Knowledge and Information Systems (2020) 62:1611–1639
https://doi.org/10.1007/s10115-019-01403-9

REGULAR PAPER

# Finding events in temporal networks: segmentation meets densest subgraph discovery

**Polina Rozenshtein[1,7] · Francesco Bonchi[2,3] · Aristides Gionis[1] · Mauro Sozio[4] · Nikolaj Tatti[5,6]**

Received: 5 January 2019 / Revised: 4 September 2019 / Accepted: 13 September 2019 /
Published online: 3 October 2019


## Abstract

In this paper, we study the problem of discovering a timeline of events in a temporal network. We model events as dense subgraphs that occur within intervals of network activity. We formulate the event discovery task as an optimization problem, where we search for a partition of the network timeline into $k$ non-overlapping intervals, such that the intervals span subgraphs with maximum total density. The output is a sequence of dense subgraphs along with corresponding time intervals, capturing the most interesting events during the network lifetime. A naïve solution to our optimization problem has polynomial but prohibitively high running time. We adapt existing recent work on dynamic densest subgraph discovery and approximate dynamic programming to design a fast approximation algorithm. Next, to ensure richer structure, we adjust the problem formulation to encourage coverage of a larger set of nodes. This problem is **NP**-hard; however, we show that on static graphs a simple greedy algorithm leads to approximate solution due to submodularity. We extend this greedy approach for temporal networks, but we lose the approximation guarantee in the process. Finally, we demonstrate empirically that our algorithms recover solutions with good quality.

**Keywords** Densest subgraph · Segmentation · Dynamic programming · Approximate algorithm

✉ Polina Rozenshtein
polina.rozenshtein@aalto.fi

1 Department of Computer Science, Aalto University, Espoo, Finland

2 ISI Foundation, Turin, Italy

3 Eurecat, Barcelona, Spain

4 Telecom ParisTech University, Paris, France

5 F-Secure Corporation, Helsinki, Finland

6 University of Helsinki, Helsinki, Finland

7 Nordea Data Science Lab, Helsinki, Finland

## 1 Introduction

Real-world networks are highly dynamic in nature, with new relations (edges) being continuously established among entities (nodes) and old relations being broken. Analyzing the temporal dimension of networks can provide valuable insights about their structure and function; for instance, it can reveal temporal patterns, concept drift, periodicity, temporal events, etc. In this paper, we focus on the problem of *finding dense subgraphs*, a fundamental graph-mining primitive. Applications include community detection in social networks [16,18,48], gene expression and drug interaction analysis in bioinformatics [22,45], graph compression and summarization [21,30,32], spam and security threat detection [13,26], and more.

When working with temporal networks, one has first to define how to deal with the temporal dimension, i.e., how to identify which are the temporal intervals in which the dense structures should be sought. Instead of defining those intervals a priori, in this paper *we study the problem of automatically identifying the intervals that provide the most interesting structures*. We consider a subgraph interesting if it boasts high density. As a result, we are able to discover a sequence of dense subgraphs in the temporal network, capturing the evolution of interesting events that occur during the network lifetime. As a concrete example, consider the problem of *story identification* in online social media [3,8]: The main goal is to automatically discover emerging stories by finding dense subgraphs induced by some entities, such as twitter hashtags, co-occurring in a social media stream.

In our case, we are also interested in finding different stories over the network lifetime. For instance, as one story wanes and another one emerges, one dense subgraph among entities dissipates and another one appears. Thus, by segmenting the timeline of the temporal network into intervals, and identifying dense subgraphs in each interval, we can capture the evolution and progression of the main stories over time. As another example, consider a collaboration network, where a sequence of dense subgraphs in the network can reveal information about the main trends and topics over time, along with the corresponding time intervals.

**Challenges and contributions** The problem of finding the $k$ densest subgraphs in a static graph has been considered in the literature from different perspectives. One natural idea is to iteratively (and greedily) find and remove the densest subgraphs [49], which unfortunately does not provide any theoretical guarantee. More recent works study the problem of finding $k$ densest graphs with limited overlap, while they provide theoretical guarantees in some cases of interest [7,24]. However, these approaches do not generalize to temporal networks.

For temporal networks, to our knowledge, there are only few papers that consider the task of finding temporally coherent densest subgraphs. The most similar to our work aims at finding a heavy subgraph present in all, or $k$, snapshots [46]. Another related work focuses on finding a dense subgraph covered by $k$ scattered intervals in a temporal network [44]. Both methods, however, focus on finding a single densest subgraph.

In this paper, instead, we aim at producing a partition of the temporal network that $(i)$ it captures dense structures in the network; $(ii)$ it exhibits temporal cohesion; and $(iii)$ it is amenable to direct inspection and temporal interpretation. To accomplish our objective, we formulate the problem of $k$- DENSEST- EPISODES (Sect. 2), which requires to find a partition of the temporal domain into $k$ non-overlapping intervals, such that the intervals span subgraphs with maximum total density. The output is a sequence of dense subgraphs along with corresponding time intervals, capturing the most interesting events during the network lifetime.

For example, consider a simple temporal network shown in Fig. 1. It consists of five nodes $\{A, B, C, D, E\}$, which interact at six different time stamps $(1, 2, 4, 5, 7, 8, 10)$. Our goal

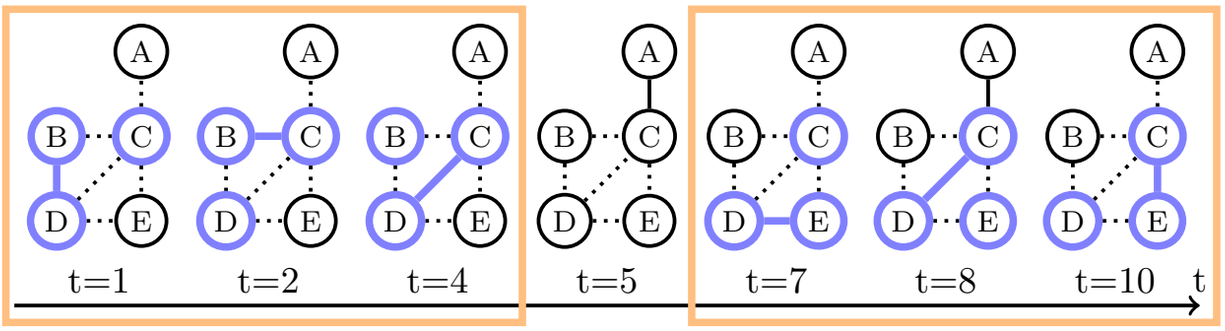



**Fig. 1** An example temporal network with five nodes and seven time stamps. The solid lines depict interactions that occur at a given time stamp, while the dotted lines depict interactions that occur at different time stamps. The highlighted time intervals, nodes, and interactions depict events discovered in the network

is to discover time intervals that provide the densest subgraphs. One interesting interval is $I = [7, 10]$, in that four different interactions $\{(A, C), (C, D), (C, E), (D, E)\}$ occur during $I$, with three of them constructing a prominently dense subgraph—a clique $\{C, D, E\}$. Thus, a pair (interval, a subgraph covered by the interval) ($[7, 10]$, $\{(C, D), (C, E), (D, E)\}$) summarizes an interesting episode in the history of interactions of this toy network. Another interesting interval is $[1, 4]$ as it contains a clique $\{B, C, D\}$. Thus, our network partition would be $(([1, 4], \{(B, C), (B, D), (D, E)\}), ([7, 10], \{(C, D), (C, E), (D, E)\}))$. Note that interaction $(A, C)$ is completely ignored as it does not contribute to any dense subgraph.

A naïve solution to this problem has polynomial but prohibitively high running time. Thus, we adapt existing recent work on dynamic densest subgraph discovery [19] and approximate dynamic programming [47] to design a fast approximation algorithm (Sect. 3).

Next (Sect. 4), we shift our attention to encouraging coverage of a larger set of nodes, so as to produce richer and more interesting structures. The resulting new problem formulation turns out to be **NP**-hard. However, on static graphs a simple greedy algorithm leads to approximate solution due to the submodularity of the objective function. Following this observation, we extend this greedy approach for the case of temporal networks. Despite the fact that the approximation guarantee does not carry on when generalizing to the temporal case, our experimental evaluation indicates that the method produces solutions of very high quality.

Experiments on synthetic and real-world datasets (Sect. 5) and a case study on Twitter data (Sect. 6) confirm that our methods are efficient and produce meaningful and high-quality results.

## 2 Problem formulation

We are given a *temporal graph* $G = (V, \mathcal{T}, E)$, where $V$ denotes the set of nodes, $\mathcal{T} = [0, 1, \ldots, t_{\max}] \subset \mathbb{N}$ is a discrete time domain, and $E \subseteq V \times V \times \mathcal{T}$ is the set of all temporal edges. Given a temporal interval $T = [t_1, t_2]$ with $t_1, t_2 \in \mathcal{T}$, let $G[T] = (V[T], E[T])$ be the subgraph induced by the set of temporal edges $E[T] = \{(u, v) \mid (u, v, t) \in E, t \in T\}$ with $V[T]$ being the set of endpoints of edges $E[T]$.

**Definition 1** (*Episode*) Given a temporal graph $G = (V, \mathcal{T}, E)$, we define an episode as a pair $(I, H)$ where $I = [t_1, t_2]$ is a temporal interval with $t_1, t_2 \in \mathcal{T}$ and $H$ is a subgraph of $G[I]$.

Our goal is to find a set of interesting episodes along the lifetime of the temporal graph. In particular, our measure of interestingness is the density of the subgraph in the episodes. We adopt the widely used notion of density of a subgraph $H = (V(H), E(H))$ as the average degree of the nodes in the subgraph, i.e., $d(H) = \frac{|E(H)|}{|V(H)|}$. While several definitions for density have been studied in the literature, the one we focus on enjoys the following nice properties: It can be optimized exactly [27] and approximated efficiently [15], while a densest subgraph can be computed in real-world graphs containing up to tens of billions of edges [17].

**Problem 1** ($k$-Densest-Episodes) Given a temporal graph $G = (V, \mathcal{T}, E)$ and an integer $k \in \mathbb{N}$, find a set of $k$ episodes $S = \{(I_\ell, H_\ell)\}$, for $\ell = 1, \ldots, k$ such that $\{I_\ell\}$ are disjoint intervals and the profit $\sum_{\ell=1}^{k} d(H_\ell)$ is maximized.

We can solve Problem 1 in polynomial time. To see this, let $S^*$ be an optimum solution for Problem 1 and let $\mathcal{I}(S^*) = \{I_1, \ldots, I_k\}$ and $\mathcal{G}(S^*) = \{H_1, \ldots, H_k\}$. Observe that without loss of generality, we can assume that the union of the intervals in $\mathcal{I}(S^*)$ is equal to the set of time stamps $\mathcal{T}$, that is, $\mathcal{I}(S^*)$ is a $k$-segmentation of $\mathcal{T}$. This follows from the fact that by increasing the length of the $I_\ell$'s, the density of the corresponding densest subgraphs cannot decrease.

Given an interval $I_\ell$, a densest subgraph in $G(I_\ell)$ can be found by running any algorithm for computing a densest subgraph: in $\mathcal{O}(nm \log n)$ time by the easy-to-implement algorithm of Goldberg et al. [27,43] or in $\mathcal{O}(nm \log(n^2/m))$ time by the more involved algorithm by Gallo et al. [25], where $n$ and $m$ denote the number of nodes and edges in $G(I_\ell)$, respectively. An optimal segmentation can be solved by a standard dynamic-programming approach, requiring $\mathcal{O}(k|\mathcal{T}|^2)$ steps [9]. By combining the subroutine for computing an optimal segmentation with either subroutine for computing a densest subgraph for each given interval, one can find a solution to Problem 1 in $\mathcal{O}(k|\mathcal{T}|^2 nm \log n)$, or $\mathcal{O}(k|\mathcal{T}|^2 nm \log(n^2/m))$, respectively.

As a post-processing step, we can trim the intervals in an optimal solution $S^* = \{(I_\ell, H_\ell)\}$ by calculating the minimum subinterval of $I_\ell$, which spans all edges of $H_\ell$, for each $\ell = 1, \ldots, k$.

## 3 Approximate dynamic programming

The simple algorithm discussed in the previous section has a running time, which is prohibitively expensive for large graphs. In this section, we develop a fast algorithm with approximation guarantees.

The derivations below closely follow the ones in [47], which improves [29]. However, we cannot use those results directly: Both papers work with minimization problems and use the fact that the profit of an interval is not less than the profit of its subintervals. In contrast, our problem is a maximization problem and requires a tailored solution.

Given a time interval $T = [t_1, t_2]$, we write $d^*(T)$ to denote the density of the densest subgraph in $T$, that is, $d^*(T) = \max_{H \subseteq G(T)} d(H)$. For simplicity, we define $d^*([t_1, t_2]) = 0$ if $t_2 < t_1$. Problem 1 is now a classic $k$-segmentation problem of $\mathcal{T}$ maximizing the total sum of scores $d^*(T)$ for individual time intervals. For notation simplicity, we assume that all time stamps $\mathcal{T}$ are enumerated by integers from 1 to $r$.

Let $o[i, \ell]$ be the profit of the optimal $\ell$-segmentation using only the first $i$ time stamps. Then,

$$o[i, \ell] = \max_{j<i} o[j, \ell - 1] + d^*(j + 1, i), \tag{1}$$

**Algorithm 1:** `ApproxDP`$(k, \epsilon)$ computes $k$-segmentation with $\epsilon$-approximation guarantee

**Input**: number of intervals $k$, parameter $\epsilon$
**Output**: approximate solution $s[i, \ell]$ for $i \in [1, r]$, $\ell \in [1, k]$

```
1 for i = 1, ..., r do s[i, 1] = d*([1, i]) for ℓ = 2, ..., k do
2 |   A = [ ];
3 |   for i = 1, ..., r do
4 |   |   add i to A;
5 |   |   s[i, ℓ] = max{s[i − 1, ℓ], s[i, ℓ − 1], max_{a∈A}(s[a − 1, ℓ − 1] + d*([a, i]))};
6 |   |   A = SPRS(A, s[i, ℓ], ℓ, ϵ)
7 |   end
8 end
9 return s
```

**Algorithm 2:** `SPRS`$(A, \sigma, \ell, \epsilon)$, a subroutine keeping the candidate list short.

**Input**: current enumerated candidates $A = (a_1, a_2, \ldots, a_{|A|})$, sparsification factor $\sigma = s[i, \ell]$, current number of intervals $\ell$, approximation parameter $\epsilon$
**Output**: sparsified $A$

```
1 δ = σ ϵ/(k+ℓϵ);
2 j = 1;
3 while j < |A| − 1 do
4 |   if s[a_{j+2}, ℓ − 1] − s[a_j, ℓ − 1] ≤ δ then remove a_{j+1} from A else j = j + 1
5 end
6 return A
```

and $o[i, k]$ can be computed recursively.

Our goal is to approximate $o[i, \ell]$ quickly with a score which we will denote by $s[i, \ell]$. The main idea behind the speedup is not to test all possible values of $j$ in Eq. 1. Instead, we are going to keep a small set of candidates, denoted by $A$, and only use those values for testing.

The challenge is how to keep $A$ small enough while at the same time guarantee the approximation ratio. The pseudo-code achieving this balance is given in Algorithm 1, while a subroutine that keeps the candidate list short is given in Algorithm 2. Algorithm 1 executes a standard dynamic programming search: It assumes that partition of $i' < i$ first data points into $\ell - 1$ intervals is already calculated and finds the best last interval $[a, i]$ for partitioning of $i$ first points into $l$ intervals. However, it does not consider all possible candidates $[a, i]$, but only a sparsified list, which guarantees to preserve a quality guarantee. The sparsified list is built for a fixed number of intervals $\ell$ starting from empty list. Intuitively, it keeps only candidates $A = \{a_j\}$ with significant difference in $s[a_j, \ell - 1]$. The significance of the difference depends on the current best profit $s[i, \ell]$: The larger the value of the solution found, the less cautious we can be about lost candidates and the coarser becomes $A$. Thus, we need to refine $A$ by Algorithm 2 after each processed $i$.

We first study the approximation guarantee of `ApproxDP`, assuming that $d^*(\cdot)$ is calculated exactly.

**Proposition 1** *Let $s[i, \ell]$ be the profit table constructed by `ApproxDP`$(k, \epsilon)$. Then, $s[i, \ell](\frac{\ell\epsilon}{k} + 1) \geq o(i, \ell)$.*

To prove the proposition, let us first fix $\ell$ and let $A_i$ be the set of candidates in $A$ to be tested on line 6 of round $i$. Let $\delta_i$ be the value of $\delta$ in Algorithm 2, called on iteration $i$. Then, $\delta_{i-1}$ is the coarsening parameter used to sparsify $A_i$.

**Lemma 1** *For every $b \in [1, i]$, there is $a_j \in A_i$, such that*

$$s[a_j - 1, \ell - 1] + d^*([a_j, i]) \geq s[b - 1, \ell - 1] + d^*([b, i]) - \delta_{i-1}.$$

***Proof*** We say that a list of numbers $A = (a_j)$ is $i$-dense, if

$$s[a_{j+1} - 1, \ell - 1] - s[a_j - 1, \ell - 1] \leq \delta_{i-1} \text{ or } a_{j+1} = a_j + 1,$$

for every $a_j \in A$ with $j < |A|$. We first prove by induction over $i$ that $A_i$ is $i$-dense.

Assume that $A_{i-1}$ is $(i-1)$-dense. The `SPRS` procedure never deletes the last element, so $(i-1) \in A_{i-1}$, and $A_{i-1} \cup \{i\}$ is $(i-1)$-dense. Note that $\delta_{i-2} \leq \delta_{i-1}$, because $s[i, \ell]$ is monotone, and $s[i, \ell] \geq s[i-1, \ell]$, due to explicit check in line 6 of procedure `ApproxDP`. Thus, $A_{i-1} \cup \{i\}$ is $i$-dense. Since $A_i = \texttt{SPRS}(A_{i-1}) \cup \{i\}$, and `SPRS` does not create gaps larger than $\delta_{i-1}$, the list $A_i$ is $i$-dense.

Let $a_j$ be the largest element in $A_i$, such that $a_j \leq b$. Then, either $a_j \leq b < a_{j+1}$ or $b = a_{|A_i|}$ and $a_j = a_{|A_i|}$. In the first case, due to monotonicity, we have $s[a_{j+1}, \ell - 1] \geq s[b, \ell - 1]$, which gives $s[b-1, \ell-1] - s[a_j - 1, \ell - 1] \leq \delta_{i-1}$. The second case is trivial.

Due to monotonicity, $d^*([a_j, i]) \geq d^*([b, i])$. This concludes the proof. □

We can now complete the proof of Proposition 1.

***Proof of Proposition 1*** We will prove the result with induction over $\ell$. The claim holds for $\ell = 1$ and any $i$ as we initialize $s[i, 1]$ by optimal values (on line 1 of Algorithm 1). We assume that the approximation guarantee holds for $\ell - 1$, that is,

$$s[i, \ell - 1](1 + \frac{\epsilon}{k}(\ell - 1)) \geq o[i, \ell - 1]$$

and we prove the result for $\ell$.

Let $\alpha = (1 + \frac{\epsilon}{k}(\ell - 1))$. Let $b$ be the starting point of the last interval of optimal solution $o[i, \ell]$, and let $a_j$ be as given by Lemma 1. We upper bound

$$\delta_{i-1} = s[i-1, \ell-1]\frac{\epsilon}{k + \epsilon\ell} \leq s[i, \ell]\frac{\epsilon}{k + \epsilon\ell} \leq s[i, \ell]\frac{\epsilon}{\alpha k}. \tag{2}$$

Then,

$$\begin{aligned}
\alpha s[i, \ell] &\geq \alpha(s[a_j - 1, \ell - 1] + d^*([a_j, i])) && (a_j \in A_i)\\
&\geq \alpha(s[b - 1, \ell - 1] + d^*([b, i]) - \delta_{i-1}) && (\text{Lemma 1})\\
&= \alpha s[b - 1, \ell - 1] + \alpha d^*([b, i]) - \alpha\delta_{i-1}\\
&\geq o[b - 1, \ell - 1] + \alpha d^*([b, i]) - \alpha\delta_{i-1} && (\text{induction})\\
&\geq o[b - 1, \ell - 1] + d^*([b, i]) - \alpha\delta_{i-1} && (\alpha \geq 1)\\
&\geq o[b - 1, \ell - 1] + d^*([b, i]) - s[i, \ell]\frac{\epsilon}{k} && (Eq.\ 2)\\
&= o[i, \ell] - s[i, \ell]\frac{\epsilon}{k}.
\end{aligned}$$

As a result, $s[i, \ell](1 + \frac{\epsilon}{k}\ell) \geq o[i, \ell])$. □

Let us now address the running time of the approximate dynamic programming.

