temporal order. The Δt weight of each link is defined as the inter-event time between the two adjacent events corresponding to the connected nodes in G. This way, an event graph has $m = |\mathcal{E}|$ vertices and η directed edges. This static graph representation provides a losless description of a temporal network and can be exploited to infer several properties of $\mathcal{G}$ without computations on the temporal structure (Kivelä et al. 2018). Indeed, thanks to the EG representation, the out-component size distribution of $\mathcal{G}$ can be precisely computed (Badie-Modiri et al. 2020), yet with high computational and memory costs.

To reduce this cost at the price of an inexact computation, Badie-Modiri et al. (2020) proposed an approximate solution to estimate the out-component size distribution of a temporal network using its EG representation combined with the HyperLogLog algorithm.

The HyperLogLog (HLL) algorithm takes as input a set, and it outputs an approximation of its size (Flajolet et al. 2007). More precisely, an HLL structure uses a

[1] In these definitions, we neglect the duration of events for simplicity, but all definitions could incorporate durations in a straightforward way.

representation on s registers each storing a number, initialised to zero to start with. Every element of the set is hashed into a binary vector that is then cut in two parts. The first part indicates the identifier of the register that will be used and the position of the leftmost 1 in the second part is stored in that register if it is larger than the current value. Finally, the size of the set is estimated with an ensemble indicator function based on the registers. The main advantage of the algorithm is that the whole set is not stored to estimate its size and the estimation can be done with constant space and time complexity $O(s)$. The error of the estimation is $O(1/\sqrt{s})$. Moreover, the size of the union of two sets can also be estimated in constant time and space by merging two HLL structures. A final property is that each element of the set is considered one by one, hence compatible with a streaming approach. Let us stress that the hash functions used in HLL are not related to the ones we will use in Sect. "Hashing the temporal network" to compress the network representation.

The HLL algorithm can be used to estimate out-component sizes in an EG without tracking the exact set of nodes involved (Badie-Modiri et al. 2020). This approach reduces the time complexity of the out-component distribution computation to $O(m\log(m)+\eta)$ up to some constant factors that depend on the hyper parameters s of the HLL algorithm, which sets the trade-off between computational efficiency and accuracy.

Streaming matrix algorithm for out-component size calculations

We develop a streaming matrix algorithm as an exact solution for the question of computing the largest out-component of each node in a temporal network. The proposed solution can process chronologically streamed nodes and events of a temporal network in real time, with a space complexity that does not depend on the number m of events.

To demonstrate the basic idea of the method, let us consider the simple example of an information spreading process on a temporal network between n nodes modelled by a deterministic SI process (a short definition is recalled in the Introduction). To follow-up on the evolving components during the SI process, we design a matrix with rows representing the in-component and columns representing the out-component of each node. At time $t=0$, when each node has a unique information that it has not propagated yet to any other nodes, we obtain the identity matrix with ones in the diagonal and zeros otherwise. Propagation happens between nodes u and v at the time of their interactions, when they mutually share all unique elements of information they already learned from others (including their own) during earlier times of the process.[2] This propagation rule is associated to the "OR" operation between the corresponding lines of the matrix, which yields the *union* of the set of unique information known by the two nodes. By the last event of the temporal network, the unique information of a node u is known by all other nodes in its out-component (depicted by column of the matrix). Thus, to compute the size of u's out-component, we simply have to count the number of unique nodes that are aware of u's unique information, i.e., the number of ones in the corresponding column of the matrix.

[2] Information sharing could be deemed non-mutual in case of directed interactions in the temporal network.

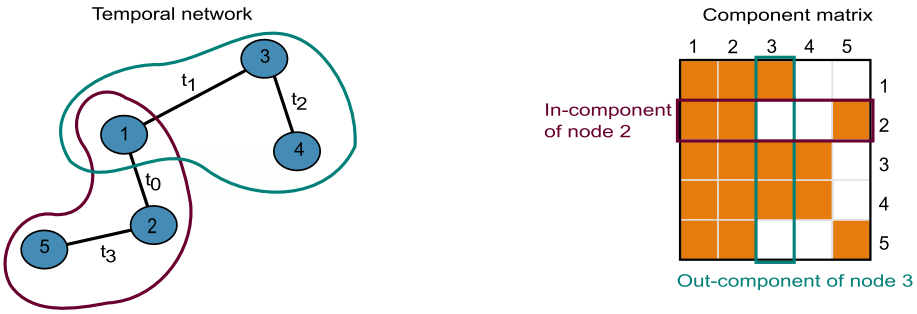

Fig. 1 Left: a temporal network defined as a series of events ordered in time ($t_i < t_j$ for $i < j$). The out-component of node 3 is circled in blue; the in-component of node 2 is circled in purple. Right: the corresponding component matrix. A row depicts the in-component of a node (we emphasise that of node 2). A column depicts the out-component of a node (we emphasise that of node 3): a non-zero element in the uth column at coordinate v, means that node v belongs to the out-component of node u

The component matrix

The component matrix is a binary matrix of size $n \times n$, where n is the number of nodes in $\mathcal{G}$. An illustration is provided in Fig. 1. An element (i, j) of the matrix is 1 if and only if the node i is reachable from the node j by any temporal path. Thus, the i-th line of the component matrix is the in-component of the node i and the j-th column of the matrix is the out-component of the node j.

Algorithm 1 The matrix algorithm. Calculates S

Require: Time ordered list of events E
$S \Leftarrow I_n$
for e in $\mathcal{E}$ **do**
  $u \Leftarrow e[0]$
  $v \Leftarrow e[1]$
  $r \Leftarrow S[u]\ OR\ S[v]$
  $S[u] \Leftarrow r$
  $S[v] \Leftarrow r$
end for

The precise algorithm to compute this component matrix is given as pseudo-code in Algorithm 1. It starts with the identity matrix. Then, for every event, the rows corresponding to the interacting nodes are used to compute a binary OR operation, and those rows are replaced by that resulting OR. Finally, at the end of the series of events, the output matrix is the component matrix. This algorithmic construction process is described in Fig. 2.