**Proposition 2** *The running time of `ApproxDP` is $\mathcal{O}(\frac{k^2}{\epsilon}r)$.*

***Proof*** Let us fix $i$ and $\ell$, and count the number of candidates in $A_i$. Note that $|A_i| = |\texttt{SPRS}(A_{i-1})| + 1$. The list of candidates $\texttt{SPRS}(A_{i-1})$ corresponds to a monotonically increasing sequence of $s[a, \ell]$, with consecutive elements being at least $\delta_{i-1}$ apart. Thus, $|\texttt{SPRS}(A_{i-1})| \leq \frac{s[i-1,\ell]}{\delta_{i-1}} = \frac{k+\ell\epsilon}{\epsilon} \leq \frac{k(1+\epsilon)}{\epsilon}$ and the number of operations in one call of the inner loop (lines 4–8) of Algorithm 1 is $\mathcal{O}(k/\epsilon)$. Since this loop is called $kr$ times, the result follows. □

Since computing $d^*$ requires time $\mathcal{O}(nm \log n)$, the total running time is $\mathcal{O}(r\frac{k^2}{\epsilon} nm \log n)$, where $r = |\mathcal{T}|$. We further speed up our algorithm by approximating the value $d^*$ by means of one of the approaches developed by [19]. In particular, we employ the algorithm that maintains a $2(1 + \epsilon)$-approximate solution for the incremental densest subgraph problem (i.e., edge insertions only), while having poly-logarithmic amortized cost. We shall refer to such an algorithm as `ApprDens`.

`ApprDens` allows us to efficiently maintain the approximate density of the densest subgraph $d^*([a, i])$ for each $a$ in $A_i$ in `ApproxDP`, as larger values of $i$ are processed and edges are added. Whenever we remove an item $a$ from $A_i$ in `SPRS`, we also drop the corresponding instance of `ApprDens`.

From the fact that an approximate densest subgraph can be maintained with poly-logarithmic amortized cost, it follows that our algorithm has quasi-linear running time.

**Proposition 3** *`ApproxDP` combined with `ApprDens` runs in $\mathcal{O}(\frac{k^2}{\epsilon_1\epsilon_2^2}|\mathcal{T}|m_t \log^2 n)$ time, where $\epsilon_1$ and $\epsilon_2$ are the respective approximation parameters for `ApproxDP` and `ApprDens` and $m_t$ is the maximum number of edges per time stamp.*

For real-world highly dynamic temporal networks, we can safely assume that $m_t$ is a small constant.

***Proof*** To fill in cell $s[i, l]$, we need to update $|A_i| = \mathcal{O}(k/\epsilon)$ graphs by adding at most (some edges can be already in the graphs) $m_i$ edges—the number of edges with $t_i$ time stamp. Let $m_t$ be the maximum number of edges per time stamp. Theorem 4 in [19] states that maintaining the graph with $m_i$ edges requires $\mathcal{O}(m_i\epsilon_2^{-2} \log^2 n)$ time. We still need to fill $k|\mathcal{T}|$ cells in the DP matrix. Combining these two results proves the proposition. □

When combining `ApproxDP` with `ApprDens`, we wish to maintain the same approximation guarantee of `ApprDens`. Recall that `ApproxDP` leverages the fact that the profit function is monotone and non-increasing. Unfortunately, `ApprDens` does not necessarily yield a monotone score function, as the density of the computed subgraph might decrease when a new edge is inserted. This can be easily circumvented by keeping track of the best solution, i.e., the subgraph with highest density. The following proposition holds.

**Proposition 4** *`ApproxDP` combined with `ApprDens` yields a $2(1 + \epsilon_1)(1 + \epsilon_2)$-approximation guarantee.*

***Proof*** Let $d_a^*(T)$ be the density of the graph returned by `ApprDens` for a time interval $T$. Let $O$ be the optimal $k$-segmentation, let $\mathcal{I}(O)$ be the intervals of this solution, and let $q_1 = \sum_{I\in\mathcal{I}(O)} d^*(I)$ be its score. Let also $q_2 = \sum_{I\in\mathcal{I}(O)} d_a^*(I)$. Let $q_3$ be the score of the optimal $k$-segmentation $O_a$ using $d_a^*$. Note that the intervals constituting the solution $O_a$ may not be the same as in $O$, as they are optimal solutions for different interval scoring functions $d^*$ and $d_a^*$. Thus, $q_3$ may not be equal to $q_2$. Let $q_4$ be the score of the segmentation produced by `ApproxDP`. Then,

$$q_1 \leq 2(1 + \epsilon_2)q_2 \leq 2(1 + \epsilon_2)q_3 \leq 2(1 + \epsilon_2)(1 + \epsilon_1)q_4,$$

completing the proof. □

We will refer to this combination of `ApproxDP` with `ApprDens` as Algorithm KGAPPROX.

## 4 Encouraging larger and more diverse subgraphs

Problem 1 is focused on total density maximization; thus, its solution can contain graphs which are dense, but union of their node sets can cover only a small part of the network. Such segmentation is useful when we are interested in the densest temporally coherent subgraphs, which can be understood as tight cores of temporal clusters. However, segmentations with larger but less dense subgraphs, covering a larger fraction of nodes, can be useful to get a high-level explanation of the whole temporal network. To allow for such segmentations, we extend Problem 1 to take into account node coverage.

Denote the set of subgraphs $G_i$, which are included in solution episodes $S = \{(I_i, G_i)\}$ as $\mathcal{G} = \{G_i\}$ for $i = 1, \ldots, k$ . Given a collection of subgraphs $\mathcal{G}$, let $x_v(\mathcal{G}) = |\{G_i \in \mathcal{G} : v \in V(G_i), G_i \in \mathcal{G}\}|$ be the number of subgraphs in $\mathcal{G}$, which include node $v$. We consider generalized cover functions of the type

$$\text{cover}(\mathcal{G} \mid w) = \sum_{v \in V} w(x_v(\mathcal{G})),$$

where $w$ is a nonnegative non-decreasing concave function of $x_v(\mathcal{G})$. If $w(x_v(\mathcal{G}))$ is a 0–1 indicator function, then the function $\text{cover}(\mathcal{G} \mid w)$ is a standard cover, which is intuitive and easy to optimize by a greedy algorithm. Another instance of the generalized cover function, inspired by text summarization research [35], is $w(x_v(\mathcal{G})) = \sqrt{x_v(\mathcal{G})}$. It ensures that the marginal gain of a node decreases proportionally to the number of times the node is covered. We add the cover term to the cost function of Problem 1, and we obtain the resulting problem formulation.

**Problem 2** ($k$-Densest-Episodes-EC) Given a temporal graph $G = (V, \mathcal{T}, E)$, integer $k$, parameter $\lambda \geq 0$, find a $k$-segmentation $S = \{(I_i, G_i)\}$ of $G$, such that $\text{profit}(S) = \sum_{G_i \in \mathcal{G}} d(G_i) + \lambda \, \text{cover}(\mathcal{G} \mid w)$ is maximized.

Unlike Problem 1, this problem cannot be solved in polynomial time.

**Proposition 5** *Problem 2 is **NP**-hard.*

***Proof*** We will prove the hardness by reducing the set packing problem to $k$- DENSEST-EPISODES- EC. In the set packing problem, we are given a collection $\mathcal{C} = \{C_1, \ldots, C_\ell\}$ of sets and are asked whether there are $p$ disjoint sets. We can safely assume that $|C_i| = 3$.

Assume that we are given such a collection, and let us construct the temporal graph. The nodes $V$ consist of two sets $V_1$ and $V_2$. The first set $V_1$ corresponds to the elements in $\bigcup_i C_i$. The second set $V_2$ consists of $q = 6\ell + 3$ nodes. There are $2\ell$ time stamps. At the $2i$th time stamp, we connect the nodes corresponding to $C_i$, while at odd time stamps, we full-connect $V_2$. Finally, we set $k = \ell + p$ and $\lambda = 1/(|V| + 1)$. We use 0–1 indicator function for $w$.

We claim that there is a solution to the set packing problem if and only if there is a solution to $k$- DENSEST- EPISODES- EC with the profit of at least $\ell(q-1)/2 + p + \lambda(3p + q)$.

To prove the only if direction, assume there is a collection $\mathcal{C}'$ of $p$ disjoint sets. Build a $k$-segmentation by selecting each clique spanning $V_2$ to be in its own segment, as well as the three cliques corresponding to the sets in $\mathcal{C}'$. This solution will have the necessary profit.

Let us now prove the if direction. Assume an optimal $k$-segmentation $S$. It is easy to see that if the $i$th segment contains an odd time stamp, then $G_i$ must be the clique spanning $V_2$. On the other hand, if the $i$th segment is equal to $[2j, 2j]$, then $G_j$ is a clique connecting $C_j$.

Let $a$ be the number of segments containing odd time stamps, we can safely assume that $a > 0$. Let $b$ be the number of segments containing only even time stamps. Let $c$ be the total number of nodes in $V_1$ covered by at least one segment. Then,

$$\text{profit}(S) = a(q-1)/2 + b + \lambda(c+q).$$

We assume that $\text{profit}(S) \geq \ell(q-1)/2 + p + \lambda(3p+q)$. Since $b + \lambda(c+q) \leq \ell + 1 < (q-1)/2$ and $\lambda(c+q) < 1$, this is only possible if $a = \ell$, $b = p$, and $c = 3p$. This completes the proof. □

### 4.1 *k* static overlapping densest subgraphs

Given the complexity of Problem 2, we start with analysis of a static graph case. We formulate the *k-overlapping-densest-subgraphs problem* and design a linear algorithm with an approximation guarantee. We will later apply the developed approach to temporal graphs; however, the algorithm can be used as an efficient stand-alone method for finding overlapping dense subgraphs.

**Problem 3** (*k static overlapping densest subgraphs*) Given a static graph $H = (V, E')$, integer $k$, and real $\lambda \geq 0$, find a set of $k$ subgraphs $\mathcal{H} = \{H_i \subseteq H\}$, such that $\text{profit}_{ST}(\mathcal{H}) = \sum_{H_i \subseteq \mathcal{H}} d(H_i) + \lambda \cdot \text{cover}(\mathcal{H} \mid w)$ is maximized.

Next, we show below how to obtain a constant-factor approximate solution. We start with showing that the generalized cover function has beneficial combinatorial properties: It is submodular, nonnegative, and non-decreasing with respect to the set of subgraphs. The density term of the cost function of Problem 2 (and Problem 3) is a linear function of subgraphs, and thus the whole cost function is nonnegative, non-decreasing, and submodular.

**Proposition 6** *Function* $\text{cover}(\mathcal{G} \mid w)$ *is a nonnegative, non-decreasing, and submodular function of subgraphs.*

***Proof*** For a fixed $v \in V$ function $x_v(\mathcal{G})$ is non-decreasing modular (and submodular): for any set of subgraphs $X$ and a new subgraph $x$ holds that $x_v(X \cup \{x\}) - x_v(X) = 1$ if $v$ belongs to $x$ and does not belong to any subgraph in $X$, otherwise 0. By the property of submodular functions, composition of concave non-decreasing and submodular non-decreasing is non-decreasing submodular. Function $\text{cover}(\mathcal{G} \mid w)$ is submodular non-decreasing as a nonnegative linear combination. Nonnegativity follows from nonnegativity of $w$. □

To solve Problem 3, we can search greedily over subgraphs. Let $\mathcal{H}_{i-1} = \{H_1, \ldots, H_{i-1}\}$, and define marginal node gain, given weight function $w$, as

$$\delta(v \mid \mathcal{H}_{i-1}, w) = w(x_v(\mathcal{H}_{i-1} \cup \{v\})) - w(x_v(\mathcal{H}_{i-1})). \tag{3}$$

Here, $\{v\}$ refers to a graph containing only $v$. Then, denote the marginal gain of subgraph $H_i$ given already selected graphs $\mathcal{H}_{i-1}$ as

$$\chi(H_i \mid \mathcal{H}_{i-1}, w) = d(H_i) + \lambda \sum_{v \in H_i} \delta(v \mid \mathcal{H}_{i-1}, w). \tag{4}$$

Greedy algorithm for Problem 3 consequently builds the set $\mathcal{H}$ by adding $H_i$, which maximizes gain $\chi(H_i \mid \mathcal{H}_{i-1})$. If we can find $H_i$ optimally, such algorithm yields $1 - 1/e$ approximation due to submodular maximization with cardinality constrains (see [42] for this classic result).

To find the optimal $H_i$, we need to solve the following problem.

**Problem 4** Given a static graph $H = (V, E')$, a set of subgraphs $\mathcal{H}_{i-1} = \{H_1, \ldots, H_{i-1}\}$, find a graph $F \subseteq H$, such that $\chi(F \mid \mathcal{H}_{i-1})$ is maximized.

Luckily, Problem 4 can be transformed into a (weighted) densest subgraph problem. In order to do so, we will define a weighted fully connected graph $R = (V, V \times V, a)$ having the same nodes $V$ as $H$ with the weights $a(u, v)$ defined as

$$a(u, v) = I[(u, v) \in E'] + \frac{\lambda}{1 + I[u = v]}(\delta(u \mid \mathcal{H}_{i-1}, w) + \delta(v \mid \mathcal{H}_{i-1}, w)).$$

Here, $I[\cdot]$ is an indicator function, returning 1 if the condition is true, and 0 otherwise. Note that we allow self-loop edges. Let $R'$ be a subgraph in $R$ and let $F$ be the induced subgraph in $H$ having the same nodes as $R'$. Then, it is now straightforward to see that

$$\chi(F \mid \mathcal{H}_{i-1}, w) = d(R').$$

In other words, solving Problem 4 is equivalent to solving densest subgraph problem in $R$. Consequently, we can solve Problem 4 exactly in $\mathcal{O}(|V|^3)$ time [25]. Alternatively, we can estimate it efficiently with 1/2-approximation in $O(|V|^2)$ time by Charikar et al. [15]. We will use the latter algorithm and refer to it as `StaticGreedy`.

Now we have everything to design and analyze an approximation algorithm for Problem 3. Algorithm 3 greedily finds $k$ subgraphs to solve Problem 3.

Each subgraph is sought with 1/2-approximation guarantee, and due to submodularity, greedy optimal subgraph search would be a $(1-1/e)$-approximation. Combining these results leads to the following statement.

**Proposition 7** *Algorithm 3 is a* $1/2(1 - 1/e) \approx 0.31606$ *approximation for Problem 3.*

***Proof*** Let $y$ be the value of $profit_{ST}$ score of $k$ greedily sought subgraph, assuming that each subgraph was sought optimally. The $i$th subgraph has a marginal gain $y_i$, thus $y = \sum_i^k y_i$. Let optimal solution of Problem 3 be $y^*$. Due to greedy submodular optimization $y \geq (1-1/e)y^*$, Algorithm 3 uses 1/2-approximation algorithm `StaticGreedy` for subgraph search, thus $\bar{y}_i \geq y_i/2$, where $\bar{y}_i$ is the marginal gain of the $i$-th subgraph included into the solution. Let $\bar{y}$ be the value of final solution output by Algorithm 3. Putting everything together, we have $\bar{y} = \sum_i^k \bar{y}_i \geq \sum_i^k y_i/2 = y/2 \geq 1/2(1 - 1/e)y^*$. This concludes the proof. □

The running time of Algorithm 3 is defined by the running time of the greedy subroutine and is $\Theta(k|V|^2)$.

### 4.2 Greedy dynamic programming

Similarly to Problem 1, we will use dynamic programming for Problem 2. However, as the problem is hard, we have to rely on greedy choices of the subgraphs. Thus, the obtained solution does not have any quality guarantee.

Let $M[\ell, i]$ be the profit of $i$ first points into $\ell$ intervals, let $C[\ell, i]$ be the set of subgraphs $\mathcal{G}_\ell = \{G_1, \ldots, G_\ell\}$ selected on these $\ell$ intervals, $1 \leq \ell \leq k$ and $0 \leq i \leq m$.

**Algorithm 3:** `StaticKDensest`

**Input**: static graph $H = (V, E')$, integer $k$, parameter $\lambda \geq 0$
**Output**: a set of $k$ subgraphs $\mathcal{H} = \{H_j \subseteq H\}$

$\mathcal{H} = \emptyset$;
**for** $j = 1, \ldots, k$ **do**
  $H_j = F$ /* where $F$ is a solution of Problem 4 for $H = (V, E')$ and $\mathcal{H}_{i-1} = \mathcal{H}$ */
  $\mathcal{H} = \mathcal{H} \cup \{H_j\}$;
**end**
**return** $\mathcal{H}$

Define marginal gain interval $[j, i]$, given that $j - 1$ are already segmented into $\ell - 1$ interval, (here $\chi$ is defined in Eq. 4):

$$\text{gain}([j, i], C[\ell - 1, j - 1]) = \max_{G' \subseteq G([j,i])} \chi(G' \mid C[\ell - 1, j - 1]). \tag{5}$$

This leads to a dynamic program

$$M[\ell, i] = \max_{1 \leq j \leq i+1} M[\ell - 1, j - 1] + \text{gain}([j, i], C[\ell - 1, j - 1]) \text{ for } 1 < \ell \leq k,$$
$$M[1, i] = d^*([0, i]) \text{ for } 0 \leq i \leq m,$$
$$M[k', 0] = 0 \text{ for } 1 \leq k' \leq k.$$

After filling this table, $M[k, m]$ contains the profit of $k$-segmentation with subgraph overlaps. $C[k, m]$ will contain selected subgraphs, and the intervals and subgraphs can be reconstructed, if we keep track of the starting points of selected last intervals. Note that profit $M[k, m]$ is not optimal, because the choice of subgraph $G_i$ depends on the interval and the previous choices.

We perform dynamic programming by approximation algorithm `ApproxDP`, and the densest subgraph for each candidate interval is retrieved by Epasto et al. [19]. We refer to the resulting algorithm as KGCVR.

To keep track on number of $x_v$ when we construct $\mathcal{G}$, we need to keep frequencies of each node. To avoid extensive memory costs, in the experiments we use Min-Count sketches.

## 5 Experimental evaluation

We evaluate the performance of the proposed algorithms on synthetic graphs and real-world social networks. The datasets are described below. Unless specified, we post-process the output of all algorithms and report the optimal densest subgraphs in the output intervals. Our datasets and implementations are publicly available.[1]

### 5.1 Synthetic data

We generate a temporal network with $k$ planted communities and a background network. All graphs are Erdős-Rényi. The communities $G'$ have the same density, disjoint set of nodes, and are planted in consecutive non-overlapping intervals. The background network $G$ includes

[1] https://github.com/polinapolina/segmentation-meets-densest-subgraph.

nodes from all planted communities $G'$. The edges of $G$ are distributed uniformly on the timeline. In a typical setup, the length of the whole time interval $T$ is $|T| = 1000$ time units, while the edges of each $G'$ are generated in intervals of length $|T'| = 100$ time units. The densities of the communities and the background network vary. The number of nodes in $G$ is set to 100.

We produced two families of synthetic temporal networks: *Synthetic1* and *Synthetic2*. In the first setting (dataset family *Synthetic1*), we vary the average degree of the background network from 0.5 to 4 and fix the density of the planted 5-cliques to 4. *Synthetic1* allows to test the robustness of our algorithms against background noise. In the second setting (dataset family *Synthetic2*), we vary the density of planted eight-node graphs from 2 to 7, while the average degree of the background network is fixed to 2. A separate synthetic dataset *Synthetic3* is designed to test the effect of setting different parameters $k$ in the algorithms. The dataset contains $k = 10$ intervals with the activity of eight-node subgraphs with average degree 5, and the background noise has average degree 2.

### 5.2 Real-world data

We use the following real-world datasets: *Facebook* [51] is a subset of Facebook activity in the New Orleans regional community. Interactions are posts of users on each other walls. The data cover the time period from 9.05.06 to 20.08.06. The *Twitter* dataset tracks activity of Twitter users in Helsinki in year 2013. As interactions, we consider tweets that contain mentions of other users. The *Students*[2] dataset logs activity in a student online network at the University of California, Irvine. Nodes represent students, and edges represent messages with ignored directions. *Enron*:[3] is a popular dataset that contains e-mail communication of senior management in a large company and spans several years.

For a case study, we create a hashtag network from Twitter dataset (the same tweets from users in Helsinki in year 2013): Nodes represent hashtags—there is an interaction, if two hashtags occur in the same tweet. The time stamp of the interaction corresponds to the time stamp of the tweet. We denote this dataset as *Twitter#*.

### 5.3 Optimal baseline

A natural baseline for kGAPPROX is OPTIMAL, which combines exact dynamic programming with finding the optimal densest subgraph for each candidate interval. Due to the high running time of OPTIMAL, we generate a very small dataset with 60 time stamps, where each time stamp contains a random graph with 3–6 nodes and random density. We vary the number of intervals $k$ and report the value of the solution (without any post-processing) and the running time in Fig. 2. On this toy dataset, kGAPPROX is able to find near-optimal solution, while being significantly faster than OPTIMAL.

### 5.4 Results on synthetic datasets

Next, we evaluate the performance of kGAPPROX on the synthetic datasets *Synthetic1* and *Synthetic2* by assessing how well the algorithm finds the planted subgraphs. We report mean

[2] http://toreopsahl.com/datasets/#online_social_network.

[3] http://www.cs.cmu.edu/~./enron/.

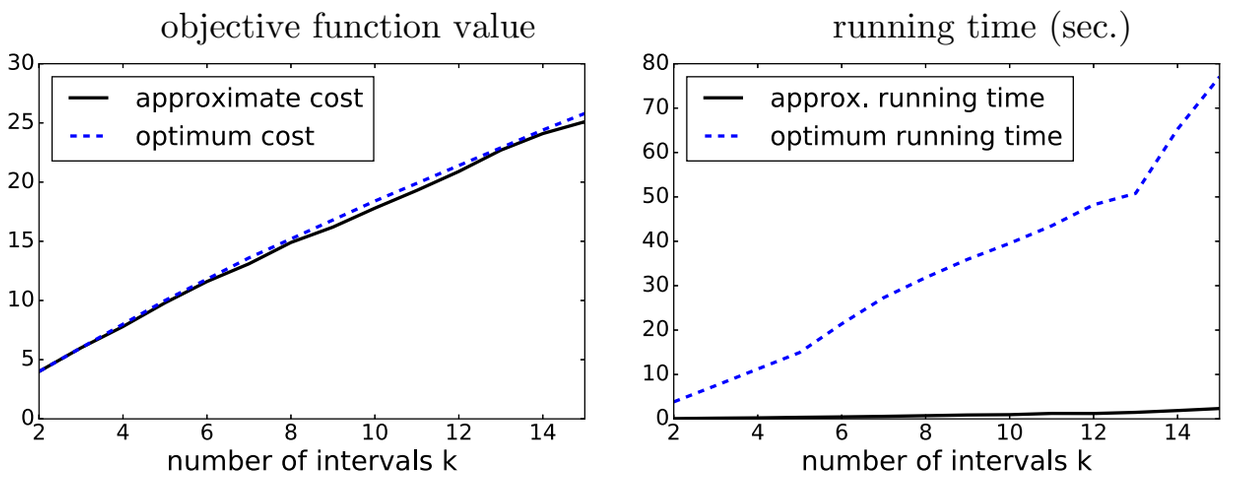



**Fig. 2** Comparison between optimum and approximate solutions (OPTIMAL and KGAPPROX). Approximate algorithm was run with $\epsilon_1 = \epsilon_2 = 0.1$. Running time is in seconds

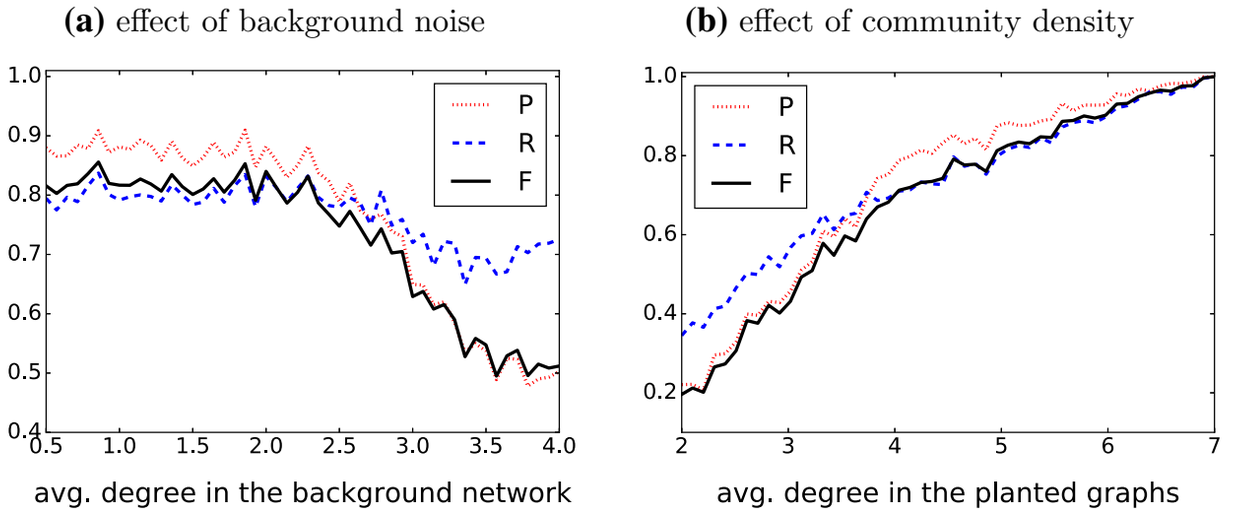



**Fig. 3** Precision, recall, and $F$-measure on synthetic datasets. For plot **a**, the community average degree is fixed to 5 (*Synthetic1* dataset), and for plot, **b** the background network degree is fixed to 2 (*Synthetic2* dataset). Plot **a** the mean standard deviation for precision is 0.193, for recall is 0.183, and for $F$-measure is 0.180. Plot **b** the mean standard deviation for precision is 0.188, for recall is 0.178, and for $F$-measure is 0.173

precision, recall, and $F$-measure, calculated with respect to the ground-truth subgraphs. All results are averaged over 100 independent runs.

First, Fig. 3a depicts the quality of the solution as a function of background noise. Recall that the *Synthetic1* dataset contains planted eight-node subgraphs with average degree 5. Precision and recall are generally high for all values of average degree in the background network. However, precision degrades as the density of the background network increases, as then it becomes cost-beneficial to add more nodes in the discovered densest subgraphs.

Second, Fig. 3b shows the quality of the solution of KGAPPROX as a function of the density in the planted subgraphs. Note that, in *Synthetic2* the density of the background network is 2. Similarly to the previous results, the quality of the solution, especially recall, degrades much only when the density of the planted and the background network becomes similar.

Figure 4 demonstrates how well the true event intervals are recovered in the case of a synthetic *Synthetic3* dataset with $k = 10$ planned events intervals. The true value of $k$ was treated as unknown, and KGAPPROX was run with all possible integer values of $k$ in [2, 20].

Figure 4a shows the quality of the intervals, precision, and recall are calculated with respect to the length of the overlap between the true interval and the output one.

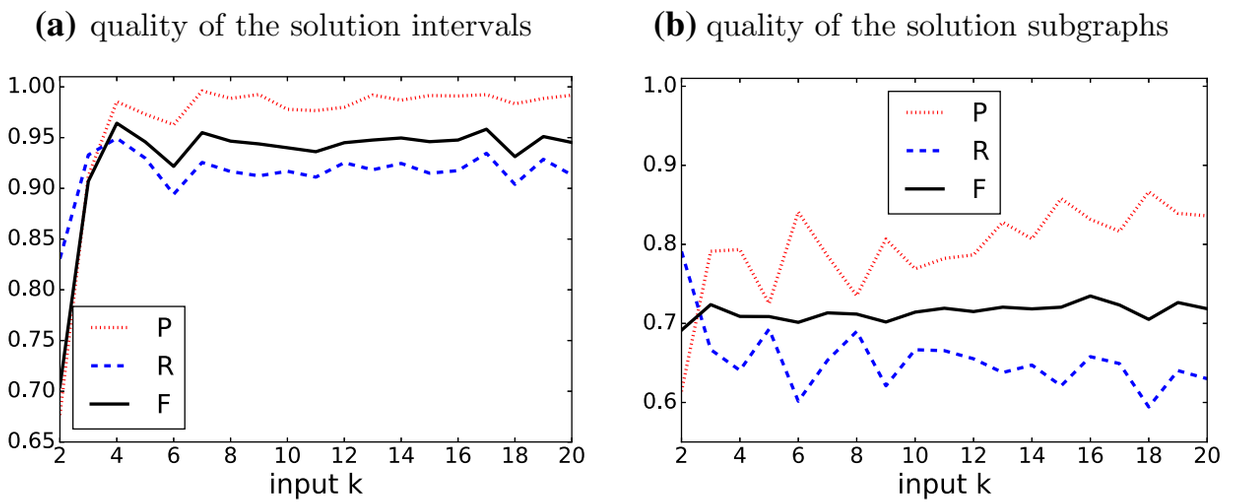



**Fig. 4** Quality of the solutions in the case of unknown $k$. Planted $k = 10$ intervals with eight-node subgraphs each (*Synthetic3* dataset). Plot **a** shows the quality of the solution segmentation, and plot **b** shows the quality of the subgraphs sought in the intervals

Since the number of intervals in the segmentation and the ground truth is different, we compare each output interval to its best match in terms of $F$-measure. That is, let $(I_1, \ldots, I_k)$ and $(I'_1, \ldots, I'_{k'})$ be the set of ground-truth intervals and solution intervals with $k$ not necessarily be equal to $k'$. For each $I'_i$ in the solution, we find the best matching interval in the ground truth $I^*_i = \max_{I_i \in (I_1, \ldots, I_k)} F(I'_i, I_i)$. Here, $F$ is $F$-measure, with precision and recall being calculated with respect to time stamps: the number of time stamps from the ground-truth interval $I_i$, which also belong to the interval $I'_i$, divided by the number of time stamps in $I'_i$ (precision) or divided by the number of time stamps in $I_i$ (recall). Once such matching interval $I^*_i$ is found for each $I'_i$, we calculate and report precision $P(I^*_i, I_i)$, recall $R(I^*_i, I_i)$, and $F$-measure $F(I^*_i, I_i)$, defined with respect to the time stamps as described above.

All the reported measures are averaged over the output intervals (and over 100 runs). After matching the intervals, we also evaluate the quality of the densest subgraphs and compare their node sets to the ground-truth events in the corresponding intervals (Fig. 4b). As we can see, the intervals are in general recovered quite well, even though the algorithm is given an incorrect value of $k$. The quality of the subgraph recovery is generally lower, which is the results of shifted borders of the intervals.

### 5.5 Results on real-world datasets

As the optimal partition algorithm OPTIMAL is not scalable for real datasets, we present comparative results of KGAPPROX with baselines KGOPTDP and KGOPTDS. The KGOPTDP algorithm performs exact dynamic programming, but uses an approximate incremental algorithm for the densest subgraph search (the incremental framework by Epasto et al. [19]). Vice versa, KGOPTDS performs approximate dynamic programming while calculating the densest subgraph optimally for each candidate interval (by Goldberg's algorithm [27]). Note that KGOPTDP has $2(1+\epsilon_{\text{DS}})^2$ approximation guarantee and KGOPTDS has $(1+\epsilon_{\text{DP}})$ approximation guarantee. However, even these non-optimal baselines are quite slow in practice and we use a subset of 1 000 interactions of *Students* and *Enron* datasets for comparative reporting.

To ensure fairness, we report the total density of the optimal densest subgraphs in the intervals returned by the algorithms.

In Table 1, we report the density of the solutions reported by KGAPPROX, KGOPTDP, and KGOPTDS, and Table 2 shows their running time. We experiment with different parameters for

**Table 1** Comparison of kGApprox with kGoptDP and kGoptDS baselines: total community density

| Dataset | Community density | | | | | | |
|---|---|---|---|---|---|---|---|
| | $\epsilon_{DP}$ | | | | | | |
| | | kGApprox | 0.01 | 0.1 | 1 | 2 | kGoptDS |
| *Students* 1000 | $\epsilon_{DS}$ | 0.01 | 4.24 | 4.24 | 4.24 | 4.24 | 6.30 |
| | | 0.1 | 4.24 | 4.24 | 4.24 | 4.24 | 6.22 |
| | | 1 | 3.82 | 3.82 | 3.82 | 3.82 | 5.76 |
| | | 2 | 3.82 | 3.82 | 3.82 | 3.82 | 5.61 |
| | | kGoptDP | 5.73 | 5.73 | 3.82 | 3.82 | |
| *Enron* 1000 | $\epsilon_{DS}$ | 0.01 | 10.4 | 10.4 | 10.0 | 10.5 | 11.3 |
| | | 0.1 | 10.3 | 10.4 | 10.0 | 10.3 | 11.0 |
| | | 1 | 9.54 | 9.54 | 8.80 | 9.83 | 11.0 |
| | | 2 | 7.34 | 7.34 | 7.34 | 7.34 | 10.8 |
| | | kGoptDP | 10.5 | 11.0 | 10.4 | 8.90 | |

the approximate densest subgraph search ($\epsilon_{DS}$) and for approximate dynamic programming ($\epsilon_{DP}$).

For both datasets, the best solution (i.e., the solution with the highest value of the profit function of Problem 1) was found by kGoptDS. This is expected as this algorithm has the best approximation factor. The solution cost decreases as $\epsilon_{DP}$ increases. On the other hand, kGoptDS has the largest running time, which decreases with increasing $\epsilon_{DP}$, but even with the largest parameter value ($\epsilon_{DP} = 2$) kGoptDS takes about an hour.

The kGoptDP algorithm typically finds the second-best solution; however it only marginally outperforms kGApprox (e.g., $\epsilon_{DS} = 0.1$), while requiring up to several orders of magnitude of higher computational time. Naturally, the quality of the solution degrades with increasing $\epsilon_{DS}$.

The solution quality degrades with increasing the approximation parameters for all algorithms. However, the degradation is not as dramatic as the worst-case bound suggests, while using such an approximation parameter offers significant speedup. kGApprox provides the fastest estimates of a good quality for a wide range of approximation parameters. Note that kGApprox is more sensitive to the changes in the quality of the densest subgraph search regulated by $\epsilon_{DS}$.

## 5.6 Running time and scalability

Figure 5 shows running time of kGApprox as a function of the approximation parameters $\epsilon_{DS}$ and $\epsilon_{DP}$. The figure confirms the theory, that is, $\epsilon_{DS}$ has significant impact on the running time, while the algorithm scales very well with $\epsilon_{DP}$.

We demonstrate scalability in Fig. 6, plotting the running time for increasing number of interactions, for *Facebook* and *Twitter* datasets. Recall that the theoretical running time is $\mathcal{O}(k^2 m \log n)$, where $n$ is the number of nodes and $m$ the number of interactions. In practice, the running time grows fast for the first thousand interactions and then saturates to linear dependence. This happens because in the beginning of the network history the number of nodes grows fast. In addition, new, denser than previously seen, subgraphs are more likely to

**Table 2** Comparison of kGapprox with kGoptDP and kGoptDS baselines: total community density: running time

| Dataset | | Running time (sec) | | | | | |
|---|---|---|---|---|---|---|---|
| | | $\epsilon_{DP}$ | | | | | |
| | | kGapprox | 0.01 | 0.1 | 1 | 2 | kGoptDS |
| *Students* 1000 | $\epsilon_{DS}$ | 0.01 | 0.62 | 0.62 | 0.63 | 0.64 | 23678 |
| | | 0.1 | 0.23 | 0.23 | 0.24 | 0.23 | 8952 |
| | | 1 | 0.13 | 0.26 | 0.13 | 0.13 | 3394 |
| | | 2 | 0.36 | 0.20 | 0.20 | 0.36 | 3769 |
| | | kGoptDP | 162 | 43.5 | 29.5 | 13.23 | |
| *Enron* 1000 | $\epsilon_{DS}$ | 0.01 | 56.4 | 55.5 | 42.3 | 31.8 | 25788 |
| | | 0.1 | 3.02 | 2.85 | 2.07 | 1.70 | 16070 |
| | | 1 | 0.43 | 0.44 | 0.29 | 0.28 | 7834 |
| | | 2 | 0.22 | 0.22 | 0.23 | 0.23 | 3469 |
| | | kGoptDP | 1654 | 61.15 | 17.82 | 6.07 | |

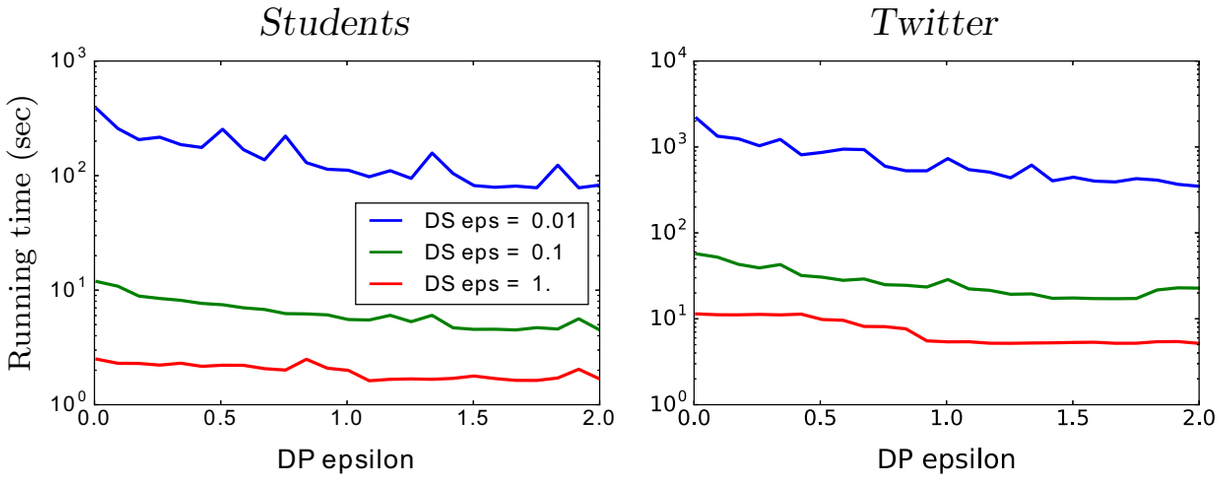



**Fig. 5** Effect of different approximation parameters in kGapprox. $k = 20$

occur. Thus, the approximate densest subgraph subroutine has to be computed more often. Furthermore, the number of intervals $k$ contributes to running time as expected.

Figure 7 shows how the cost of the solution changes as the network evolves. Setting larger $k$ results in larger total density. However, the relative change of the solution values is approximately the same for all $k$: As the number of time stamps goes from 100 to 100000, the total density increases about 2.5 times for *Facebook* dataset and 3.5 times for *Twitter* dataset. This means that while different $k$ lead to technically different segmentations, they capture the rate of network evolution.

Naturally, setting larger $k$ results in discovering subgraphs of smaller individual density, as it follows from Fig. 8. However, the relative difference between the mean density for different $k$ is typically less than the relative difference between the values of $k$ itself. This means that the algorithm tends not to split intervals of dense subgraphs to achieve a better total density, but rather discovers new dense subgraph intervals as $k$ increases.