Complexity of the algorithm

Since we use an $n \times n$ matrix to store the intermediate results, the space complexity is $O(n^2)$, which may be reduced using sparse matrices for storage. For time complexity, we can divide the algorithm into several steps. The initialisation of the identity matrix can be done in $O(n)$ by simply setting the n diagonal elements to the "True" value at the outset. To update the matrix we perform the OR operation between two vectors of size

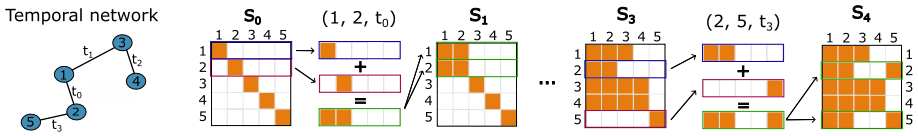

Fig. 2 From a given temporal network (left graph), we compute the Component Matrix (right matrix) of size $n \times n$, with $n = 5$, by scanning the series of events. For each event, we compute the OR operation between the rows of the matrix corresponding to the interacting nodes and replace them by the result. Matrix $\mathbf{S_4}$ is the component matrix at the end of the streaming after $m = 4$ events

n once for each of the m events. The complexity of each update is $O(n)$, bounded by the maximum out-component sizes n, but could be further reduced with a sparse matrix format. Consequently, the total complexity of the updates is $O(nm)$. Finally, counting the number of non-zero element (or non-"False" elements) can be done at the same time as the update without any added complexity. Thus, the overall time complexity of the component matrix algorithm is $O(n) + O(nm) = O(nm)$.

Streaming computation of average out-component size

One way to further reduce the complexity of the computation is to look for *an approximation* of the *average size* of a maximum out-component rather than its exact size. This can be implemented with the component matrix using the HyperLogLog counting algorithm that has been recalled before. It allows us to approximately describe and count the "True" values on each row of the matrix. The rows of the component matrix describe a set of nodes: the in-component. An HLL structure of size s (arbitrarily chosen, independently of n and m) can be used to estimate the size of an in-component. Thus, n HLL structures replace the former matrix. In its matrix form, the algorithm starts with the identity matrix. For the HLL structures, we simply initialise them with a single element: the i-th structure will be initialised with "i". Then, for every event (i, j, t), the OR operation between the lines i and j of the matrix is computed, which is equivalent to the union of the two in-components. For the HLL structures, this results in merging them. Finally, every HLL structure can give an approximation of the size of its corresponding line in the component matrix.

Interestingly, the average size of the maximum out-components at time t is defined as:

$$\bar{s}_t = \frac{1}{n} \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} S_t(u, v), \tag{1}$$

where the sums are interchangeable. Thus it can be computed both as the average size of out-components or in-components. Actually, the HyperLogLog structure can compute a size estimate with no additional cost in $O(1)$ time complexity for each in-component, which are coded in the matrix as the number of "Trues" in a row. According to Eq. 1, the average value of these maximum in-component size values can give us an estimate directly for the average of the maximum out-component sizes. Thus, the HyperLogLog approach can reduce the algorithm's space complexity to $O(n)$ and the time complexity to [3] $O(m)$. As an advantage, the matrix algorithm using HyperLogLog preserves the

[3] Remember, however, that the $O(\cdot)$ notation hides a constant, whose value results from a trade-off between cost and precision for the HLL algorithm.

streaming aspect of the algorithm, assuming events arrive in chronological order. However, in turn, it does not provide the whole maximum out-component size distribution but only an estimate of its mean value.

Component size distribution from reversed event sequence

By reversing time, we can easily obtain a solution to compute the whole maximum out-component size distribution. But it comes at the expense of losing the streaming property of our algorithm, as this solution takes as input, the whole interaction sequence in reversed order, processing it from the end to the beginning. By reversing the order of the sequence of events, the in-components become the out-components. In this case the component matrix algorithm does not fuse the rows anymore but it has to be adjusted the fuse the columns instead.

Thanks to the reversal of the sequence of events, we can use the HyperLogLog counting method to estimate the full distribution of the maximum out-component sizes at a lower cost.

More specifically, for every node, we initialise a HyperLogLog structure with constant size, which contains only the node itself as previously. Then, for every event (u_i, v_i, t_i), considered *in reverse chronological order*, we merge the structures of u_i and v_i (corresponding to columns, i.e., to current estimates of out-components) in $O(1)$ time. Finally, we approximate the size of the maximum out-component of every node with their HyperLogLog estimates. This results in an approximation of the whole distribution of out-components' sizes in $O(n)$ space complexity, $O(m)$ time complexity, and scanning the events' sequence only once. While this seems to be a very efficient solution, the constants in the complexity evaluations are quite large in practice, setting back the effective performance of this solution in some regimes of n and m, while providing better results in others, as we demonstrate in the next section.

As a summary, the first part of Table 1 reports the complexity and properties of the methods described so far. The second part of the table also anticipates on the method to be described in the next section.

Experimental validation

We perform several computational experiments to demonstrate the effectiveness of the component matrix algorithm and to compare its performance to the corresponding EG based solution. The code used in our experiments is freely available (Vaudaine et al. 2023).