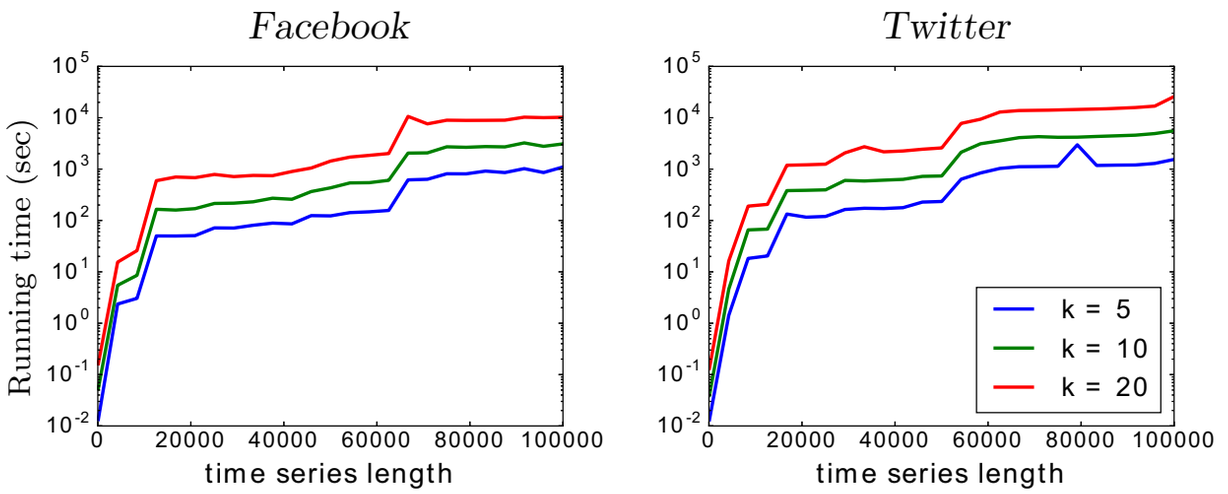



**Fig. 6** Scalability testing with $\epsilon_{\text{DS}} = \epsilon_{\text{DP}} = 0.1$

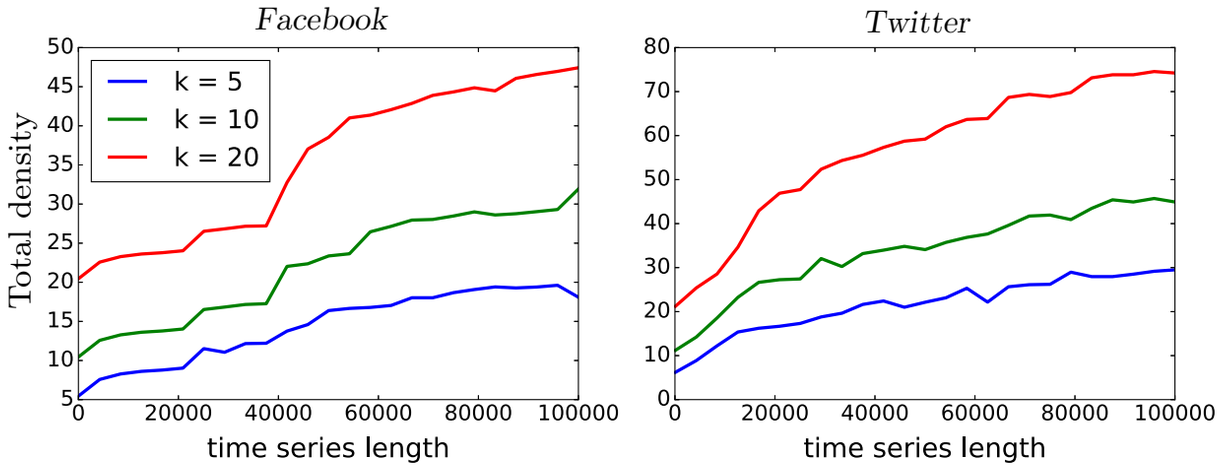



**Fig. 7** Total density of the solution subgraphs for different values of $k$ and different lengths of the time series ($\epsilon_{\text{DS}} = \epsilon_{\text{DP}} = 0.1$)

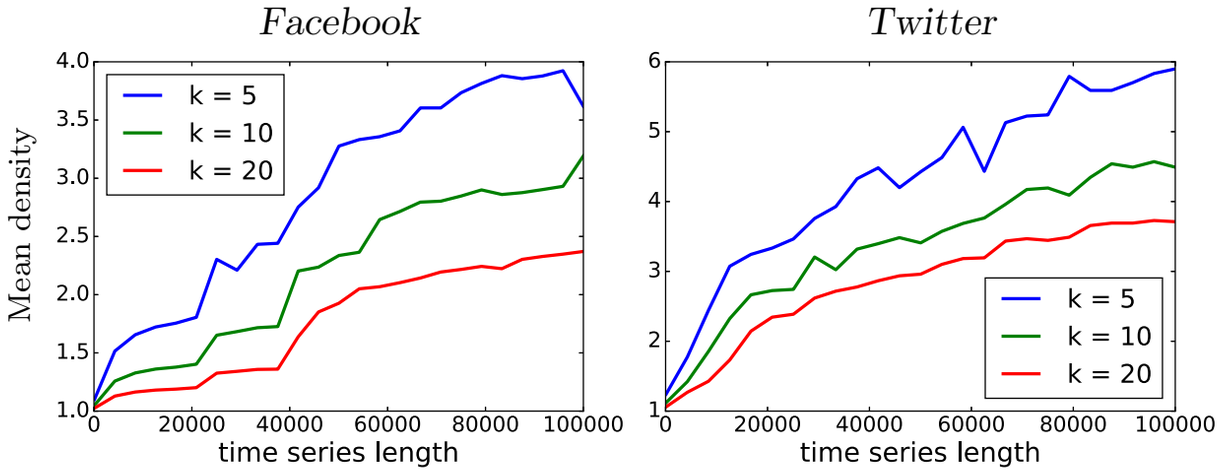



**Fig. 8** Mean density of the solution subgraphs for different values of $k$ and different lengths of the time series ($\epsilon_{\text{DS}} = \epsilon_{\text{DP}} = 0.1$)

## 5.7 Subgraphs with larger node coverage—static graphs

Next, we evaluate StaticGreedy. To measure coverage, we simply count the number of distinct nodes in the output subgraphs. We use the 10K first interactions of *Students* dataset,

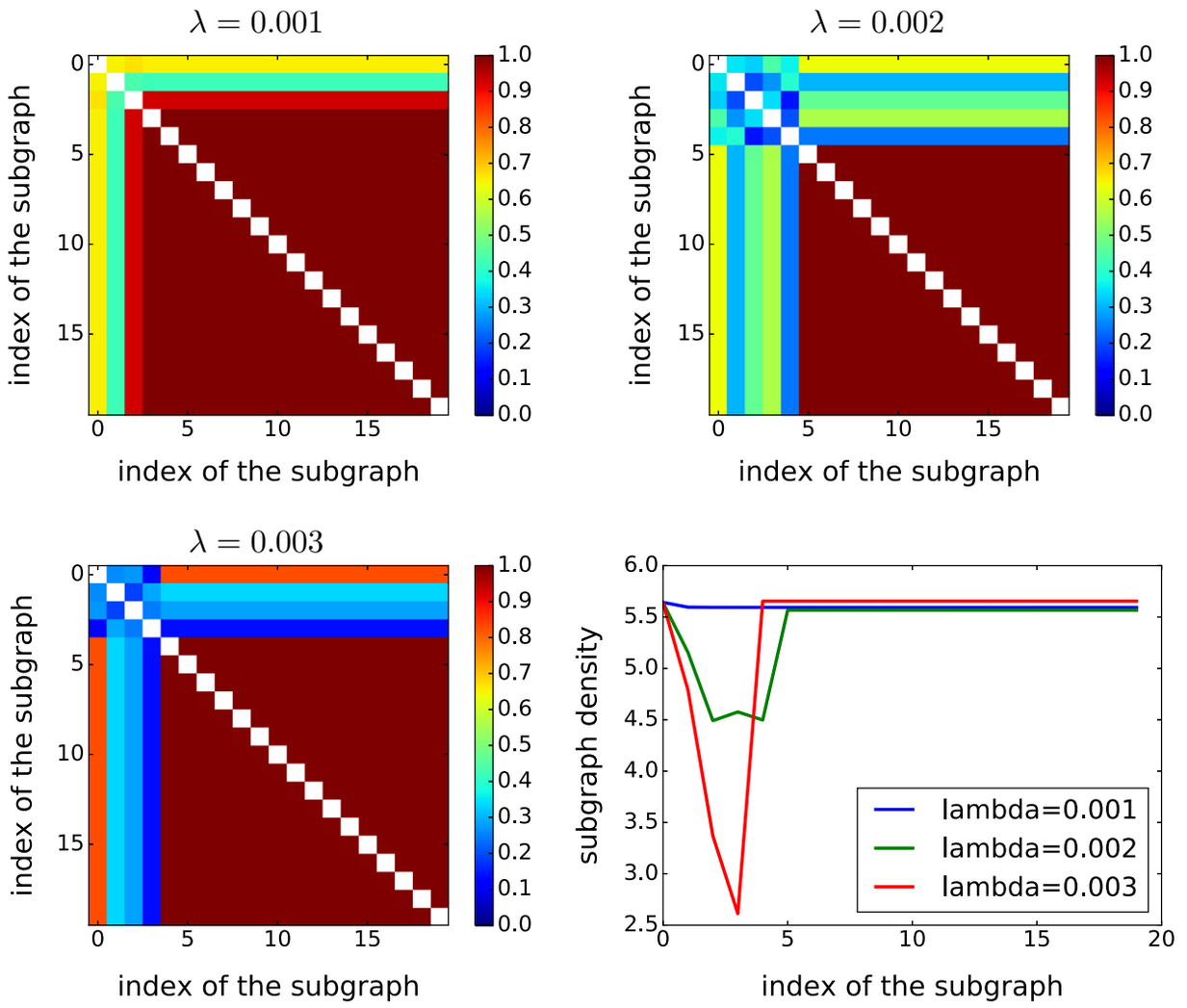



**Fig. 9** Pairwise similarities (three heatmap plots on the left) and densities (right plot) of subgraphs returned by StaticGreedy

set $k = 20$, and test different values of $\lambda$. Figure 9 shows the density and the pairwise Jaccard similarity of the node sets of the retrieved subgraphs. The subgraphs are shown in the order they are discovered. Smaller values of $\lambda$ give larger density, and larger values of $\lambda$ give more cover. We observe that, for all values of $\lambda$, in the beginning StaticGreedy returns diverse and dense subgraphs, but soon after it starts outputting graphs, which have been already selected to the solution on the previous iterations. We speculate that the algorithm finds all dense subgraphs that exist in the dataset. Regarding setting $\lambda$, we observe that $\lambda = 0.002$ offers a good trade-off in finding subgraphs of high density and moderate overlap.

### 5.8 Subgraphs with larger node coverage—dynamic graphs

Finally, we evaluate the performance of kGCvr algorithm. We vary the parameter $\lambda$ and compare different characteristics of the solution, with the solution returned by kGApprox. For different values of $\lambda$, Table 3 shows average density and total number of covered nodes, and Table 4 shows average size of the subgraphs and average pairwise Jaccard similarity. Although kGCvr does not have an approximation guarantee, for small values of $\lambda$ it finds subgraphs of the density close to kGApprox. Similarly to the static case, $\lambda$ provides an efficient trade-off between density and coverage.

**Table 3** Total density and total cover size of kGCvr's outputs with $k = 5$ and $\epsilon_{DS} = \epsilon_{DP} = 0.1$

| Dataset | λ | Density | | Cover | |
|---|---|---|---|---|---|
| | | kGCvr | kGApprox | kGCvr | kGApprox |
| *Students* | 1e−6 | 10.690 | 11.151 | 136 | 130 |
| | 1e−5 | 7.0869 | 11.151 | 813 | 130 |
| | 1e−4 | 5.0273 | 11.151 | 889 | 130 |
| *Enron* | 1e−6 | 19.995 | 19.871 | 38 | 37 |
| | 1e−5 | 19.962 | 19.871 | 40 | 37 |
| | 1e−4 | 6.5684 | 19.871 | 1144 | 37 |
| *Facebook* | 1e−8 | 5.3714 | 5.3933 | 83 | 120 |
| | 1e−7 | 4.2749 | 5.3933 | 3470 | 120 |
| | 1e−6 | 3.2673 | 5.3933 | 4100 | 120 |
| *Twitter* | 1e−7 | 9.9970 | 10.138 | 128 | 152 |
| | 1e−6 | 6.5500 | 10.138 | 3808 | 152 |
| | 1e−5 | 3.5389 | 10.138 | 4604 | 152 |

**Table 4** Average subgraph size and average Jaccard similarity between the subgraphs in the output of kGCvr with $k = 5$ and $\epsilon_{DS} = \epsilon_{DP} = 0.1$

| Dataset | λ | Size | | JSim | |
|---|---|---|---|---|---|
| | | kGCvr | kGApprox | kGCvr | kGApprox |
| *Students* | 1e−6 | 48.75 | 37.6 | 0.1449 | 0.0951 |
| | 1e−5 | 261.0 | 37.6 | 0.0788 | 0.095 |
| | 1e−4 | 286.0 | 37.6 | 0.0910 | 0.0951 |
| *Enron* | 1e−6 | 16.0 | 16.2 | 0.3619 | 0.3851 |
| | 1e−5 | 17.0 | 16.2 | 0.3660 | 0.3851 |
| | 1e−4 | 288.8 | 16.2 | 0.0808 | 0.3851 |
| *Facebook* | 1e−8 | 22.75 | 27.6 | 0.0185 | 0.0163 |
| | 1e−7 | 882.0 | 27.6 | 0.0027 | 0.0163 |
| | 1e−6 | 1228.75 | 27.6 | 0.0335 | 0.0163 |
| *Twitter* | 1e−7 | 44.25 | 54.0 | 0.1590 | 0.1673 |
| | 1e−6 | 1061.75 | 54.0 | 0.0837 | 0.1673 |
| | 1e−5 | 1379.0 | 54.0 | 0.0773 | 0.1673 |

## 5.9 Parameter selection

Both problem formulations, $k$- Densest- Episodes and $k$- Densest- Episodes- EC, follow the classic sequence segmentation problem setting [10] and take as input the number of segments ($k$) in the timeline partition. It is primarily assumed that the value of $k$ can be specified by prior knowledge and user expectation. In the case of problem formulations $k$- Densest- Episodes and $k$- Densest- Episodes- EC, we can show ("Appendix A") that the total profit is a strictly increasing function of the number of segments and reaches its maximum when $k$ is equal to the number of intervals. Thus, the value of $k$ cannot be guided by the optimal value. Furthermore, it is hard to assess the quality of the subgraphs in the segmentation: Larger

intervals with denser subgraphs correspond to larger events, while splitting an interval in favor of less dense subgraphs corresponds to sub-events. Duplicating events in the neighboring intervals can also lead to different sub-segmentation, when $k$ increases; thus, we cannot recommend to decrease $k$ if duplicates occur. However, we do not view that uncertainty with respect to the choice of $k$ as a weakness of the approach: It allows the user to explore the data at different granularity levels and possibly observe a hierarchy of events.

The problem formulations kGApprox and kGCvr require the approximation parameters $\epsilon_{\text{DP}}$ and $\epsilon_{\text{DS}}$. As we discussed in the section about the performance of kGApprox, kGoptDP, and kGoptDS (Table 1), by design our approximation algorithms are more sensitive to the changes in the quality of the densest subgraph search. The parameter $\epsilon_{\text{DS}}$ affects the calculation of profits of the intervals, and these values are used to guide the dynamic programming algorithm, while loose values of these approximation parameters are likely to misguide it. As it follows from the scalability results (Fig. 5), the algorithms scale better with the change of $\epsilon_{\text{DS}}$ rather than $\epsilon_{\text{DS}}$. However, both parameters contribute equally to the solution quality guarantee $2(1+\epsilon_{\text{DS}})(1+\epsilon_{\text{DP}})$ of Problems $k$- Densest- Episodes and the order of the approximation factor depended on the largest of $\epsilon_{\text{DS}}$ and $\epsilon_{\text{DP}}$. Thus, it is not guaranteed (and not fully supported by empirical results) that reducing only $\epsilon_{\text{DS}}$ will lead to better results faster. As a rule of thumb in most of our experiments, we use $\epsilon_{\text{DS}} = \epsilon_{\text{DP}} = 0.1$, which gives a satisfactory guarantee of 2.42 and is sufficiently fast. We use the same parameters for the experiments with kGCvr.

The last parameter to discuss is the parameter $\lambda$ in problem $k$- Densest- Episodes- EC, which controls the node coverage in the solution. The sensitivity and the range of meaningful values of this parameter depend non-trivially on the topological and temporal properties of the network. To select a good value for $\lambda$, one could try sampling different values and plot the density of the resulting subgraphs, similarly to Fig. 9. Then, one can choose a value for $\lambda$, which provides a good trade-off between diversity and density: Too small value of $\lambda$ may lead to dense but repeating structures, and too large value may yield too large and not dense-enough subgraphs.

## 6 Case study

We present a case study using graphs of co-occurring hashtags from Twitter messages in the Helsinki region. We create two subsets of *Twitter#* dataset: one covering all tweets in November 2013 and another in December 2013. November dataset consists of 4758 interactions, 917 nodes, and the corresponding static graph has average degree density 3.546. December dataset has 5559 interactions, 1039 nodes, and the density is 3.290.

Figures 10 and 11 show the dense subgraphs discovered by the kGApprox algorithm on these datasets, with $k = 4$ and $\epsilon_{\text{DS}} = \epsilon_{\text{DP}} = 0.1$.

For the November dataset, kGApprox creates a small 1-day interval in the beginning and then splits the rest time almost evenly. This first interval includes the nodes `movember`, `liiga`, `halloween`, and `digiexpo`, which cover a broad range of global (e.g., movember and Halloween) and local events (e.g., game industry event DigiExpo and Finnish ice hockey league). The next interval is represented by a large variety of well-connected tags related to `mtv` and media, corresponding to the MTV Europe Music Awards 13 on November 10. There are also other ice hockey-related tags, e.g., `leijonat` and Father's Day tags, e.g., `isänpäivä`, which was on November 13. The third interval is mostly represented by Slush-related tags; Slush is the annual large startup and tech event in Helsinki. The last interval is completely dedicated to ice hockey with many team names.

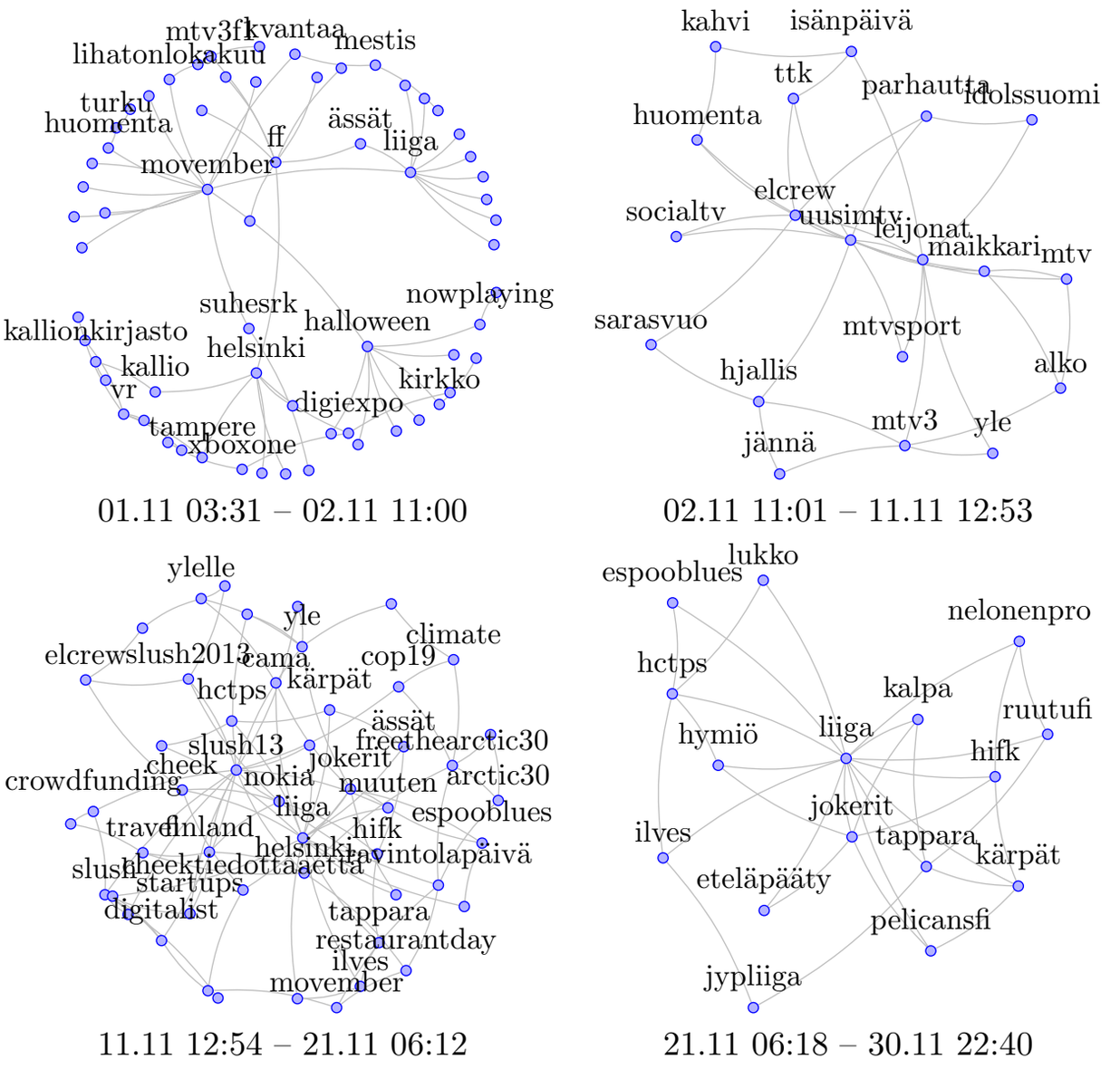



**Fig. 10** Subgraphs, discovered in the network of Twitter hashtags *Twitter#* from November 2013 kGAPPROX algorithm with $k = 4$, $\epsilon_{DS} = \epsilon_{DP} = 0.1$

There are three major public holidays in December: Finland's Independence Day on December 6, Christmas on December 25, and New Year's Eve on December 31. kGAPPROX allocates one interval for Christmas and New Year from December 21 to 31. Ice hockey is also represented in this interval, as well as in the third interval. Remarkably, the Independence Day holiday is split into two intervals. The first one is from December 1 to December 6, 3:30pm, and the corresponding graph has two clusters: the first one contains general holidays-related tags and the second one is focused on Independence Day President's reception. (Itsenäisyyspäivän vastaanotto or colloquially Linnan juhlat/Slotts balen). This is a large event that starts on December 6, 6pm, is broadcasted live, and is discussed in media for the following days. The second interval for December 6–9 is a truthful representation of this event.

To demonstrate the qualitative performance of kGAPPROX for different parameters, we consider three parameters settings: $case_1$: $\epsilon_{DS} = 0.1, \epsilon_{DP} = 0.1$; $case_2$: $\epsilon_{DS} = 0.01, \epsilon_{DP} = 0.1$; and $case_3$: $\epsilon_{DS} = 0.1$, $\epsilon_{DP} = 0.01$. Table 5 shows the characteristics of the solution graphs ($H_1$, $H_2$, $H_3$, $H_4$) discovered in the different settings.

The first two rows show the average degree density and the number of nodes (size) of each graph. Rows 3 and 4 compare an $i$th graph in one solution (i.e., in one parameters

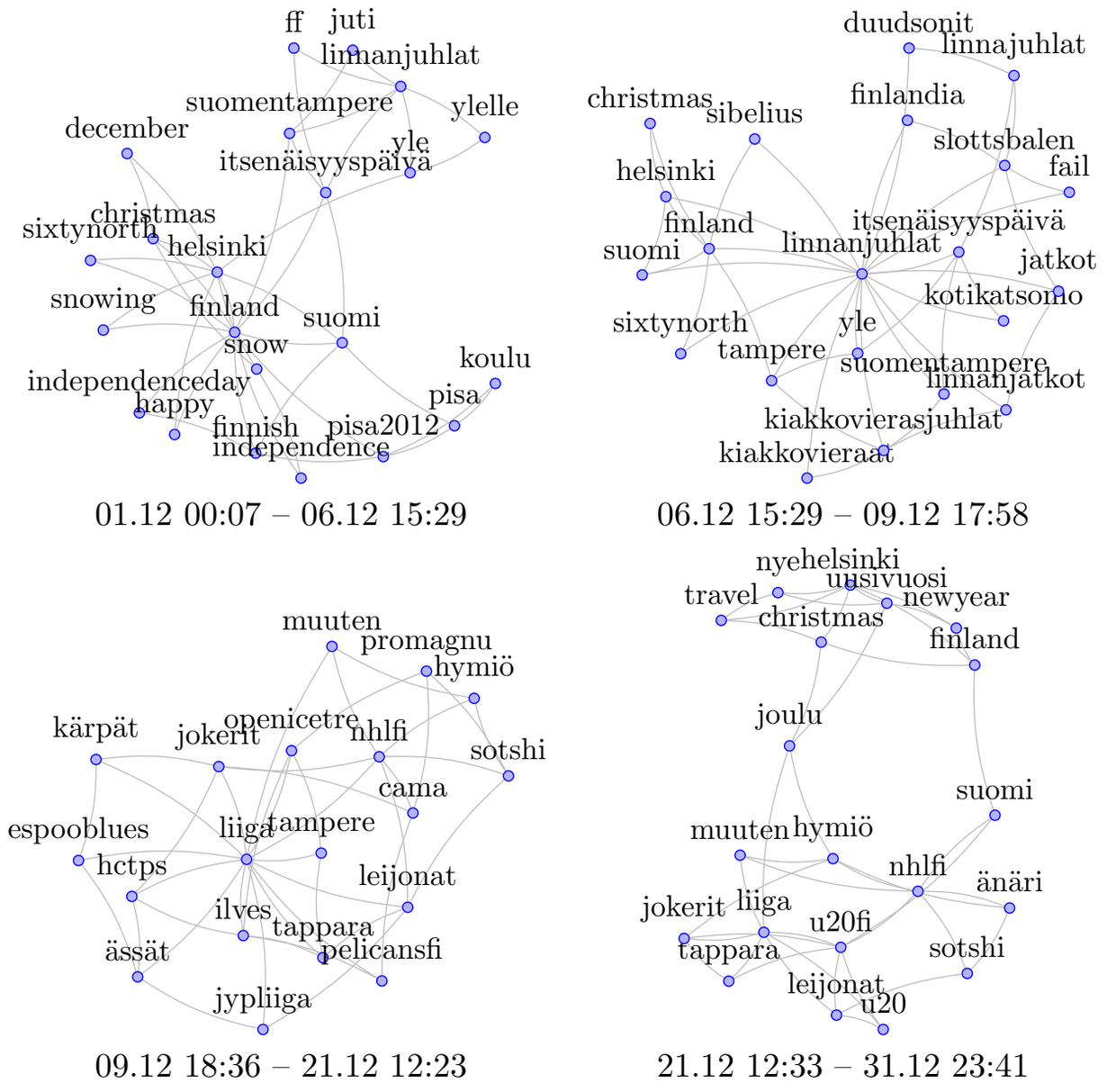



**Fig. 11** Subgraphs, discovered in the network of Twitter hashtags *Twitter#* from December 2013 by KGAPPROX algorithm with $k = 4$, $\epsilon_{\text{DS}} = \epsilon_{\text{DP}} = 0.1$

setting) with the $i$th graph in other solutions (i.e., in other parameters setting). We report the average overlap in nodes and average Jaccard similarity of node sets. Larger overlap and larger Jaccard similarity values provide evidence that the algorithm outputs similar $i$-th episode graphs for different settings. For the November datasets, the first two episodes are identical for all settings. Episodes 3 and 4 are similar for cases $case_1$ and $case_3$, but different for $case_2$: As it is discussed before, the change in the densest subgraph search contributes to the change in the solution. There is a similar trend for the December dataset, although the similarity values are typically lower.

Rows 5 and 6 present the similarities between the graphs in *one* solution. We compare an $i$-th graph in a solution to all other episode graphs in that solution. We report an average overlap in nodes and average Jaccard similarity of node sets. Lower overlap and smaller Jaccard similarity values indicate that the graphs in the solution differ. All similarity values for both datasets are quite low. Although the average overlap in nodes can be as high as 7.333, such an overlap is not prominent when the sizes of the graphs are taken into consideration, as it is shown by the Jaccard similarity metric.

**Table 5** Characteristics of the episode graphs $H_i$ discovered for different parameters of $\epsilon_{DS}$ and $\epsilon_{DP}$ in the case-study dataset

| Metric | November | | | | December | | | |
|---|---|---|---|---|---|---|---|---|
| | $H_i$ | $case_1$ | $case_2$ | $case_3$ | $H_i$ | $case_1$ | $case_2$ | $case_3$ |
| Density | $H_1$ | 1.8 | 1.8 | 1.8 | $H_1$ | 3.647 | 4.857 | 3.647 |
| | $H_2$ | 3.487 | 3.487 | 3.487 | $H_2$ | 3.778 | 3.867 | 3.778 |
| | $H_3$ | 3.563 | 5.091 | 3.563 | $H_3$ | 4.4 | 3.6 | 2.667 |
| | $H_4$ | 4.0 | 1.333 | 4.0 | $H_4$ | 4.1 | 3.0 | 5.302 |
| Size | $H_1$ | 10 | 10 | 10 | $H_1$ | 17 | 14 | 17 |
| | $H_2$ | 39 | 39 | 39 | $H_2$ | 19 | 30 | 9 |
| | $H_3$ | 32 | 22 | 32 | $H_3$ | 35 | 10 | 6 |
| | $H_4$ | 9 | 3 | 9 | $H_4$ | 20 | 6 | 43 |
| Avg. overlap (across the solutions) | $H_1$ | 10 | 10 | 10 | $H_1$ | 14.5 | 12.0 | 14.5 |
| | $H_2$ | 39 | 39 | 39 | $H_2$ | 6.0 | 3.0 | 6.0 |
| | $H_3$ | 22 | 12 | 22 | $H_3$ | 6.5 | 4.5 | 3.0 |
| | $H_4$ | 4.5 | 0 | 4.5 | $H_4$ | 12.0 | 5.5 | 12.5 |
| Avg. Jac. similarity (across the solutions) | $H_1$ | 1.0 | 1.0 | 1.0 | $H_1$ | 0.816 | 0.632 | 0.816 |
| | $H_2$ | 1.0 | 1.0 | 1.0 | $H_2$ | 0.541 | 0.083 | 0.542 |
| | $H_3$ | 0.643 | 0.285 | 0.643 | $H_3$ | 0.178 | 0.141 | 0.103 |
| | $H_4$ | 0.5 | 0.0 | 0.5 | $H_4$ | 0.335 | 0.189 | 0.286 |
| Avg. overlap (inside the solution) | $H_1$ | 3.667 | 3.0 | 3.667 | $H_1$ | 5.333 | 1.333 | 4.0 |
| | $H_2$ | 4.333 | 4.333 | 4.333 | $H_2$ | 4.667 | 5.333 | 3.667 |
| | $H_3$ | 5.0 | 4.667 | 5.0 | $H_3$ | 7.333 | 3.667 | 1.333 |
| | $H_4$ | 3.0 | 0 | 3.0 | $H_4$ | 6.667 | 2.333 | 4.333 |
| Avg. Jac. similarity (inside the solution) | $H_1$ | 0.127 | 0.091 | 0.127 | $H_1$ | 0.199 | 0.033 | 0.153 |
| | $H_2$ | 0.086 | 0.087 | 0.081 | $H_2$ | 0.195 | 0.158 | 0.151 |
| | $H_3$ | 0.108 | 0.119 | 0.108 | $H_3$ | 0.172 | 0.160 | 0.030 |
| | $H_4$ | 0.113 | 0.0 | 0.113 | $H_4$ | 0.182 | 0.119 | 0.088 |

$case_1$: $\epsilon_{DS} = 0.1, \epsilon_{DP} = 0.1$; $case_2$: $\epsilon_{DS} = 0.01, \epsilon_{DP} = 0.1$; $case_3$: $\epsilon_{DS} = 0.1, \epsilon_{DP} = 0.01$

We can conclude that for all parameters, the solution for the case study consists of diverse graphs. However, changing the accuracy of the densest subgraphs search may lead to differences in the output episodes graphs.

## 7 Related work

Partitioning a graph in dense subgraphs is a well-established problem. Many of the existing works adopt as density definition the average-degree notion [2,23,33,50]. The densest subgraph, under this definition, can be found in polynomial time [27]. Moreover, there is a 2-approximation greedy algorithm by Charikar [15] and Asahiro et al. [4], which runs in linear time of the graph size. Many recent works develop methods to maintain the average-degree densest subgraph in a streaming scenario [14,19,20,38,39]. Alternative density definitions,

such as variants of quasi-clique, are often hard to approximate or solve by efficient heuristics due to connections to **NP**-complete Maximum Clique problem [1,37,49].

A line of work focuses on dynamic graphs, which model node/edge additions/deletions. Different aspects of network evolution, including evolution of dense groups, were studied in this setting [6,11,34,41]. However, here we use the interaction network model, which is different to dynamic graphs, as it captures the instantaneous interactions between nodes.

Another classic approach to model temporal graphs is to consider graph snapshots, find structures in each snapshot separately (or by incorporating information from previous snapshots), and then summarize historical behavior of the discovered structures [5,12,28,36,40]. These approaches usually focus on the temporal coherence of the dense structures discovered in the snapshots and assume that the snapshots are given. In this work, we aggregate instantaneous interaction into timeline partitions of arbitrary lengths.

To the best our knowledge, the following works are better aligned with our approach. A work of Rozenshtein et al. [44] considers a problem of finding the densest subgraph in a temporal network. However, first, they do not aim at creating a temporal partitioning. Second, they are interested in finding a single dense subgraph whose edges occur in $k$ short time intervals. On the contrary, in this work we search for an interval partitioning and consider only graphs that are span continuous intervals. Other close works are by Jethava and Beerenwinkel [31] and Semertzidis et al. [46]. However, these works consider a set of snapshots and search for a single heavy subgraph induced by one or several intervals. The work of Semertzidis et al. [46] explores different formulations for the *persistent heavy subgraph problem*, including maximum average density, while Jethava and Beerenwinkel [31] focus solely on maximum average density.

## 8 Conclusions

In this work, we consider the problem of finding a sequence of dense subgraphs in a temporal network. We search for a partition of the network timeline into $k$ non-overlapping intervals, such that the intervals span subgraphs with maximum total density. To provide a fast solution for this problem, we adapt recent work on dynamic densest subgraph and approximate dynamic programming. In order to ensure that the episodes we discover consist of a diverse set of nodes, we adjust the problem formulation to encourage coverage of a larger set of nodes. While the modified problem is **NP**-hard, we provide a greedy heuristic, which performs well on empirical tests.

The problems of temporal event detection and timeline segmentation can be formulated in various ways depending on the type of structures that are considered to be interesting. Here, we propose segmentation with respect to maximizing subgraph density. The intuition is that those dense subgraphs provide a sequence of interesting events that occur in the lifetime of the temporal network. However, other notions of interesting structures, such as frequency of the subgraphs, or statistical non-randomness of the subgraphs, can be considered for future work. In addition, it could be meaningful to allow more than one structure per interval. Another possible extension is to consider overlapping intervals instead of a segmentation.

**Acknowledgements** Open access funding provided by Aalto University. Part of this work was done while the first author was visiting ISI Foundation. This work was partially supported by three Academy of Finland Projects (286211, 313927, and 317085) and the EC H2020 RIA Project "SoBigData" (654024). We thank the anonymous reviewers for their valuable comments.

## A Supporting proofs

For the ease of notations, let us write $[t]$ for the one-time stamp interval $[t, t]$.

**Proposition 8** *The total profit of optimal solution of* $k$- DENSEST- EPISODES *is a strictly increasing function of the number of segments* $k$ *and reaches its maximum when* $k$ *is equal to the number of intervals.*

***Proof*** Given a temporal graph $G = (V, \mathcal{T}, E)$ with $m$ time stamps,

let $S^k = \{(I_\ell^k, H_\ell^k)\}$, for $\ell = 1, \ldots, k$ and be the solution for $k$- DENSEST- EPISODES for some $k < m$. Let $S^{k'} = \left\{(I_\ell^{k'}, H_\ell^{k'})\right\}$, for $\ell = 1, \ldots, k'$ be the solution for $k$- DENSEST- EPISODES for some other $k' = k + 1$.

We will show that the profit value for $S^k$ is less than the profit value for $S^{k'}$: $\sum_{\ell=1}^{k} d(H_\ell^k) < \sum_{\ell=1}^{k'} d(H_\ell^{k'})$.

Denote the profit of a solution $S$ as $Pr(S)$.

Fix some $\ell \in [1, \ldots, k]$ so that the corresponding episode $((I_\ell^k, H_\ell^k))$ of $S^k$ has the interval $I_\ell^k$ with more than one time stamp. By the problem definition, $H_\ell^k$ is the densest subgraph of the interval $I_\ell^k$. Let $(E^*, V^*)$ be edges and nodes of $H_\ell^k$. Now consider an arbitrary split of $I_\ell^k$ into $L$ and $R$ non-empty intervals and construct a new sequence of episodes $S'$ with $k+1$ episode, which is the same as $S^k$ except for the split episode $I_\ell^k$.

Let $E^*[L]$ be the subset of $E^*$, which appear in $L$, and $E^*[R]$ be the subset of $E^*$, which appear in $R$. Similarly, define $V^*[L]$ and $V^*[R]$.

Now the following is true for the density of the densest subgraph in $L$:

$d^*(G[L]) \geq 2\frac{|E^*[L]|}{|V^*[L]|} \geq 2\frac{|E^*[L]|}{|V^*|}$.

The same inequality can be written for $d^*(G[R])$.

Also $|E^*[L]| + |E^*[R]| \geq |E^*|$.

Thus,

$d^*(G[L]) + d^*(G[R]) \geq 4\frac{|E^*[L]|+|E^*[R]|}{|V^*|} \geq 4\frac{|E^*|}{|V^*|} = 2d^*(G[I_\ell^k]) > d^*(G[I_\ell^k])$.

This leads to the larger profit of segmentation $S'$: $Pr(S^k) < Pr(S')$.

Splitting the interval $I_\ell^k$ into $L$ and $R$ gives a $k+1$ segmentation, which profit is by optimality of $S^{k'}$ should be not larger than the profit of $S^{k'}$: $Pr(S') \leq Pr(S^{k'})$.

Thus, $Pr(S^k) < Pr(^{k'})$ and we conclude the proof. □

Similar statement can be proven for $k$- DENSEST- EPISODES- EC in the same way: As cover is a nonnegative and non-decreasing function of the subgraphs (Proposition 6), splitting an episode is still always beneficial.

## References

1. Alvarez-Hamelin J I, Dall'Asta L, Barrat A, Vespignani A (2006) Large scale networks fingerprinting and visualization using the k-core decomposition. In: NIPS
2. Andersen R, Chellapilla K (2009) Finding dense subgraphs with size bounds. In: WAW, pp 25–37

3. Angel A, Sarkas N, Koudas N, Srivastava D (2012) Dense subgraph maintenance under streaming edge weight updates for real-time story identification. PLVDB 5(6):574–585
4. Asahiro Y, Iwama K, Tamaki H, Tokuyama T (2000) Greedily finding a dense subgraph. J Algorithms 34(2):203–221
5. Asur S, Parthasarathy S, Ucar D (2009) An event-based framework for characterizing the evolutionary behavior of interaction graphs. TKDD 3(4):16
6. Backstrom L, Huttenlocher D, Kleinberg J, Lan X (2006) Group formation in large social networks: membership, growth, and evolution. In: KDD, pp 44–54
7. Balalau OD, Bonchi F, Chan T, Gullo F, Sozio M (2015) Finding subgraphs with maximum total density and limited overlap. In: WSDM, pp 379–388
8. Balalau O D, Castillo C, Sozio M (2018) Evidense: a graph-based method for finding unique high-impact events with succinct keyword-based descriptions. In: Proceedings of the twelfth international conference on web and social media, ICWSM, pp 560–563
9. Bellman R (2013) Dynamic programming, Courier Corporation
10. Bellman R, Kotkin B (1962) On the approximation of curves by line segments using dynamic programming. II, Technical report, RAND CORP SANTA MONICA CALIF
11. Berlingerio M, Bonchi F, Bringmann B, Gionis A (2009) Mining graph evolution rules. In: ECML PKDD, pp 115–130
12. Berlingerio M, Pinelli F, Calabrese F (2013) Abacus: frequent pattern mining-based community discovery in multidimensional networks. DMKD 27(3):294–320
13. Beutel A, Xu W, Guruswami V, Palow C, Faloutsos C (2013) Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In: WWW, pp 119–130
14. Bhattacharya S, Henzinger M, Nanongkai D, Tsourakakis C (2015) Space-and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In: STOC, pp 173–182
15. Charikar M (2000) Greedy approximation algorithms for finding dense components in a graph. In: APPROX, pp 84–95
16. Chen J, Saad Y (2012) Dense subgraph extraction with application to community detection. TKDE 24(7):1216–1230
17. Danisch M, Chan T H, Sozio M (2017) Large scale density-friendly graph decomposition via convex programming. In: Proceedings of the 26th international conference on World Wide Web, WWW 2017
18. DiTursi D, Ghosh G, Bogdanov P (2017) Local community detection in dynamic networks. arXiv:1709.04033
19. Epasto A, Lattanzi S, Sozio M (2015) Efficient densest subgraph computation in evolving graphs. In: WWW, pp 300–310
20. Esfandiari H, Hajiaghayi M, Woodruff D (2015) Applications of uniform sampling: Densest subgraph and beyond. arXiv:1506.04505
21. Feder T, Motwani R (1995) Clique partitions, graph compression and speeding-up algorithms. JCSS 51(2):261–272
22. Fratkin E, Naughton BT, Brutlag DL, Batzoglou S (2006) Motifcut: regulatory motifs finding with maximum density subgraphs. Bioinformatics 22(14):e150–e157
23. Galbrun E, Gionis A, Tatti N (2014) Overlapping community detection in labeled graphs. DMKD 28(5–6):1586–1610
24. Galbrun E, Gionis A, Tatti N (2016) Top-*k* overlapping densest subgraphs. DMKD 30(5):1134–1165
25. Gallo G, Grigoriadis MD, Tarjan RE (1989) A fast parametric maximum flow algorithm and applications. SIAM J. Comput. 18:30–55
26. Gibson D, Kumar R, Tomkins A (2005) Discovering large dense subgraphs in massive graphs. In: PVLDB, pp 721–732
27. Goldberg A V (1984) Finding a maximum density subgraph, University of California Berkeley
28. Greene D, Doyle D, Cunningham P (2010) Tracking the evolution of communities in dynamic social networks. In: ASONAM
29. Guha S, Koudas N, Shim K (2001) Data-streams and histograms. In: STOC, pp 471–475
30. Hernández C, Navarro G (2012) Compressed representation of web and social networks via dense subgraphs. In: SIGIR, pp 264–276
31. Jethava V, Beerenwinkel N (2015) Finding dense subgraphs in relational graphs. In: ECML PKDD, pp 641–654
32. Karande C, Chellapilla K, Andersen R (2009) Speeding up algorithms on compressed web graphs. Internet Math 6:373–398
33. Khuller S, Saha B (2009) On finding dense subgraphs. In: ICALP
34. Li R-H, Yu JX, Mao R (2014) Efficient core maintenance in large dynamic graphs. TKDE 26(10):2453–2465