Experimental setting

For a fair comparison, both the EG based and the component matrix based methods were used to solve the same task, that is to compute the largest maximum out-component size of a temporal network. While the EG based method solves a larger problem first, i.e. estimating the out-component size of any node *at any event*, one can extract the maximum out-component size for every node from its solution, simply by taking

Table 1 Summary of methods used to compute the maximum out-component size distribution

Method	Time cpx.	Space cpx.	Exact	Stream	OC	$P(\|OC\|)$
EG + HLL	$O(m\log(m)+\eta)$	$O(m+\eta)$	No	No	No	Whole
SI process	$O(mn)$	$O(n^2)$	Yes	No	Yes	Whole
Matrix	$O(mn)$	$O(n^2)$	Yes	Yes	Yes	Whole
Matrix + HLL	$O(m)$	$O(n)$	No	Yes	No	Average
Matrix + reverse t	$O(mn)$	$O(n^2)$	Yes	No	Yes	Whole
Matrix + reverse t + HLL	$O(m)$	$O(n)$	No	No	No	Whole
Matrix + hashing (//)	$O(mn_s)$	$O(n_s^2K)$	No	Yes	Yes	Whole
Matrix + hashing	$O(mn_sK)$	$O(n_s^2)$	No	Yes	Yes	Whole

Time and space complexity depends on the number of nodes n and events m, and number of edges η in the event graph (EG). Column entitled "Exact" indicates if the method provides exact (Yes) or approximate (No) solution. The column called "Stream" indicates if the method can stream events in chronological order. Column "OC" shows if the method can compute not only the out-component sizes but the involved nodes as well. The column "$P(|OC|)$" shows if the whole out-component distribution (Whole) or only its average (Average) can be computed. The hashing framework is described in the next section, with n_s number of super-nodes, K number of hash functions, and // indicating the possible parallelisable method. Note about the Matrix method: its space complexity is in $O(n^2)$ but can be reduced to $O(\bar{s}n\log n)$ where $\bar{s}$ is the average size of out-components in the case of sparse matrices

the size that corresponds to the first emergence of a given node. The overall asymptotic memory and time complexity of this solution scales similarly to the EG+HLL algorithm (Badie-Modiri et al. 2020), as it is summarised in Table 1. Taking this model as reference, we compare it with the performance of the proposed method based on the exact component matrix, as well as with its variant which uses HLL algorithms to obtain an approximation. In each method using HLL, we tune s to obtain less than 1% error for the average component size. Note that reversing time does not change neither the time nor the memory complexity of the component matrix algorithm with or without HLL as summarised in Table 1.

Results

To compare the different methods, we report in Fig. 3 the ratio of computation times and of memory usages between the compared algorithms. First, let us focus on the relative performance of the component matrix method in Fig. 3a and c. Interestingly, results depicted in panel (a) suggest that although this method provides an exact solution for the task, it performs always better than the EG+HLL algorithm in terms of computation time. A similar scaling is true in terms of memory complexity (panel (c)), although the large quadratic cost of the component matrix makes this method to perform worse than the reference for small numbers of events or for large numbers of nodes. Nevertheless, we can conclude that the component matrix method largely outperforms the event graph method in terms of computational time and memory for large numbers of events, especially with networks of smaller size, where the gain can reach several orders of magnitude.

Regarding the HLL variant of the component matrix algorithm shows more variable performance. In terms of computational time (see panel (b) in Fig. 3), although it is worse for small numbers of events and large networks, the performance of our HLL variant is comparable to that of the exact matrix algorithm, for the other parameter values. However, regarding memory consumption (see panel (d)), our HLL matrix variant

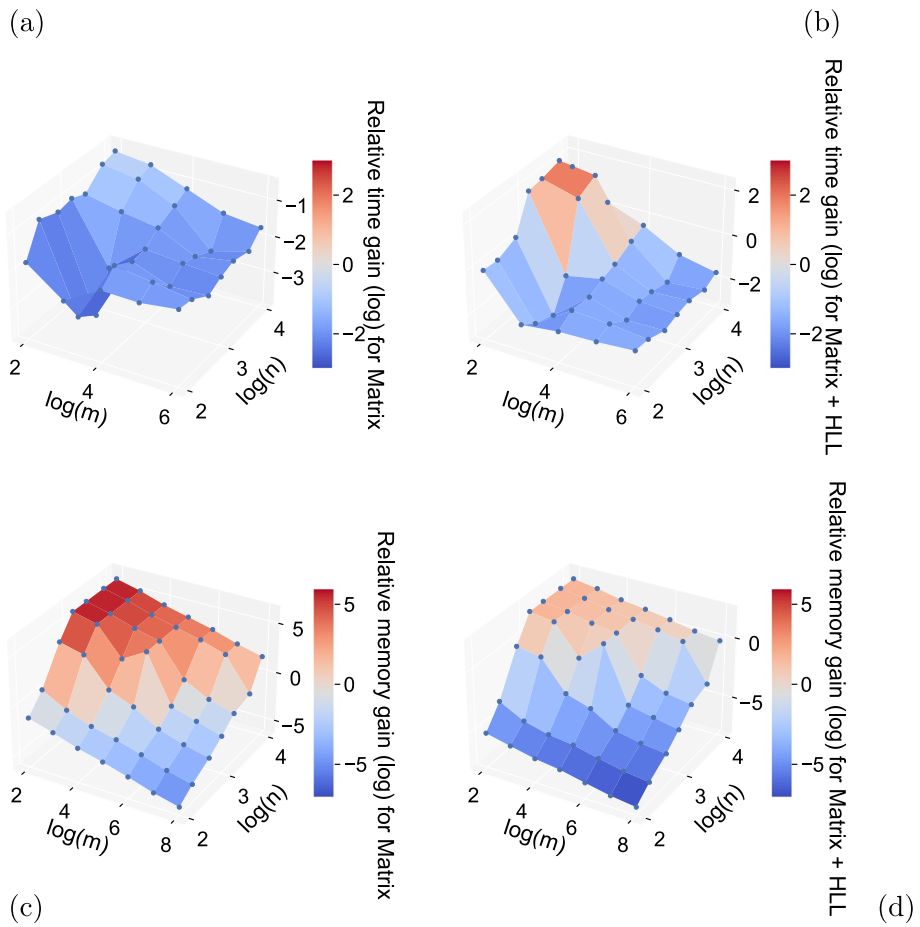

Fig. 3 Fraction of computational time (top row) and memory usage (bottom row) of the component matrix methods divided by the ones of the EG+HLL method. **a, c** Depict results for the exact component matrix method, while **b, d** are for its approximate solution using HLL. All scales are logarithmic with colour blue indicating when the component matrix method performs better (and red in the contrary case)