35. Lin H, Bilmes J (2011) A class of submodular functions for document summarization. In: ACL, pp 510–520
36. Lin Y-R, Chi Y, Zhu S, Sundaram H, Tseng B L (2008) Facetnet: a framework for analyzing communities and their evolutions in dynamic networks. In: WWW, pp 685–694
37. Makino K, Uno T (2004) New algorithms for enumerating all maximal cliques. In: SWAT, pp 260–272
38. McGregor A, Tench D, Vorotnikova S, Vu HT (2015) Densest subgraph in dynamic graph streams. In: MFCS. Springer, Berlin
39. Mitzenmacher M, Pachocki J, Peng R, Tsourakakis C, Xu SC (2015) Scalable large near-clique detection in large-scale networks via sampling. In: KDD, pp 815–824
40. Mucha PJ, Richardson T, Macon K, Porter MA, Onnela J-P (2010) Community structure in time-dependent, multiscale, and multiplex networks. Science 328(5980):876–878
41. Myers SA, Leskovec J (2014) The bursty dynamics of the twitter information network. In: WWW, pp 913–924
42. Nemhauser G, Wolsey L, Fisher M (1978) An analysis of approximations for maximizing submodular set functions. Math Progr 14(1):265–294
43. Orlin JB (2013) Max flows in o (nm) time, or better. In: Proceedings of the forty-fifth annual ACM symposium on Theory of computing
44. Rozenshtein P, Tatti N, Gionis A (2017) Finding dynamic dense subgraphs. TKDD 11(3):27
45. Saha B, Hoch A, Khuller S, Raschid L, Zhang X-N (2010) Dense subgraphs with restrictions and applications to gene annotation graphs. In: RECOMB
46. Semertzidis K, Pitoura E, Terzi E, Tsaparas P (2018) Finding lasting dense subgraphs. Data Mining and Knowledge Discovery
47. Tatti N (2018) Strongly polynomial efficient approximation scheme for segmentation. arXiv:1805.11170
48. Taylor D, Caceres RS, Mucha PJ (2017) Super-resolution community detection for layer-aggregated multilayer networks. Phys Rev X 7(3):031056
49. Tsourakakis C, Bonchi F, Gionis A, Gullo F, Tsiarli M (2013) Denser than the densest subgraph: extracting optimal quasi-cliques with quality guarantees. In: KDD, pp 104–112
50. Tsourakakis CE (2014) A novel approach to finding near-cliques: the triangle-densest subgraph problem. arXiv:1405.1477
51. Viswanath B, Mislove A, Cha M, Gummadi K (2009) On the evolution of user interaction in facebook, In: WOSN, pp 37–42



**Polina Rozenshtein** received a M.Sc. degree and an Ph.D. degree from Aalto University, Espoo, Finland, in 2014 and 2018. She is currently a Senior Data Scientist at Nordea Data Science Lab, Helsinki, Finland. Prior to that, she was a postdoctoral researcher in Data Mining Group of Computer Science Department at Aalto University. Her research interests include data mining, combinatorial optimization, dynamic graph mining, social networks analysis, computational social science, and data analysis for social good.


**Francesco Bonchi** is Deputy Director at the ISI Foundation, Turin, Italy, with responsibility over the Industrial Research area. At ISI Foundation, he is also Research Leader for the "Algorithmic Data Analytics" group. He is also (part-time) Research Director for Big Data & Data Science at Eurecat (Technological Center of Catalunya), Barcelona. Previously, he was Director of Research at Yahoo Labs Barcelona, where he was leading the Web Mining Research group. He has been the General Chair of IEEE DSAA 2018, PC Chair of ECML PKDD'18, ACM HT'17, IEEE ICDM'16, ECML PKDD 2010. He is a member of the Steering Committee of ECML PKDD and IEEE DSAA and associate editor of many journals in the data management and mining area (IEEE TBD, IEEE TKDE, ACM TKDD, ACM TIST, DMKD). More information at http://www.francescobonchi.com/.

**Aristides Gionis** is a professor in the Department of Computer Science in Aalto University. He is currently a fellow in the ISI foundation, Turin, while he has been a visiting professor in the University of Rome. Previously, he has been a senior research scientist and group leader in Yahoo! Research, Barcelona. He obtained his Ph.D. in 2003 from Stanford University, USA. He is currently serving as an action editor in the Data Management and Knowledge Discovery Journal (DMKD), an associate editor in the ACM Transactions on Knowledge Discovery from Data (TKDD), and an associate editor in the ACM Transactions on the Web (TWEB). He has contributed in several areas of data science, such as algorithmic data analysis, Web mining, social media analysis, data clustering, and privacy-preserving data mining. His current research is funded by the Academy of Finland (Projects Nestor, Agra, AIDA) and the European Commission (Project SoBigData).

**Mauro Sozio** is currently associate professor in the Department of Computer Science at Telecom ParisTech University in Paris, France. Previously, he held a visiting scientist position at IBM Almaden (USA) and a senior researcher position at the Max-Planck Institute for Informatics (Germany). He received his Ph.D. in computer science from "Sapienza" University of Rome in 2007. He has contributed in several research areas of computer science, such as graph mining and social network analysis, approximation algorithms, and distributed algorithms. He serves or he has served as PC or senior PC member in top venues for data mining, the Web, and databases such as TheWebConf, KDD, PVLDB, ICDM, and others where he has also published more than 20 research papers.

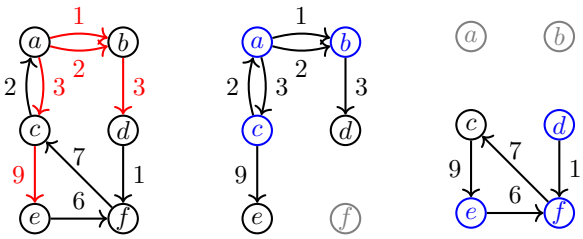
**Nikolaj Tatti** is an associate professor at University of Helsinki. Previously, he was a senior data scientist at F-Secure, a HIIT research fellow in Aalto University, and an FWO postdoctoral fellow in University of Antwerp. He received his Ph.D. in 2008 from Helsinki University of Technology, Finland. His current research interest is developing and analyzing new data mining methodology with diverse applications. He has published over 60 peer-reviewed papers in top data mining conferences and journals.

# An Index For Temporal Closeness Computation in Evolving Graphs

Lutz Oettershagen* Petra Mutzel*

**Abstract**

Temporal closeness is a generalization of the classical closeness centrality measure for analyzing evolving networks. The temporal closeness of a vertex $v$ is defined as the sum of the reciprocals of the temporal distances to the other vertices. Ranking all vertices of a network according to the temporal closeness is computationally expensive as it leads to a single-source-all-destination (SSAD) temporal distance query starting from each vertex of the graph. To reduce the running time of temporal closeness computations, we introduce an index to speed up SSAD temporal distance queries called *Substream* index. We show that deciding if a *Substream* index of a given size exists is NP-complete and provide an efficient greedy approximation. Moreover, we improve the running time of the approximation using min-hashing and parallelization. Our evaluation with real-world temporal networks shows a running time improvement of up to one order of magnitude compared to the state-of-the-art temporal closeness ranking algorithms.




(a) $\mathcal{G}$ (b) $\mathcal{G}_1$ (c) $\mathcal{G}_2$

Figure 1: (a) Temporal graph $\mathcal{G}$ with availability times shown at the edges. The transition times are one for all edges. The edge stream $\xi(a)$ is highlighted red; it contains all edges that can be part of a temporal walk starting at $a$. (b) Temporal subgraph $\mathcal{G}_1$ of $\mathcal{G}$ contains all edges, such that the temporal closeness of the vertices $a$, $b$, and $c$ can be computed. (c) Temporal subgraph $\mathcal{G}_2$ contains all edges, such that the temporal closeness for $d$, $e$, or $f$ can be determined. Vertices that can not be reached are colored gray.

## 1 Introduction

Computing closeness centrality is an essential task in network analysis and data mining [2, 18, 24, 26]. In this work, we focus on improving the efficiency of computing temporal closeness in evolving networks. We represent evolving networks using *temporal graphs*, which consists of a finite set of vertices and a finite set of temporal edges. Each temporal edge is only available at a specific discrete point in time, and edge transition takes a strictly positive amount of time. Temporal graphs are often good models for real-life scenarios due to the inherently dynamic nature of most real-world activities and processes. For example, temporal graphs are used to model and analyze bioinformatics networks [12, 21], communication networks [4, 7], contact networks [5, 17], social networks [10, 16], and transportation networks [22].

Temporal closeness is one of the popular and essential centrality measures for temporal networks, and various variants of temporal closeness have been discussed [14, 19, 26, 24]. Here, we consider one of the standard variants, the *harmonic temporal closeness* of a vertex, which is defined as the sum of the reciprocals of the *durations* of the fastest paths to all other vertices [18]. Unfortunately, the computation with respect to the minimum duration distance is expensive and can be prohibitive for large temporal networks [18, 29]. To overcome this obstacle, we introduce an index for efficiently answering single-source-all-destination (SSAD) temporal distance queries in temporal graphs.

**Our idea:** We exploit the often very limited reachability in temporal graphs. Given a temporal graph, we construct $k$ smaller (possibly non-disjoint) subgraphs. We guarantee that for each vertex of the input graph, there exists one of the $k$ subgraphs, such that temporal distance queries, and hence its temporal closeness, can be answered with a single pass over the chronologically ordered edges of the subgraph using a state-of-the-art streaming algorithm introduced in [29]. For example, Figure 1 shows a temporal graph (a), for which the temporal distance queries starting from vertices $a$, $b$, or $c$ can be answered using only the temporal subgraph shown in (b) and starting from any of the remaining vertices using only the temporal subgraph shown in (c).

It is insufficient to store the temporal graph in an adjacency list representation and compute the minimum

*Institute of Computer Science, University of Bonn, {lutz.oettershagen,petra.mutzel}@cs.uni-bonn.de

duration distance using label setting algorithms. The streaming approach for computing the temporal distances is often already significantly faster [18, 29].

**Contributions:**

1. We propose the *Substream* index for temporal closeness computation in temporal graphs. We show that deciding if a *Substream* index of a given size can be constructed is an NP-complete problem.
2. We introduce an efficient approximation for constructing a *Substream* with guarantees on the resulting index size. Next, we improve our approximation with min-hashing and shared-memory parallelization to speed up the index construction.
3. In our evaluation on real-world temporal graphs, we show that our approach achieves up to an order of magnitude faster temporal closeness computation times (indexing + querying) compared to the state-of-the-art algorithms.

## 2 Preliminaries

We use $\mathbb{N}$ to denote the strictly positive integers. For $\ell \in \mathbb{N}$, we denote with $[\ell]$ the set $\{1, \ldots, \ell\}$. A *directed temporal graph* $\mathcal{G} = (V, \mathcal{E})$ consists of a finite set of vertices $V$ and a finite set $\mathcal{E}$ of directed *temporal edges* $e = (u, v, t, \lambda)$ with *tail* $u$ and *head* $v$ in $V$, *availability time* (or *timestamp*) $t \in \mathbb{N}$ and *transition time* $\lambda \in \mathbb{N}$. The transition time of an edge denotes the time required to traverse the edge. We only consider directed temporal graphs—it is possible to model undirectedness by using a forward- and a backward-directed edge with equal timestamps and transition times for each undirected edge. We call a sequence $S = (e_1, \ldots, e_m)$ of $m$ temporal edges with non-decreasing availability times (ties are broken arbitrarily) a *temporal edge stream*. A temporal edge stream $S$ induces a temporal graph $\mathcal{G} = (V(S), S)$ with $V(S) = \{u, v \mid (u, v, t, \lambda) \in S\}$, where we, for notational convenience, interpret the sequence $S$ as a set of edges. Given a temporal edge stream $S$, we denote with $n$ the number of vertices of the induced temporal graph, and with $n^+$ the number of vertices with at least one outgoing edge, i.e., non-sink vertices. Let $S$ and $S'$ be temporal edge streams and $S' \subseteq S$, i.e., $S'$ contains only edges from $S$. We call $S'$ a *substream* of $S$. If it is clear from the context, we use the view of a temporal graph $\mathcal{G}$ and the corresponding edge stream interchangeably. The size of an edge stream $S$ consisting of $m$ edges is $|S| = m$. We assume that $m \geq \frac{n}{2}$, which holds unless isolated vertices exist, to simplify the discussion of running time complexities. Note that edge streams do not have isolated vertices. Let $S_1$ and $S_2$ be two temporal edge streams, we denote by $S_3 = S_1 \cup S_2$ the union of the two temporal edges streams and $S_3$ is a temporal edge stream, i.e., the edges of $S_3$ are ordered in non-decreasing order of their availability time. $S_3$ can be computed in $\mathcal{O}(|S_1| + |S_2|)$ time due to the ordering of the edges in non-decreasing availability times. We denote the lifetime of a temporal graph $\mathcal{G} = (V, \mathcal{E})$ (or edge stream $S$) with $T(\mathcal{G}) = [t_{min}, t_{max}]$ with $t_{min} = \min\{t \mid e = (u, v, t, \lambda) \in \mathcal{E}\}$ and $t_{max} = \max\{t + \lambda \mid e = (u, v, t, \lambda) \in \mathcal{E}\}$.

**Temporal Distance and Closeness** A *temporal walk* between vertices $v_1, v_{\ell+1} \in V$ of length $\ell$ is a sequence of $\ell$ temporal edges $(e_1 = (v_1, v_2, t_1, \lambda_1), e_2 = (v_2, v_3, t_2, \lambda_2), \ldots, e_\ell = (v_\ell, v_{\ell+1}, t_\ell, \lambda_\ell))$ such the head of $e_i$ equals the tail of $e_{i+1}$, and $t_i + \lambda_i \leq t_{i+1}$ for $1 \leq i < \ell$. A *temporal path* $P$ is a temporal walk in which each vertex is visited at most once. The *starting time* of $P$ is $s(P) = t_1$, the *arrival time* is $a(P) = t_\ell + \lambda_\ell$, and the *duration* is $d(P) = a(P) - s(P)$. A *minimum duration* or *fastest* $(u, v)$-path is a path from $u$ to $v$ with shortest duration among all paths from $u$ to $v$. The harmonic temporal closeness is defined in terms of minimum duration.

Definition 1. (Harmonic Temporal Closeness) *Let $\mathcal{G} = (V, \mathcal{E})$ be a temporal graph. We define the harmonic temporal closeness for $u \in V$ as* $c(u) = \sum_{v \in V \setminus \{u\}} \frac{1}{d(u,v)}$.

If $v$ is not reachable from vertex $u$, we set $d(u, v) = \infty$, and we define $\frac{1}{\infty} = 0$.

**Temporal Reachability** We say vertex $v$ is *reachable* from vertex $u$ if there exists a temporal $(u, v)$-path. We denote with $\xi(v)$ the subset of edges that can be used by any temporal walk starting at $v$ ordered in non-decreasing availability times (ties are broken arbitrarily), i.e., $\xi(v)$ is a temporal edge stream. For example, in Figure 1a the edges of $\xi(a)$ are highlighted in red. Temporal graphs are, in general, not strongly connected and have limited reachability with respect to temporal paths due to the missing symmetry and transitivity.

**Restrictive Interval** Temporal distance, reachability, and closeness computations can be additionally restricted to a time interval $\tau = [a, b]$ such that only edges $e = (u, v, t, \lambda) \in \mathcal{E}$ are considered that start and arrive in the interval $\tau$, i.e., $a \leq t < b$ and $a < t + \lambda \leq b$.

## 3 Substream Index

The substream index constructs $k$ temporal subgraphs from a given temporal graph, leveraging the following simple observation.

Observation 1. *The temporal durations between vertex $v \in V$ and all other vertices, can be determined solely with edges in $\xi(v)$.*

Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$ in its edge stream representation $S$ and $2 \leq k < n$, we first construct $k$ substreams $S_1, \ldots, S_k$ of $S$. Each vertex $v \in V$ is assigned to exactly one of the new substreams, such that the substream contains all edges that can be used by any temporal walk leaving $v$. We use an additional empty stream $S_0 = \emptyset$ to which we assign all sink-vertices, i.e., vertices with no outgoing edges. The substreams and the vertex assignment together form the substream index, which we define as follows.

DEFINITION 2. *Let $S$ be a temporal edge stream and $k \geq 2 \in \mathbb{N}$. We define the pair $\mathcal{I} = (\mathcal{S}, f)$ with $\mathcal{S} = \{S_0, S_1, \ldots, S_k\}$, $S_i \subseteq S$ for $1 \leq i \leq k$, $S_0 = \emptyset$, and $f : V \to [k] \cup \{0\}$ as* substream index*, with $f$ maps $v \in V(S)$ to an index $i$ of subset $S_i \in \mathcal{S}$, such that $\xi(v) \subseteq S_i$.*

A pair $(v, \tau)$, with $v \in V$ and $\tau$ is a restrictive time interval is a query to the substream index. We answer it by running the fastest paths streaming algorithm from [29], on the substream $S_{f(v)}$ for vertex $v$ and restricting time interval $\tau$. The streaming algorithm uses a single pass over the edges in the substream $S_{f(v)}$. Next, we define the size of a substream index as the maximum substream size.

DEFINITION 3. *Let $\mathcal{I} = (\mathcal{S}, f)$ be a substream index. The size of $\mathcal{I}$ is $size(\mathcal{I}) = \max_{S \in \mathcal{S}}\{|S|\}$.*

Before discussing the query times, we bound the number of vertices that can be assigned to a substream.

LEMMA 3.1. *The maximal number of vertices assigned to any substream $S \in \mathcal{S}$ is at most $2 \cdot size(\mathcal{I})$*

*Proof.* For $i \in [k]$, the number of vertices occurring in $S_i$ is $|\{u, v \mid (u, v, t, \lambda) \in S_i\}| \leq 2|S_i|$. Hence, the maximal number of vertices assigned to any substream $S_i$ is at most $2 \cdot size(\mathcal{I})$. ∎

We now discuss the query time of the substream index.

THEOREM 3.1. *Let $\mathcal{I}$ be a substream index, $\Delta = size(\mathcal{I})$, and let $\delta$ be the maximal in-degree of a vertex in any of the substreams. Given a query $(u, \tau)$, let $\sigma_+(u)$ the set of availability times of edges leaving the query vertex $u$, and $\rho = \min\{\delta, |\sigma_+(u)|\}$. Answering a fastest path query is possible in $\mathcal{O}(\Delta \log \rho)$, and if the transition times are equal for all edges, in $\mathcal{O}(\Delta)$.*

*Proof.* The result follows from the running times of the streaming algorithms [29] and Lemma 3.1. ∎

Theorem 3.1 is the basis for the running time improvement of the temporal closeness computation. Using the substream index, ranking all vertices according to their temporal closeness is possible with $n$ fastest path queries with a total running time in $\mathcal{O}(n\Delta)$.

As a trade-off, we need additional space for storing the substreams—for a temporal graph with $m$ edges and $n$ vertices, the space complexity of the substream index is in $\mathcal{O}(k \cdot m + n)$. Finally, it is noteworthy that a substream index can be used to output the distances as well as the corresponding paths, and can be used to speed up other temporal distance queries, e.g., earliest arrival or latest departure time queries.

### 3.1 Hardness of Finding a Minimal Index

Unfortunately, deciding if there exists a substream index with a given size is NP-complete.

THEOREM 3.2. *Given a temporal graph $\mathcal{G} = (V, \mathcal{E})$, and $k, B \in \mathbb{N}$. Deciding if there exists a substream index $\mathcal{I}$ with $k+1$ substreams and $size(\mathcal{I}) \leq B$ is NP-complete.*

*Proof.* We use a polynomial-time reduction from UNARYBINPACKING, which is the following NP-complete problem [8]:

**Given:** A set of $m$ items, each with positive size $w_i \in \mathbb{N}$ encoded in unary for $i \in [m]$, $k \in \mathbb{N}$, and $B \in \mathbb{N}$.