is much more efficient, as it does not have to store the component matrix of size n^2. For large network sizes the model requires approximately the same memory size as the reference model, while it is doing significantly better for the rest of the parameter space.

Advantages and limitations

As stated before, a major advantage of the component matrix algorithm as compared to other methods is its space complexity that does not depend on the number m of events in the temporal network but scales as the square of its node set size n. Meanwhile, its time complexity scales only linearly with m. This is especially suitable for data streaming scenarios when nodes and events arrive in chronological order. Actually, adding a new node to the network requires only to add a new row and column to the component matrix set as "False", except the diagonal element. As for new events, insertion follows the update rule discussed earlier, as the algorithm operates in a streaming manner. Furthermore, the component matrix method requires only one pass over the event sequence. At any time step t when a new event appears, it only requires information about the previous

Table 2 Computation times and memory usage for the EG + HLL, Matrix and Matrix + HLL methods for $n = 100$ and $m = 100$ (first and second line) and for for $n = 10^4$ and $m = 10^8$ (third and fourth line)

	EG + HLL	**Matrix**	**Matrix + HLL**
Time (ms)	40.3	0.4	27.4
Memory (kB)	1	80	0.92
Time (s)	30,172	700	811
Memory (MB)	200,000	800	0.085

state of the component matrix S_{i-1} at time $t-1$ (or conversely in the case of reversed time). On the other hand, the exact component matrix method scales poorly in space complexity in terms of n, the number of nodes, as it operates on an $n \times n$ matrix. This shortcut can be addressed by the HLL method to obtain approximate results. A sparse matrix implementation can also be very beneficial to solve this problem, if the average out-component size is much smaller than n. Otherwise, when it is comparable to n and the number of non-zero elements in the matrix is in $O(n^2)$, even a sparse matrix solution would scale quadratically.

Reference point

Figure 3 only gives the ratio between the computation times or the amount of memory required for the computations. We provide in Table 2 some concrete values for each method. The smallest temporal network on Fig. 3 is for $n = 100$ and $m = 100$. The largest temporal network in Fig. 3 is for $n = 10^4$ and $m = 10^8$. The associated computation times and memory usages are reported in Table 2.

Hashing the temporal network

Hashing the temporal network consists in reducing its number of nodes, thus compressing it, by (randomly) assigning nodes of the initial temporal network to "super-nodes" of a hashed graph. An event, or an interaction, between two nodes at time t in the initial temporal network becomes a new event between their hashed representatives in the hashed graph at time t. Reducing the number of nodes is notably attractive because this reduces the complexity of various different algorithms including the computation of the component matrix, even though this may cause information loss about the initial graph because of some conflicts. More precisely, one hash function defines a mapping from nodes to super-nodes, and that can lead to a loss of information due to node collisions. To balance this effect, we propose to use several different hash functions and to fuse the obtained results together. Indeed, using multiple hash functions reduces such conflicts as it is very unlikely that two nodes are always merged in the same super node. The method is taking advantage of that. The overall framework is shown in Fig. 4.

Hash functions

To reduce the number of nodes of the static graph underlying the input temporal graph (in short: "the input static graph"), and therefore the computation complexity of the out-component size distribution, we use hash functions. These functions take

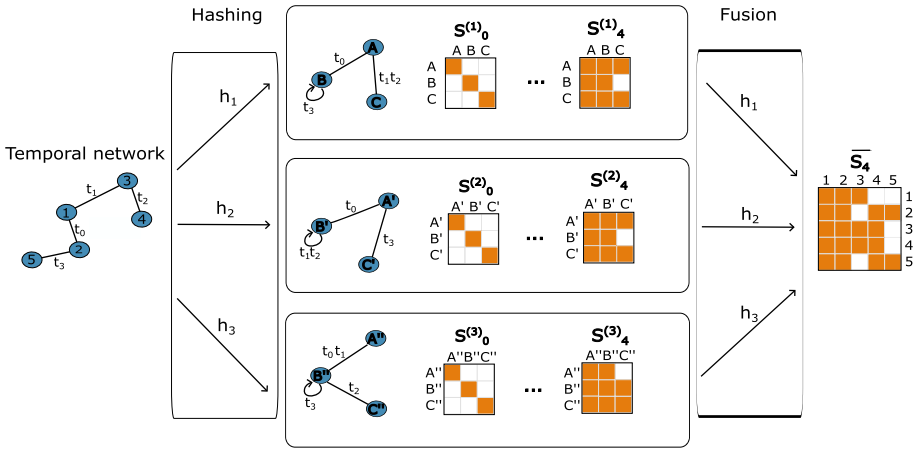

Fig. 4 From a 5 nodes temporal network, several hashed version are computed with 3 nodes each. Then, every hashed graph can be used to compute a small component matrix thanks to our matrix algorithm. Finally, the different component matrices can be fused to compute an approximate solution of the component matrix of the initial temporal network

as input a set of labels of n nodes, $\{1, \ldots, n\} = [n]$, and hash them into n_s super-nodes $\{1, \ldots, n_s\} = [n_s]$. The labels are the nodes of the input static graph and the buckets are the super-nodes of the resulting hashed static graph. Since $n_s < n$, some nodes will collide into the same super-node, reducing the overall cost of computation over the hashed temporal network associated to the hashed static graph, but reducing also the amount of available information.

There are many ways to design hash functions. In this work, we rely on classical approaches and we use existing universal hash functions as they ensure a low number of collisions whatever the original data. Indeed, the probability of this occurrence is controlled by the parameter k. We choose $k = 4$, first because, for this value of k, there exists an analytical family of hash functions that are easily computable and second because this value also ensures a reasonably small probability of collisions. A great advantage of using these hash functions is that the user does not have to design them explicitly depending on his goal. The functions are, by definition, blind to the structure of the network. Thus, the choice of the hash functions does not depend on the structure of the network. In fact, we draw multiple hash functions from the same class and use them regardless of the structure of the graph.

We use k-universal (randomised) hash functions (Thorup and Zhang 2004). A class H of random functions is k-universal if $\forall x_1, \ldots, x_k \in [n], \forall v_1, \ldots, v_k \in [n_s]$,

$$\Pr\{h(x_i) = v_i, \forall i \in [k]\} = 1/n_s^k \tag{2}$$

where the probability is on the draw of h. Qualitatively, this means that the probability that one node of the initial graph is assigned to the same super-node by two different hash functions is low and controlled by the choice of k.

In our work, we use $k = 4$ and the hash functions are based on a large prime number: $Prime = 2^{61} - 1$. Their computation is divided into four steps. First, let us define a table A of size (3, *order*), where *order* is an order parameter defined as:

$$\begin{aligned} &\forall i \in \{0,1,2\}, \forall j \in [order], \\ A(i,j) &\equiv ((rand(2^{31}-1) \ll 32) + rand(2^{31}-1)) \mod Prime \end{aligned} \tag{3}$$

where $\ll 32$ denotes the shift of the binary representation 32 bits to the left.

Second, the number *acc* is computed recursively $order - 1$ times thanks to:

$$acc(i,u) \equiv MultAddMod(u, acc(i,u), A(i,j)) \mod Prime \tag{4}$$

where $j \in \{1, \ldots, order\}$, *MultAddMod* is a function defined in the paper and initially, $acc(i,u) = A(i,0)$.

Third, T_0, T_1, T_2 are tables of size n_s that are defined as:

$$\forall u \in [n_s], T_i[u] = acc(i,u) \tag{5}$$

Fourth, for every node, we define three quantities x_0, x_1, x_2 as $\forall u \in [n], x_0(u) = low(u), x_1(u) = high(u), x_2(u) = x_0(u) + x_1(u)$ where $low(u)$ outputs the 32 rightmost bits of the binary representation of u and $high(u)$ outputs the 32 leftmost bits. Finally,

$$\forall u \in [n], h(u) = T_0(x_0(u)) \star T_1(x_1(u)) \star T_2(x_2(u)) \tag{6}$$

where $\star$ is the bitwise exclusive OR.

The hashed static graph is made of super-nodes defined by the output of a hash function and of "super"-edges connecting them: if u and v are connected in the initial static graph, then the super-nodes $h(u)$ and $h(v)$ become connected by a super-edge, whose weight is binary. Finally, for every event (u, v, t), a super-event is defined as $(h(u), h(v), t)$.

Fusion to compute the distribution of out-components

The main goal of our work is to compute the out-component, or its size, of every node in the input temporal network with lower complexity than existing methods reported in Table 1. To do so, we hash the set of n nodes of the input temporal network into n_s super-nodes with K different hash functions h_j.

$$\forall i \in [n] \; \forall j \in [K], h_j(u_i) \in [n_s] \tag{7}$$

These hash functions are drawn independently at random.

Here, the hash functions $h_j, \forall j \in [K]$ are not injective thus not invertible: there are usually several nodes mapped to the same super-node. We define the inverse of h as the function that, given a super-node of the hashed static graph, computes the set of corresponding nodes in the initial static graph:

$$\forall v \in [n_s], \; h^{-1}(v) = \{u \in [n] / h(u) = v\} \tag{8}$$

Denote $\mathrm{OC}(u)$ (resp. $\mathrm{OC}(h(u))$) the out-component of node u (resp. super-node $v = h(u)$). Assuming we can compute (an estimate of) the out component $\mathrm{OC}(v)$ of a super-node v in the hashed graph obtained with hash function h_j, we can also define

$$h_j^{-1}(\mathrm{OC}(v)) = \bigcup_{x \in \mathrm{OC}(v)} h^{-1}(x) \tag{9}$$

Instead of estimating the out-component for each of the n nodes in the temporal network, we first hash the network into K hashed graphs of n_s nodes and m events, then estimate the out-component for every node in the hashed graphs and finally aggregate the information by intersecting the (estimated) out-components given by each hashed graph. We then define

$$\forall i \in [n], \widehat{\mathrm{OC}}(u_i) = \bigcap_{j\in[K]} h_j^{-1}(\mathrm{OC}(h_j(u_i))). \tag{10}$$

The estimated out-component necessarily contains the true out-component, i.e. $\mathrm{OC}(u_i) \subseteq \widehat{\mathrm{OC}}(u_i)$, yet if the number K of hash functions is too small the set $\widehat{\mathrm{OC}}(u_i)$ may be much larger than the true out-component. Computing $|\widehat{\mathrm{OC}}(u_i)|$, where $|A|$ is the number of elements of a set A, one can compute an approximation of the distribution of the out-components' sizes with any of the aforementioned algorithm that is able to compute the out-components (*and not only their sizes*) on the hashed graphs. We compare the resulting approximate distribution with the true distribution.