**Question:** Is there a partition $I_1, \ldots, I_k$ of $\{1, \ldots, m\}$ such that $\max_{1 \leq i \leq k}\{W_i\} \leq B$ where $W_i = \sum_{j \in I_i} w_j$?

Given a substream index $\mathcal{I}$ for a temporal graph and $B \in \mathbb{N}$, we can verify in polynomial time if $size(\mathcal{I}) \leq B$. Hence, the problem is in NP. We reduce UNARYBINPACKING to the problem of deciding if there exists a substream index of size less or equal to $B$. Given an instance of UNARYBINPACKING with $m$ items of sizes $w_i$ for $i \in [m]$, we construct in polynomial time a temporal graph $\mathcal{G} = (V, \mathcal{E})$ that consists of $m$ vertices. More specifically, for each $i \in [m]$, we construct a vertex $v_i$ that has $w_i$ self-loops, such that each edge $e \in \mathcal{E}$ has a unique availability time. We show that if UNARYBINPACKING has a *yes* answer, then $size(\mathcal{I}) \leq B$ and vice versa.

$\Rightarrow$: Let $I_1, \ldots, I_k$ be the partition of $[m]$ such that $\max_{1 \leq i \leq k}\{W_i\} \leq B$. For our *Substream* index, we use $k$ substreams $S_i = \bigcup_{j \in I_i} \xi(v_j)$ for $i \in [k]$. Because $|\xi(v_j)| = w_j$ for $j \in [m]$ it follows that the size of the substream index $size(\mathcal{I}) = \max_{1 \leq i \leq k}\{|S_i|\} = \max_{1 \leq i \leq k}\left\{\sum_{j \in I_i} w_j\right\} \leq B$.

$\Leftarrow$: Let $\mathcal{I} = (\mathcal{S}, f)$ with $\mathcal{S} = \{S_0, \ldots, S_k\}$ be the *Substream* index with $size(\mathcal{I}) = \max_{1 \leq i \leq k}\{|S_i|\} \leq B$. From the vertex mapping $f$, we construct the partition $I_1, \ldots, I_k$ of $[m]$ such that $\max_{1 \leq i \leq k}\{W_i\} \leq B$ holds for the UNARYBINPACKING instance. Let $I_i = \{j \in [m] \mid f(v_j) = i\}$ for $i \in [k]$. Finally, with $|\xi(v_j)| = w_j$, it follows that $\max_{1 \leq i \leq k}\{\sum_{j \in I_i} w_j\} \leq B$. ∎

**Algorithm 1:** Greedy index computation.

**Input:** Temporal edge stream $S$, $k \in \mathbb{N}$.
**Output:** Substream index $(\mathcal{S}, f)$

1 Initialize $S_j = \emptyset$ for $1 \leq j \leq k$, and $f(v) = 0$ for $v \in V$
2 **for** $i = 1$ *to* $n$ **do**
3  Compute $\xi(v_i)$
4  **if** $\xi(v_i) = \emptyset$ **then** $f(v_i) \leftarrow 0$
5  **else**
6  Let $S_j$ such that $|S_j \cup \xi(v_i)|$ is minimal
7  $S_j \leftarrow S_j \cup \xi(v_i)$
8  $f(v_i) \leftarrow j$
9 **return** $(\{S_1, \ldots, S_k\}, f)$

**3.2 A Greedy Approximation** We now introduce an efficient algorithm for computing a substream index with a bounded ratio between the size of the computed index and the optimal size. Algorithm 1 shows the simple greedy algorithm for computing the substream index for a temporal graph in edge stream representation $S$. After initialization, Algorithm 1 runs $n$ iterations of the for loop in line 2. The vertices are processed in an arbitrary order $v_1, \ldots, v_n$. In iteration $i$, the algorithm first computes the edge stream $\xi(v_i)$ using a single pass over the edge stream. After computing $\xi(v_i)$, it is added to one of the substreams, and the mapping $f(v_i) = j$ is updated. In each round, $\xi(v_i)$ is added to one of the substreams $S_j \in \{S_1, \ldots, S_k\}$, such that the increase of the size of $S_j$ is minimal (line 6.). All vertices $v \in V$ for which $\xi(v)$ is empty are assigned to the empty substream $S_0$ (line 4).

We use the following bounds to show the approximation ratio of the sizes of an optimal index and one constructed by Algorithm 1.

LEMMA 3.2. *Let $\mathcal{G}$ be a temporal graph with $n^+$ non-sink vertices, $k \in \mathbb{N}$, and let $B \subseteq \{\xi(v) \mid v \in V\}$ such that $|B| = \left\lceil \frac{n^+}{k} \right\rceil$ and the union $\left|\bigcup_{\xi \in B} \xi\right|$ is maximal.*

1. *The size of an optimal index is $size(\text{OPT}) \geq \frac{m}{k}$,*
2. *and for greedy, $size(\text{GREEDY}) \leq \left|\bigcup_{\xi \in B} \xi\right|$.*

*Proof.* (1) Each edge $e \in \mathcal{E}$ has to be in at least one substream. The size of the substream index is minimized if the edges are distributed equally to all substreams—in this case, reassigning any edge from one of the $k$ substreams to another would lead to an increase of the size of the index. Hence, $size(\text{OPT}) \geq \frac{m}{k}$. (2) Assume in some iteration $i$ the edge stream $\xi(v_i)$ is added to a substream $S_j$ such that after the addition $|S_j| > |\cup_{\xi \in B} \xi|$. Algorithm 1 chooses $S_j$ such that adding $\xi(v_i)$ to any other substream does not lead to a smaller value. Hence, for all $S_\ell$ with $1 \leq \ell \leq k$ it is $|S_\ell \cup \xi(v_i)| > |\cup_{\xi \in B} \xi|$. However, this leads to a contradiction to the maximal size of the sum over all substreams $\sum_{1 \leq \ell \leq k} |S_\ell \cup \xi(v_i)| \leq k \cdot |\cup_{\xi \in B} \xi|$. ∎

THEOREM 3.3. *The ratio between the size of the greedy solution and the optimal solution for any valid input $S$ and $k$ is bounded by $\frac{size(\text{GREEDY})}{size(\text{OPT})} \leq \frac{k}{\delta}$, with $\delta = \frac{m}{|\cup_{\xi \in B} \xi|}$ and $1 \leq \delta$.*

*Proof.* With the two bounds from Lemma 3.2, follows

$$\frac{size(\text{GREEDY})}{size(\text{OPT})} \leq \frac{|\bigcup_{\xi \in B} \xi|}{\frac{m}{k}} = \frac{k \cdot |\bigcup_{\xi \in B} \xi|}{m} = \frac{k}{\delta}$$

And, because $\cup_{\xi \in B} \xi \subseteq \mathcal{E}$ it follows $1 \leq \delta$. ∎

We now discuss the running time of Algorithm 1.

THEOREM 3.4. *Given a temporal graph $S$ in edge stream representation, and $k \in \mathbb{N}$, the running time of Algorithm 1 is in $\mathcal{O}(mnk)$.*

*Proof.* Initialization is done in $\mathcal{O}(n)$. The running time for computing all $\xi(v)$ of Algorithm 1 is in $\mathcal{O}(nm)$. For the assignment of the edge streams $\xi(v)$ to the substreams $S_1, \ldots, S_k$, the algorithm needs to compute the union $\xi(v_i) \cup S_j$ for each $j \in \{1, \ldots, k\}$. The sizes of $S_j$ and $\xi(v_i)$ are bounded by the number of edges $m$, therefore, union is possible in $\mathcal{O}(m)$ time. The algorithm needs $n \cdot (k+1)$ union operations, leading to a total running time of $\mathcal{O}(nmk)$. ∎

**3.3 Improving the Greedy Algorithm** We improve the greedy algorithm presented in Section 3.2 in three ways: 1) During the construction and queries, we skip edges that are too early in the temporal edge stream and do not need to be considered for answering a given query. 2) We use bottom-$h$[1] sketches to avoid the costly union operations that we need to find the right substream to which we assign an edge stream. 3) We use parallelization and a batch-wise computation scheme to benefit from modern parallel processing capabilities.

Note that using only improvements 1) and 3), we would obtain a parallel greedy algorithm with the same approximation ratio as Algorithm 1. However, improvement 2) leads to the loss of the approximation guarantee. In our experimental evaluation in Section 4, we will see that (i) the improved algorithm usually leads to indices that are not larger than the ones computed with Algorithm 1, and (ii) the query times of the indices constructed with the improved algorithm are also faster for all data sets. In the following, we describe the improvements in detail.

[1] Usually called *bottom-k* sketch. We use $h$ instead of $k$ because $k$ denotes the number of substreams.

**3.3.1 Edge Skipping** The idea of edge skipping is to ignore all edges that have timestamps earlier than the availability time of the first edge leaving the query vertex $v$. Let $S$ be a temporal edge stream with edges $e_1, \ldots, e_m$. By definition, the edges are sorted in non-decreasing order of their availability times. The position of the first outgoing edge from vertex $v$ might be at a late position in the edge stream $S$. For example, the first outgoing edge $e_p$ at $v$ could be at a position $p > m/2$. Therefore, if we know the position $p$ of the first edge, we can start the streaming algorithm at position $p$ and skip more than half of the edges in the run of the streaming algorithm. To exploit this idea, we store for each vertex $v \in V$ the first position $p$ in the edge stream $S$ of the first edge $e_p = (v, w, t, \lambda)$ that starts at vertex $v$. To compute the first positions, we first initialize an array of length $n$ in which we store the first positions of the earliest outgoing edges for each $v \in V$. We use a single pass over the edge stream to find these positions. Hence, the array can be computed in $\mathcal{O}(n + m)$ running time, and it has a space complexity in $\mathcal{O}(n)$. We use the edge skipping in two ways. First, it is used to speed up the computation of the edges streams $\xi(v)$ during the index construction. Secondly, we compute an array of starting positions for edge skipping for each of the final substreams in $\mathcal{S}$ to speed up the query times.

**3.3.2 Bottom-h Sketches** The main drawback of Algorithm 1 is that it has to compute the union of $S_j \cup \xi(v_i)$ for all substreams $S_j$ for $1 \leq j \leq k$ in order to determine the substream to which the edge stream $\xi(v_i)$ should be added. To avoid these expensive union computations, we reduce the sizes of $\xi(v)$ for $v \in V$ by using sketches of the edge streams, and estimate the *Jaccard distance* between the sketches of the edge streams and substreams. For two sets $A$ and $B$, the Jaccard distance is defined as $J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$. The Jaccard distance between two sets can be estimated using *min-wise* hashing [3]. The idea is to generate randomized sketches of sets that are too large to handle directly. After computing the sketches, further operations are done in the *sketch space*. This way, it is possible to construct unions of sketches and estimate the Jaccard similarity between pairs of the original sets efficiently.

More specifically, let $A$ be a set of integers. A bottom-$h$ sketch $s(A)$ is generated by applying a permutation $\pi$ to the set $A$ and choosing $h$ smallest elements of the set $\{\pi(a) \mid a \in A\}$ ordered in non-decreasing value[2]. For two sets $A$ and $B$, we can obtain $s(A \cup B)$ by choosing $h$ smallest elements from $s(A)$ and $s(B)$. This way, we obtain a sample of the union $A \cup B$ of size $h$. Now, the subset $s(A \cup B) \cap s(A) \cap s(B)$ contains only the elements that are in the intersection of $A$, $B$, and the union sketch $s(A \cup B)$. We use the following result.

LEMMA 3.3. ([3]) *The value*

$$\hat{J}(A, B) = 1 - \frac{|s(A \cup B) \cap s(a) \cap s(b)|}{|s(A \cup B)|}$$

*is an unbiased estimator for the Jaccard distance.*

Using the estimated Jaccard distance between an edge stream $\xi(v)$ and a substream $S_j$, we decide if we should add $\xi(v)$ to $S_j$. If the estimated Jaccard distance is low, then we expect that adding $\xi(v)$ to $S_j$ does not lead to a significant increase in the size of $S_j$.

We now describe how we compute and use the sketches of the edge streams. During the computation of $\xi(v_i)$, the algorithm iterates over the input stream $S$, starting from position $p$ determined by edge skipping array, and processes the edges $e_p, \ldots, e_\ell, \ldots, e_m$ in chronological order. Let $e_\ell = (u, v, t, \lambda)$ be an edge that can be traversed, i.e., the arrival time at $u$ is smaller or equal to $t$. We compute a bottom-$h$ sketch using the hashed position $\pi(\ell)$ of edge $e_\ell$ in the input stream. Therefore, we compute a hash value for all edges that can be traversed, and we keep the $h$ smallest hashed values $\pi_1, \ldots, \pi_h$ as our sketch $s(\xi(v_i)) = (\pi_1, \ldots, \pi_h)$, where the hash function $\pi$ is a permutation of $[m]$. Note that the position $\ell$ of edge $e_\ell$ in the edge stream $S$ is a unique identifier of $e_\ell$. Furthermore, each edge $e_\ell = (u, v, t, \lambda)$ represents a substream of $S$ consisting of all edges in $e = (x, y, t_e, \lambda_e) \in \xi(v)$ with availability time $t_e \geq t + \lambda$, i.e., the corresponding subgraph that is reachable after traversing $e$.

In the assignment phase (line 7), Algorithm 2 proceeds similarly to Algorithm 1 in a greedy fashion. However, we adapt the assignment objective such that it leads to improved substreams in terms of size and query times. To this end, we consider the number of vertices $I_j$ assigned to substream $S_j$.

DEFINITION 4. *Let $v \in V(S)$, and $I_j = |\{v \in V(S) \mid f(v) = S_j\}|$ the number of to $S_j$ assigned vertices. We define the* ranking function $r : V(S) \times [k] \to [1, n]$ *as*

$$r(v, j) = \tfrac{1}{2}(I_j + 1) \cdot (\hat{J}(s(S_j), s(\xi(v)) + 1).$$

Using the ranking function, Algorithm 2 decides to add the edge stream $\xi(v)$ to the substream $S_j$ if $r(v, j)$ is minimal for $1 \leq j \leq k$ (line 10). By additionally considering the number $I_j$ of vertices assigned to substream $S_j$, we optimize for small substreams $S_i$ and a vertex assignment such that vertices are assigned to smaller

[2]We assume that $|A| \geq h$, otherwise we choose only $|A|$ elements.

substreams $S_i$ rather than to larger ones. Note that if a vertex $u$ is assigned to a small substream, queries starting at $u$ can be answered fast. If the ranking function $r(v, j)$ is close to one, not many vertices are assigned to $j$, or the estimated Jaccard distance between $\xi(v)$ and $S_j$ is small. On the other hand, if $r(v, j)$ is closer to $n$, the number of to $S_j$ assigned vertices is high, and/or the estimated Jaccard distance is high. The intuition is that, even if we have a substream that contains a majority of edges, we want to assign the remaining vertices to substreams with a smaller size if possible.

**3.3.3 Parallelization** Algorithm 2 shows our improved *parallel* greedy algorithm that has as input the temporal graph $S$, the number of substreams $k$, the hash-size $h$, and a batch-size $B > 0$. After the initialization and the computation of the edge skipping array, it processes the input graph in batches of size $B$ to allow a parallel computation of the edge stream assignment. The batch size determines how many vertices are processed in each iteration of the outer while-loop (line 4). For each batch of vertices, Algorithm 2 runs three phases of computation. In the first phase (line 5), Algorithm 2 first computes the edge streams $\xi(v_i)$ for all vertices $v_i$ that part of the current batch. The edge skipping array is used to find the first position $1 \leq p \leq m$ of $v_i$ in $S$. The second phase computes an assignment of the edge streams to the substreams using the bottom-$h$ sketches. To this end, we keep the sketches $s(S_j)$ of the substreams stored as $C_j$ for each $j \in [k]$. After finding the right substream, $C_j$, $I_j$ and $f(v_i)$ are updated accordingly. The third phase (line 14) constructs the substreams in parallel using the determined assignment of edge streams. Finally, after all batches are processed, edge skipping arrays for each $S_i$ are computed in parallel (line 19).

THEOREM 3.5. *Given a temporal graph in edge stream representation with $m$ edges and $n \leq m$ vertices, and $B, h, k \in \mathbb{N}$ with $h \geq 1$, $k \geq 2$, and $h \cdot k \leq m$. Then, the running time of Algorithm 2 is in $\mathcal{O}(\frac{knm}{P})$ on a parallel machine[3] with $P$ processors, for $P \leq k$ and $P \leq m$.*

*Proof.* Initialization is done in $\mathcal{O}(n)$. Computing the initial *Time Skip* index for the input $S$ takes $\mathcal{O}(m)$ time. The algorithm iterates over $\lceil n/B \rceil$ batches. In one iteration of the while loop, the running time for the parallel computation of the edge sets $\mathcal{E}(v_i)$ is in $\mathcal{O}(B \cdot (n + m)/P)$. For the bottom-$h$ sketch, we use a sorted list to keep the smallest $h$ hash values of the edges. Updating the list is done in $\log h$. Finding the indices $i$ for the substreams $S_i$ in line 10 takes $\mathcal{O}(B \cdot ((kh/P) + \log P))$ time. Therefore, the total running time of the first two phases is $\lceil n/B \rceil \cdot B \cdot (\mathcal{O}(m/P) + \mathcal{O}(kh/P + \log P)) = \mathcal{O}(\frac{nm}{P} + n \log P)$. The total running time of the update phase is $\lceil n/B \rceil \cdot (\mathcal{O}(\frac{k}{P} \cdot Bm) = \mathcal{O}(\frac{k}{P} nm)$. Computing the edge skipping arrays for $S_i$ with $i \in [k]$ in parallel takes $\mathcal{O}(\frac{k}{P} \cdot m)$ time. ∎

[3] We consider the *Concurrent Read Exclusive Write* (CREW) PRAM model.

**Algorithm 2:** Parallel index computation.

**Input:** Temporal edge stream $S$, $k, B \in \mathbb{N}$.
**Output:** Substream index $(\mathcal{S}, f)$

1 Initialize in **parallel** $I_i = 0$, $S_i = \emptyset$, $C_i = \emptyset$ for $i \in [k]$, and $f(v) = 0$ for $v \in V_S$
2 $start \leftarrow 1$, $end \leftarrow B$
3 Compute edge skipping array for $S$
4 **while** $start < n$ **do**
  `/* Phase 1: compute streams & sketches */`
5  **parallel for** $i = start, \ldots, end$ **do**
6  compute $\xi(v_i)$ and $s(\xi(v_i))$ using initial edge skipping array
  `/* Phase 2: compute stream assignments */`
7  **for** $i = start, \ldots, end$ **do**
8  **if** $\xi(v_i) = \emptyset$ **then** $f(v_i) \leftarrow 0$
9  **else**
10  in **parallel** find $j \in [k]$ such that $r(v_i, j)$ is minimal
11  $C_j \leftarrow s(C_j \cup s(\xi(v_i)))$
12  $f(v_i) \leftarrow j$
13  $I_j \leftarrow I_j + 1$
  `/* Phase 3: updating substreams */`
14  **parallel for** $j = 1, \ldots, k$ **do**
15  **for** $v$ *with* $f(v) = j$ **do**
16  $S_j \leftarrow S_j \cup \xi(v)$
17  $start \leftarrow start + B$
18  $end \leftarrow \min(end + B, n)$
19 Compute in **parallel** edge skipping array for $S_i$, $i \in [k]$
20 **return** $(\{S_1, \ldots, S_k\}, f)$

## 4 Experimental Results

We implemented our algorithms in C++ using GNU CC Compiler 9.3.0. with the flag `--O3`, and we used OpenMP v4.5. The source code is available at `https://gitlab.com/tgpublic/tgindex`. The experiments ran on a computer cluster, where each experiment had an exclusive node with an Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz and 192 GB of RAM. The time limit for each experiment was set to 48 hours.

**Algorithms:** We use the following algorithms.

- GREEDY is the implementation of Algorithm 1.
- SUBSTREAM is the implementation Algorithm 2.