Properties of the algorithm

The structure of the resulting algorithm ensures that every step before the final fusion remains compatible with streamed events arriving in chronological order, and is also amenable to parallel/independent computations for each hash function.

Moreover, the complexity of the framework depends on the setup. In a parallel setting, i.e. when the $S_i^{(k)}$ are computed separately, we need $O(K \times n_s^2)$ space to store the matrices and $O(mn_s)$ time to compute the small component matrices. In a non-parallel setting, we need $O(K \times n_s^2)$ to store the small matrices and $O(Kmn_s)$ time to compute them.

Experimental evaluation

The compression framework that we propose can be used with several observables. Here, we focused on the computation of the out-components. The whole distribution of the size of the out-components describes the largest spreading phenomena possible starting from every node. The other quantity we are interested in is the tail of the distribution, i.e. the set of nodes with largest out-component' size. To experimentally prove the effectiveness of our work, we measure the precision of the approximate method with respect to the ground-truth for both the distribution and its tail.

Experimental setting

For simulations, temporal networks are generated exactly as in the previous Section, see Sect. "Component size distribution from reversed event sequence". In the generated data, the number of nodes, n, varies between $\{100, 200, 500, 1000, 2000, 5000, 10\,000\}$ and the number of events varies from 10^4 to 10^9 as powers of 10. The number of supernodes is always a fraction of the number of nodes: $n_s = 0.3 \times n$ and the number of hash functions is $K = 5$.

The baseline algorithm is our matrix method from the previous section since it provides the exact distribution of the size of the out-components, with a controlled memory and time complexity. The results will compare the hashing version to this baseline.

In addition, some experiments have been conducted on real-world datasets freely available in Snap. The "Superuser" temporal network is a network of interactions on the stack exchange web site Super User. There are three kinds of interactions (edges): a user (node) answered a question of someone else, a user commented a question or a user commented an answer. The Superuser network is made of 194,085 nodes and 1,443,339 events. The "Reddit" dataset is a temporal network of connections between subreddits (i.e., forum dedicated to a specific topic on the website Reddit), taken here as nodes. There is an event between two subreddits when there is a post from a source subreddit that links to a target subreddit. The Reddit temporal network has 35,776 nodes and 286,560 events.

For real datasets, we split chronologically the events in 10 equal parts and compute the distribution of the largest out-component on $\{10\%, 20\%, \ldots, 100\%\}$ of the events. As for generated datasets, we use $n_s = 0.3 \times n$. We also use $K = 1$ and $K = 5$. The baseline is still the matrix algorithm of the previous section.

Performance criteria

We evaluate the hashing framework based on three criteria: time, memory and accuracy. We compare the time required by the matrix algorithm to compute the true distribution of the largest out-components, $\mathcal{D}$ with the one required by the hashing framework to compute the approximate distribution of the largest out-components $\mathcal{D}_s$. The computation time of the hashing framework includes both the computation of the hashed matrices and their fusion.

We also compare the memory usage of the matrix algorithm, with a single big matrix, with the one of the hashing framework, with several smaller matrices.

Furthermore, to compare the ground-truth out-component' size distribution $\mathcal{D}$ and the one computed thanks to the hashing framework $\mathcal{D}_s(n_s, K)$, we simply use the Earth-Mover distance, also called Wasserstein distance (Bonneel et al. 2011), computed thanks to the Python Optimal Transport library (Flamary et al. 2021). Among the other possible distances, we tried the Kullback-Leibler divergence, which proved less sensitive to subtle differences between the distributions. Thus we can define accuracy of the out-component size distribution inference as:

$$Acc(n_s, K) = \gamma(\mathcal{D}, \mathcal{D}_s(n_s, K)) \tag{11}$$

where γ is the Earth-Mover distance. The lower $Acc(n_s, K)$, the closest are the two distributions.

Results

First, we present the results for the generated data. The relative computation time, relative memory usage and accuracy of the hashing framework are reported in Fig. 5. The relative computation time figure is red meaning that the hashing framework requires more time than the matrix method to compute the target distribution. However, we can clearly see that the relative computation time decreases quickly with the number of events and slowly with the number of nodes. Generally, for datasets with more events than $m = 10^8$, the hashing framework with $K = 5$ and $n_s = 0.3 \times n$ requires less time than the full matrix method.

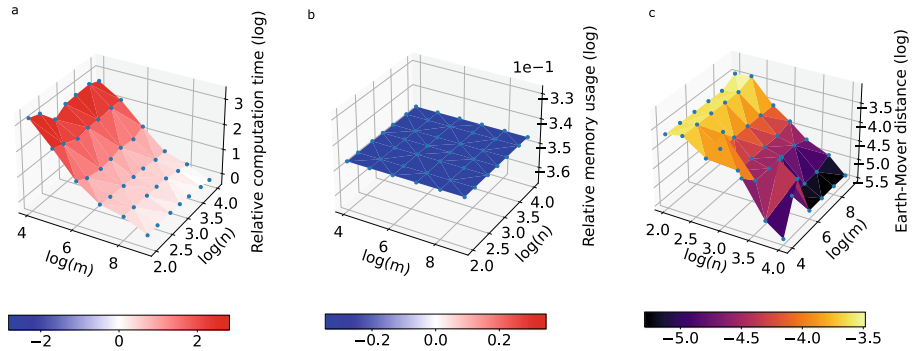

Fig. 5 **a** Relative time and **b** relative memory usage for the computation of the distribution of the largest out-components compared to the ground-truth given by the matrix method on synthetic data. **c** Accuracy of the hashing framework given by the Earth-Mover distance

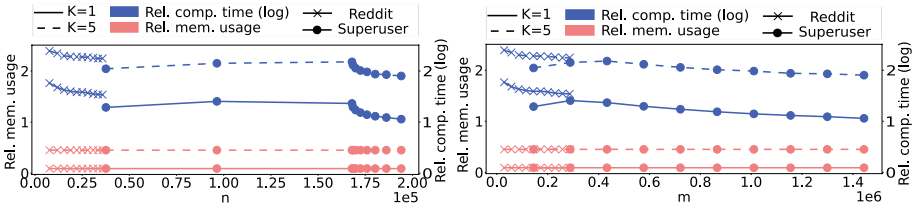

Fig. 6 Relative computation time and relative memory usage for the Reddit and Superuser datasets to compute the distribution of the largest out-components thanks to the hashing framework compared to the matrix

For the relative memory usage, the figure is blue meaning that we always gain memory. In fact, in this setup, only half the memory of the matrix method is required for the hashing framework.

As for the Earth-Mover distance, there is a regime for small datasets where the accuracy is not satisfactory but for the large majority of the generated datasets, the hashing framework performs very well.

For the real datasets, we first show the results of the relative computation time and the relative memory usage of the hashing framework compared to the matrix method in Fig. 6. Experimentally, we show that the hashing framework generally requires more time to compute the target distribution. Moreover, the relative computation time is linear with the number of hash functions. That is, for $K = 5$, that time is approximately 5 times higher than for $K = 1$ for both the Reddit dataset and the Superuser dataset. Overall, the general shape of the curves is in line with the results on generated data. For example, the relative computation time for the third point of the Reddit dataset, $n = 15{,}370$ and $m = 85{,}968$, is 198, which coincides with the corresponding value on the generated datasets. Overall, the relative computation time decreases as the number of nodes increases, as expected.

Then, the memory required for the computation is linear with the number of hash functions. We clearly see that, for $K = 5$, the memory usage is 5 times more than the one for $K = 1$. The figures for real datasets are also in line with the figures for generated datasets. Obviously, with $K = 1$ and $n_s = 0.3 \times n$, the computation requires less memory than with the full matrix algorithm for both real datasets by a factor 10. But, more

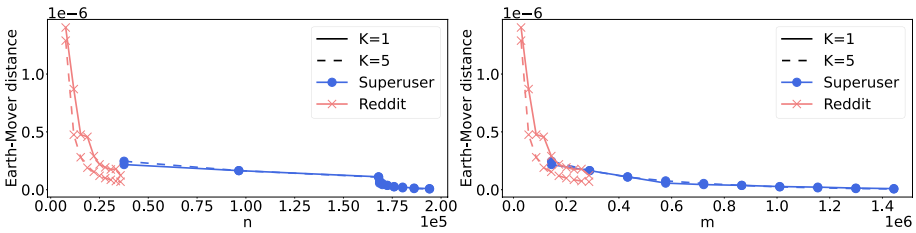

Fig. 7 Accuracy for the Reddit dataset and the Superuser dataset to compute the distribution of the largest out-components thanks to the hashing framework compared to the matrix method

importantly, with $K = 5$ and $n_s = 0.3 \times n$, the hashing framework still requires only around 50% of the memory of the matrix method.

Finally, we report the accuracy of the hashing framework compared to the matrix method in terms of the Earth-Mover distance between the true distribution computed by the matrix method and the one estimated by the hashing method in Fig. 7. Indeed, the quality of the results is important to assess the quality of the hashing framework. For both datasets, lower dimensions lead to lower accuracy of the framework. We see that the first few points, corresponding to networks of small sizes, have a significantly higher Earth-Mover distance (thus, lower accuracy) than the remaining ones. Overall, the shape of the curves still confirms the results with the generated datasets: the larger the network, the better the approximation. Secondly, as expected, the distance is lower for higher values of K. The accuracy of the method increases as there are more hash functions.

Thus overall we can conclude that hashing is relevant in high dimension. There is a computation time gain for $m \geq 10^9$ in the generated datasets while memory usage remains lower and accuracy is good. Also, increasing the number of hashes leads to a linear increase in the memory usage and a linear increase in the computation time. Obviously, this increases the accuracy of the method.

Conclusion

The continuous growth in the size of data bases requires new algorithms to process information. Moreover, structured data evolving over time represents an important challenge since it differs a lot from usual tabular data. To that end, we proposed a matrix algorithm that is able to compute both the out-components for every node of a temporal network and their sizes. Furthermore, to reduce the complexity of the analysis, we proposed a compression scheme based on hash functions that reduces the number of nodes of the network at the cost of some uncertainty. Uncertainty is lifted thanks to the use of several hashes in parallel. On each hashed graph, the matrix algorithm can be computed and, finally, all the information is merged to approximate the component matrix of the input network. Our framework is online and allows parallelization. Indeed, new nodes and news events can be processed as they come. Moreover, the different hashed graphs allow parallelization since they are independent. Additionally, hashing can make the computation private. If we do not observe the temporal network directly but only hashed versions of it and if hashes have some external randomness, our framework allows ϵ-differential privacy (Dwork et al. 2006). By construction, hashing loses some information during the compression process. When

the hashed graph is known to belong to a very small family of graphs, it is sometimes possible to reconstruct it exactly from its hashed version using this knowledge, but in general hashing is not designed for exact computations or any lossless calculations (of causal temporal paths for example). Yet, for a large set of problems, exact computations may not be necessary. One example is the computation of the upper-bound of the size of any dynamical process. In the context of temporal network analysis, the scope of applicability of hashing is still under consideration by the research community. Here, we demonstrate that it can be applied to compute an approximation of the out-components in a temporal network. This method can be particularly useful in the contexts of network reduction, privacy and spreading on networks. Moreover, hashing allows to characterize some properties of the network without knowing the whole data, only its hashed version. More specifically, we believe that our work has a lot of potential applications. The first concrete user case is to use out-component sizes as the maximum number of nodes reachable during a spreading process. For example, it can be the maximum number of people infected by a virus from a single source. Or, on Twitter, it can be the maximum number of people a piece of news spreads to. Secondly, our framework can be extended to other cases. In our work, we focused on out-components but we believe that many other quantities can be computed thanks to our compression scheme such as pairwise distances between nodes. Also, we believe that the hashing framework can be rewritten with an algebraic formulation. This would open up the work to linear problems and linear solvers. In fact, the reconstruction of the matrix could be tackled in many different ways making the framework more flexible. Moreover, privacy preserving algorithms are particularly interesting for security or privacy reasons. The work we propose can efficiently make algorithms on temporal networks private. Indeed, adding randomness in the data can lead to prevent the identification of the source of the data. Most importantly, our hashing framework transforms a temporal network into a series of smaller datasets that can be used to infer properties of the initial dataset without direct access to it. This can be very beneficial in the processing of sensitive information.

Funding

This work was supported by the DATAREDUX project, ANR19-CE46-0008. MK was supported by the CHIST-ERA project SAI: FWF I 5205-N; the SoBigData++ H2020-871042; the EMOMAP CIVICA projects; and the National Laboratory for Health Security, Alfréd Rényi Institute, RRF-2.3.1-21-2022-00006. PB was supported by the CHIST-ERA-19-XAI-006, for the GRAPHNEX ANR-21-CHR4-0009 project.

Availability of data and materials

The datasets used in this article are freely available on their corresponding website indicated in the main text. The code developed for this article is available on this link.

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 11 September 2023 Accepted: 22 December 2023
Published online: 17 January 2024

References

Adhikari B, Zhang Y, Bharadwaj A, Prakash B (2017) Condensing temporal networks using propagation, pp 417–425. https://doi.org/10.1137/1.9781611974973.47

Allen AJ, Moore C, Hébert-Dufresne L (2022) A network compression approach for quantifying the importance of temporal contact chronology. https://doi.org/10.48550/ARXIV.2205.11566. arXiv:2205.11566

Badie-Modiri A, Karsai M, Kivelä M (2020) Efficient limited-time reachability estimation in temporal networks. Phys Rev E 101:052303. https://doi.org/10.1103/PhysRevE.101.052303
Bernardo GD, Brisaboa NR, Caro D, Rodríguez MA (2013) Compact data structures for temporal graphs. In: 2013 data compression conference, p 477. https://doi.org/10.1109/DCC.2013.59
Bonneel N, Panne M, Paris S, Heidrich W (2011) Displacement interpolation using Lagrangian mass transport. ACM Trans Graph 30(6):158. https://doi.org/10.1145/2070781.2024192
Caro D, Rodríguez MA, Brisaboa NR, Fariña A (2016) Compressed kd-tree for temporal graphs. Knowl Inf Syst 49(2):553–595. https://doi.org/10.1007/s10115-015-0908-6
Dwork C, McSherry F, Nissim K, Smith A (2006) Calibrating noise to sensitivity in private data analysis. In: Halevi S, Rabin T (eds) Theory of cryptography. Springer, Berlin, pp 265–284
Flajolet P, Fusy, Gandouet O, Meunier F (2007) Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In: AOFA '07: proceedings of the 2007 international conference on analysis of algorithms
Flamary R, Courty N, Gramfort A, Alaya MZ, Boisbunon A, Chambon S, Chapel L, Corenflos A, Fatras K, Fournier N, Gautheron L, Gayraud NTH, Janati H, Rakotomamonjy A, Redko I, Rolet A, Schutz A, Seguy V, Sutherland DJ, Tavenard R, Tong A, Vayer T (2021) POT: python optimal transport. J Mach Learn Res 22(78):1–8
Holme P, Saramäki J (2012) Temporal networks. Phys Rep 519(3):97–125. https://doi.org/10.1016/j.physrep.2012.03.001
Kivelä M, Cambe J, Saramäki J, Karsai M (2018) Mapping temporal-network percolation to weighted, static event graphs. Sci Rep 8(1):12357
Liakos P, Papakonstantinopoulou K, Stefou T, Delis A (2022) On compressing temporal graphs. In: 2022 IEEE 38th international conference on data engineering (ICDE), pp 1301–1313. https://doi.org/10.1109/ICDE53745.2022.00102
Li X, Sharpnack J (2017) Compression of spatio-temporal networks via point-to-point process models. In: Proceedings of the 13th international workshop on mining and learning with graphs (MLG)
Liu Y, Safavi T, Shah N, Koutra D (2018) Reducing large graphs to small supergraphs: a unified approach. Soc Netw Anal Min. https://doi.org/10.1007/s13278-018-0491-4
Loukas A, Vandergheynst P (2018) Spectrally approximating large graphs with smaller graphs. In: International conference on machine learning
Lynn CW, Bassett DS (2021) Quantifying the compressibility of complex networks. Proc Natl Acad Sci 118(32):2023473118. https://doi.org/10.1073/pnas.2023473118
Mellor A (2017) The temporal event graph. J Complex Netw 6(4):639–659. https://doi.org/10.1093/comnet/cnx048
Newman M (2018) Networks. Oxford University Press
Thorup M, Zhang Y (2004) Tabulation based 4-universal hashing with applications to second moment estimation. In: Proceedings of the fifteenth annual ACM-SIAM symposium on discrete algorithms. SODA '04. Society for Industrial and Applied Mathematics, USA, pp 615–624
Vaudaine R, Gribonval R, Gonçalves P, Borgnat P, Karsai M (2023) Code for the article temporal network compression via network hashing. https://inria.hal.science/hal-04330187
Yousuf M, Kim S (2020) Guided sampling for large graphs. Data Min Knowl Discov. https://doi.org/10.1007/s10618-020-00683-y

Publisher's Note