Table 1: Statistics of the data sets.

| Data set | Properties | | | | |
|---|---|---|---|---|---|
| | $\|V\|$ | $\|\mathcal{E}\|$ | $\|\mathcal{T}(\mathcal{G})\|$ | avg. $\|\xi(v)\|$ | max $\|\xi(v)\|$ |
| *Infectious* | 10 972 | 415 912 | 76 943 | 1 100.1 | 9 339 |
| *AskUbuntu* | 159 316 | 964 437 | 960 866 | 3 050.8 | 117 930 |
| *Prosper* | 89 269 | 3 394 978 | 1 259 | 14 979.4 | 205 461 |
| *Arxiv* | 28 093 | 4 596 803 | 2 337 | 260 471.5 | 3 860 987 |
| *Youtube* | 3 223 585 | 9 375 374 | 203 | 136 682.2 | 4 928 847 |
| *StackOverflow* | 2 464 606 | 17 823 525 | 16 926 738 | 851 232.8 | 11 982 619.0 |

- TopChain is the state-of-the-art index for single-source-single-destination (SSSD) temporal reachability queries [30]. We set the parameter $k = 5$ as suggested in [30].
- OnePassFP is temporal closeness algorithm based on the state-of-the-art SSAD edge stream algorithm for minimum duration distances [29].
- Top-$\ell$ is the state-of-the-art temporal closeness algorithm [18]. It computes the topmost $\ell$ closeness values and vertices exactly. We set $\ell = 100$.

The C++ source codes of TopChain, OnePassFp, and Top-$\ell$ were provided by the corresponding authors and compiled using the same settings as our algorithms.

**Data sets:** We used the following real-world temporal graphs. (1) *Infectious:* a face-to-face human contact network [11]. (2) *AskUbuntu:* Interactions on the website *Ask Ubuntu* [20]. (3) *Prosper:* A temporal network based on a personal loan website [23]. (4) *Arxiv:* An author collaboration graph from the *arXiv* website [13]. (5) *Youtube:* A social network on the video platform *Youtube* [15]. (6) *StackOverflow:* Interactions on the website *StackOverflow* [20]. Table 1 shows statistics of the data sets.

### 4.1 Indexing Time and Index Size

For SubStream, we set the number of substreams to $k = 2^i$ for $i \in \{5, \ldots, 8\}$. We choose a sketch size of $h = 8$ because in our experiments if showed a good trade-off between index construction times, query times, and resulting index sizes. The construction time increases for larger values of $h$, however the gain in construction and query times diminished. We set the batch size $B$ to $n$ for data sets with less than one million vertices and 2048 otherwise. Furthermore, we used 32 threads. We report the indexing times in Table 2a and the index sizes in Table 2b. For SubStream, we run the indexing ten times and report the averages and standard deviations.

**Indexing time:** As expected, Greedy has high running times. It has up to several orders of magnitude higher running times than the other indices, and for the two largest data sets, *Youtube* and *StackOverflow* the computations could not be finished in the given time limit of 48 hours. SubStream improves the indexing time of Greedy immensely for all data sets. However,

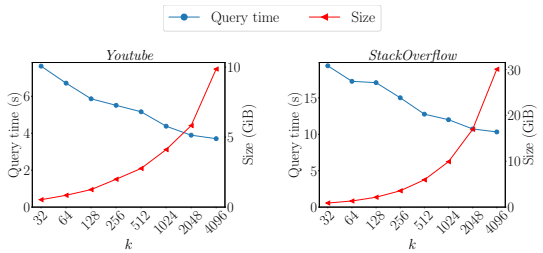



Figure 2: The trade-off between index size and query time for increasing $k$ for *Youtube* and *StackOverflow*.

SubStream has higher indexing times than TopChain for all data sets. The indexing time of TopChain is linear in the graph size, and the indexing for SubStream computes for each vertex $v \in V$ all reachable edges $\xi(v)$, hence higher running times and weaker scalability of SubStream are expected. However, the query times using TopChain for SSAD queries cannot compete with our indices and are, in most cases, orders of magnitude higher. The reason is that TopChain is designed for SSSD queries.

**Index size:** Table 2b shows that the index sizes of Greedy are only smaller than the ones of SubStream for the *Infectious* data set. In all other cases, the SubStream size are (substantially) smaller. Compared to TopChain, our SubStream can lead to larger sizes depending on $k$. However, for *Infectious* and *AskUbuntu* the sizes of SubStream are smaller sizes for all $k$. In general larger values of $k$ lead to larger indices, and shorter query times. Hence, SubStream provides a typical trade-off between index size and query time. This is also demonstrated in Figure 2, which shows the trade-off for the two largest data sets, *Youtube* and *StackOverflow*, for increasing number of substreams $k$.

### 4.2 Temporal Closeness Computation

We set $k \in \{256, 2048\}$ and $h = 8$ for SubStream. Table 3 shows the running times. For the SubStream index, the running times **include the index construction times**. Our indices improve the running times for all data sets. Large improvements are gained for the *Infectious* and *Prosper* data sets compared to OnePassFp with speed-ups of eight and 15, respectively. The speed-up for the other data sets is at least 2.2 compared to OnePassFp. OnePassFp could not compute the ranking for *StackOverflow* in the given time limit of seven days. Compared to the Top-100 algorithm, the speed-up is between 1.2 (*StackOverflow*) and 2.3 (*Prosper*) with an average speed-up of 1.5. Note that in contrast to Top-100, SubStream *computes the complete ranking of all vertices.* As expected, the running time with $k = 2048$ is shorter compared to $k = 256$, even though the indexing times for $k = 2048$ are slightly higher.

Table 2: Indexing times and sizes. We report the mean and standard deviation over ten runs for SubStream.

(a) Indexing times in (s). OOT—Out of time.

| | Greedy | | | | SubStream | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Data set** | $k=32$ | $k=64$ | $k=128$ | $k=256$ | $k=32$ | $k=64$ | $k=128$ | $k=256$ | TopChain |
| *Infectious* | 4.4 | 3.0 | 2.4 | 2.6 | 0.75±000.0 | 0.78±000.0 | 0.79±000.0 | 0.78±000.0 | 0.43 |
| *AskUbuntu* | 201.9 | 216.7 | 211.7 | 229.3 | 6.43±000.2 | 6.33±000.2 | 6.09±000.2 | 6.10±000.2 | 0.53 |
| *Prosper* | 440.2 | 427.8 | 427.2 | 486.8 | 18.44±000.7 | 17.30±000.4 | 16.70±000.3 | 16.58±000.3 | 1.98 |
| *Arxiv* | 2824.2 | 2895.9 | 3214.2 | 4223.6 | 28.29±000.8 | 25.52±001.1 | 22.56±001.1 | 21.38±002.9 | 0.78 |
| *Youtube* | OOT | OOT | OOT | OOT | 5376.90±573.1 | 4813.78±088.4 | 4572.23±068.6 | 4360.68±038.5 | 12.46 |
| *StackOverflow* | OOT | OOT | OOT | OOT | 12467.22±105.7 | 11599.58±235.1 | 11307.12±234.0 | 11051.26±184.7 | 45.95 |

(b) Index sizes in MiB. ("–" indicates that the index is not available due to time out during construction).

| | Greedy | | | | SubStream | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **Data set** | $k=32$ | $k=64$ | $k=128$ | $k=256$ | $k=32$ | $k=64$ | $k=128$ | $k=256$ | TopChain |
| *Infectious* | 2.6 | 2.9 | 3.9 | 5.5 | 7.79±000.4 | 5.92±000.3 | 5.05±000.2 | 4.88±000.1 | 28.49 |
| *AskUbuntu* | 15.2 | 29.1 | 56.3 | 106.9 | 9.36±001.7 | 10.41±001.7 | 9.87±001.3 | 12.67±000.8 | 20.41 |
| *Prosper* | 34.3 | 54.8 | 94.1 | 169.7 | 33.98±001.4 | 47.34±002.3 | 66.20±005.7 | 79.27±007.9 | 65.42 |
| *Arxiv* | 457.2 | 883.8 | 1674.3 | 3123.6 | 295.98±028.1 | 459.46±046.5 | 572.59±062.2 | 721.02±041.4 | 18.86 |
| *Youtube* | – | – | – | – | 541.53±037.5 | 863.72±100.0 | 1280.67±147.9 | 2038.11±296.5 | 351.03 |
| *StackOverflow* | – | – | – | – | 907.51±82.2 | 1366.65±111.7 | 2201.15±217.0 | 3644.07±375.5 | 1501.00 |

Table 3: Running times for the temporal closeness in seconds and hours. OOT—Out of time after 7 days.

| | SubStream | | Baselines | |
|---|---|---|---|---|
| **Data set** | $k=256$ | $k=2048$ | OnePassFp | Top-100 |
| *Infectious* | 1.67 s | **1.51** s | 12.06 s | 2.25 s |
| *AskUbuntu* | 102.95 s | **102.46** s | 229.73 s | 132.53 s |
| *Prosper* | 130.63 s | **109.33** s | 1 665.20 s | 260.87 s |
| *Arxiv* | 314.73 s | **286.86** s | 630.60 s | 398.50 s |
| *Youtube* | 63.82 h | **59.72** h | 145.98 h | 81.21 h |
| *StackOverflow* | 88.00 h | **86.49** h | OOT | 107.66 h |

The high speed-ups in the case of *Infectious* and *Prosper* can be explained by the small average and maximal sizes of $\xi(v)$ for both data sets (see Table 1), leading to an on average small maximum number of edges in the substreams of only 2.5% and 5.8% of the total number of edges. In conclusion, our SubStream index significantly improves the running times compared to the state-of-the-art algorithms for temporal closeness rankings, while computing the ranking of all vertices.

## 5 Related Work

Recent and comprehensive introductions to temporal graphs are provided in, e.g., [9, 28]. Wu et al. [29] introduce streaming algorithms for finding the fastest, shortest, latest departure, and earliest arrival paths. In [25] and [26], the authors compare temporal distance metrics and temporal centrality measures to their static counterparts. They reveal that the temporal versions for analyzing temporal graphs have advantages over static approaches on the aggregated graphs. Variants of temporal closeness have been introduced in [19, 14, 24]. Our work uses the harmonic temporal closeness definition from [18]. As far as we know, our work is the first one examining indices for *SSAD temporal distance queries* and its application for temporal closeness computation. Yu and Cheng [31] give an overview of indexing techniques for reachability and distances in static graphs. There are several works on SSSD time-dependent routing in transportation networks, e.g., [1, 6]. Wang et al. [27] propose *Timetable Labeling (TTL)*, a labeling-based index for SSSD reachability queries based on hub labelings for temporal graphs. In [30], the authors introduce an index for SSSD reachability queries in temporal graphs called TopChain. The index uses a static representation of the temporal graph as a directed acyclic graph (DAG). On the DAG, a chain cover with labels at the vertices is computed. The labeling can be used to determine the reachability between vertices. TopChain is faster than TTL and has shorter query times (see [30]). As far as we know, our work is the first one discussing indices for SSAD temporal distance queries.

## 6 Conclusion and Future Work

We introduced the *Substream* index for speeding up temporal closeness computation. Our index speeds up the vertex-ranking according to the temporal closeness up to one order of magnitude. It can be extended to support efficiently dynamic updates for edge insertions or deletions. In future work, we want to further improve the indexing time using a distributed algorithm, and to explore further applications for our new index.

**Acknowledgements** This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy–EXC-2047/1–390685813. This research has been funded by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia as part of the Lamarr-Institute for Machine Learning and Artificial Intelligence, LAMARR22B.

## References


[1] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-dependent contraction hierarchies. In *ALENEX*, pages 97–105. SIAM, 2009.

[2] Dan Braha and Yaneer Bar-Yam. *Time-Dependent Complex Networks: Dynamic Centrality, Dynamic Motifs, and Cycles of Social Interactions*, pages 39–50. Springer, Berlin, Heidelberg, 2009.

[3] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of SEQUENCES*, pages 21–29. IEEE, 1997.

[4] Julián Candia, Marta C González, Pu Wang, Timothy Schoenharl, Greg Madey, and Albert-László Barabási. Uncovering individual and collective human dynamics from mobile phone records. *Journal of physics A: mathematical and theoretical*, 41(22):224015, 2008.

[5] Martino Ciaperoni, Edoardo Galimberti, Francesco Bonchi, Ciro Cattuto, Francesco Gullo, and Alain Barrat. Relevance of temporal cores for epidemic spread in temporal networks. *Scientific reports*, 10(1):1–15, 2020.

[6] Daniel Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.

[7] Jean-Pierre Eckmann, Elisha Moses, and Danilo Sergi. Entropy of dialogues creates coherent structures in e-mail traffic. *National Academy of Sciences*, 101(40):14333–14337, 2004.

[8] David S. Garey, Michael R.and Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[9] Petter Holme. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):234, 2015.

[10] Petter Holme, Christofer R Edling, and Fredrik Liljeros. Structure and time evolution of an internet dating community. *Social Networks*, 26(2):155–174, 2004.

[11] Lorenzo Isella, Juliette Stehlé, Alain Barrat, Ciro Cattuto, Jean-François Pinton, and Wouter Van den Broeck. What's in a crowd? Analysis of face-to-face behavioral networks. *Journal of Theoretical Biology*, 271(1):166–180, 2011.

[12] Sophie Lebre, Jennifer Becq, Frederic Devaux, Michael PH Stumpf, and Gaelle Lelandais. Statistical inference of the time-varying structure of gene-regulation networks. *BMC syst biol*, 4(1):1–16, 2010.

[13] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1):2–es, 2007.

[14] Ciémence Magnien and Fabien Tarissan. Time evolution of the importance of nodes in dynamic networks. In *ASONAM*, pages 1200–1207. IEEE, 2015.

[15] Alan Mislove, Massimiliano Marcon, P. Krishna Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42. ACM, 2007.

[16] Antoine Moinet, Michele Starnini, and Romualdo Pastor-Satorras. Burstiness and aging in social temporal networks. *Physical review*, 114(10):108701, 2015.

[17] Lutz Oettershagen, Nils M Kriege, Christopher Morris, and Petra Mutzel. Classifying dissemination processes in temporal graphs. *Big Data*, 8(5):363–378, 2020.

[18] Lutz Oettershagen and Petra Mutzel. Efficient top-k temporal closeness calculation in temporal networks. In *ICDM*, pages 402–411. IEEE, 2020.

[19] Raj Kumar Pan and Jari Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, 2011.

[20] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proc of the Tenth ACM Intl Conf on Web Search and Data Mining*, pages 601–610, 2017.

[21] Teresa M Przytycka, Mona Singh, and Donna K Slonim. Toward the dynamic interactome: it's about time. *Briefings in bioinformatics*, 11(1):15–29, 2010.

[22] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *JEA*, 12:1–39, 2008.

[23] Ursula Redmond and Pádraig Cunningham. A temporal network analysis reveals the unprofitability of arbitrage in the prosper marketplace. *Expert Systems with Applications*, 40(9):3715–3721, 2013.

[24] Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frederic Amblard. Time-varying graphs and social network analysis: Temporal indicators and metrics. *arXiv preprint arXiv:1102.0629*, 2011.

[25] John Tang, Ilias Leontiadis, Salvatore Scellato, Vincenzo Nicosia, Cecilia Mascolo, Mirco Musolesi, and Vito Latora. *Applications of Temporal Graph Metrics to Real-World Networks*, pages 135–159. Springer, Berlin, Heidelberg, 2013.

[26] John Kit Tang, Mirco Musolesi, Cecilia Mascolo, Vito Latora, and Vincenzo Nicosia. Analysing information flows and key mediators through temporal centrality metrics. In *Soc Netw Sys*, page 3. ACM, 2010.

[27] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. Efficient route planning on public transportation networks: A labelling approach. In *SIGMOD*, pages 967–982, 2015.

[28] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Sci. and Engin.*, 4(4):352–366, 2019.

[29] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proc VLDB Endowment*, 7(9):721–732, 2014.

[30] Huanhuan Wu, Yuzhen Huang, James Cheng, Jinfeng Li, and Yiping Ke. Reachability and time-based path queries in temporal graphs. In *ICDE*, pages 145–156. IEEE, 2016.

[31] Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, volume 40, pages 181–215. Springer, 2010.

# Motifs in Temporal Networks

Ashwin Paranjape* Stanford University ashwinpp@stanford.edu

Austin R. Benson* Stanford University arbenson@stanford.edu

Jure Leskovec Stanford University jure@cs.stanford.edu

## ABSTRACT

Networks are a fundamental tool for modeling complex systems in a variety of domains including social and communication networks as well as biology and neuroscience. Small subgraph patterns in networks, called network motifs, are crucial to understanding the structure and function of these systems. However, the role of network motifs in temporal networks, which contain many timestamped links between the nodes, is not yet well understood.

Here we develop a notion of a temporal network motif as an elementary unit of temporal networks and provide a general methodology for counting such motifs. We define temporal network motifs as induced subgraphs on sequences of temporal edges, design fast algorithms for counting temporal motifs, and prove their runtime complexity. Our fast algorithms achieve up to 56.5x speedup compared to a baseline method. Furthermore, we use our algorithms to count temporal motifs in a variety of networks. Results show that networks from different domains have significantly different motif counts, whereas networks from the same domain tend to have similar motif counts. We also find that different motifs occur at different time scales, which provides further insights into structure and function of temporal networks.

## 1. INTRODUCTION

Networks provide an abstraction for studying complex systems in a broad set of disciplines, ranging from social and communication networks to molecular biology and neuroscience [20]. Typically, these systems are modeled as static graphs that describe relationships between objects (nodes) and links between the objects (edges). However, many systems are not static as the links between objects dynamically change over time [8]. Such *temporal networks* can be represented by a series of timestamped edges, or *temporal edges*. For example, a network of email or instant message communication can be represented as a sequence of timestamped directed edges, one for every message that is sent from one person to another. Similar representations can be used to model computer networks, phone calls, financial transactions, and biological signaling networks.

*These authors contributed equally to this work.



*WSDM '17, February 6–10, 2017, Cambridge, United Kingdom.*



DOI: http://dx.doi.org/10.1145/3018661.3018731

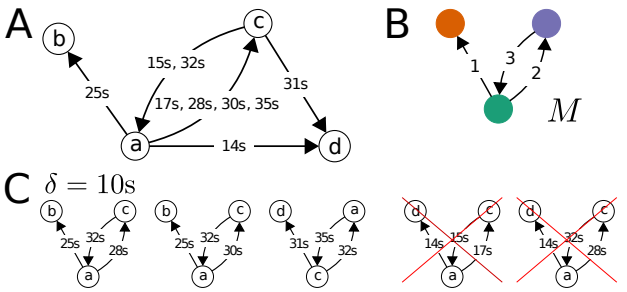



**Figure 1: Temporal graphs and $\delta$-temporal motifs. A: A temporal graph with nine temporal edges. Each edge has a timestamp (listed here in seconds). B: Example 3-node, 3-edge $\delta$-temporal motif $M$. The edge labels correspond to the ordering of the edges. C: Instances of the $\delta$-temporal motif $M$ in the graph for $\delta$ = 10 seconds. The crossed-out patterns are not instances of $M$ because either the edge sequence is out of order or the edges do not all occur within the time window $\delta$.**

While such temporal networks are ubiquitous, there are few tools for modeling and characterizing the underlying structure of such dynamical systems. Existing methods either model the networks as strictly growing where a pair of nodes connect once and stay connected forever [2, 10, 17] or aggregate temporal information into a sequence of snapshots [1, 6, 23]. These techniques fail to fully capture the richness of the temporal information in the data.

Characterizing temporal networks also brings a number of interesting challenges that distinguish it from the analysis of static networks. For example, while the number of nodes and pairs of connected nodes can be of manageable size, the number of temporal edges may be very large and thus efficient algorithms are needed when analyzing such data. Another interesting challenge is that patterns in temporal networks can occur at different time scales. For example, in telephone call networks, reciprocation (that is, a person returning a call) can occur on very short time intervals, while more intricate patterns (e.g., person $A$ calling person $B$, who then calls $C$) may occur at larger time scales. Lastly, there are many possible temporal patterns as the order as well as the sequence of edges play an important role.

**Present work: Temporal network motifs.** Here, we provide a general methodology for analyzing temporal networks. We define temporal networks as a set of nodes and a collection of directed temporal edges, where each edge has a timestamp. For example, Fig. 1A illustrates a small temporal network with nine temporal edges between five ordered pairs of nodes.

Our analytical approach is based on generalizing the notion of network motifs to temporal networks. In static networks, network motifs or graphlets are defined as small induced subgraphs occurring in a bigger network structure [4, 19, 29]. We extend static mo-

tifs to temporal networks and define $\delta$-temporal motifs, where all the edges in a given motif $M$ have to occur inside the time period of $\delta$ time units. These $\delta$-temporal motifs simultaneously account for ordering of edges and a temporal window in which edges can occur. For example, Fig. 1B shows a motif on three nodes and three edges, where the edge label denotes the order in which the edges appear. While we focus on directed edges with a single timestamp in this work, our methodology seamlessly generalizes to common variations on this model. For example, our methods can incorporate timestamps with durations (common in telephone call networks), colored edges that identify different types of connections, and temporal networks with undirected edges.

We then consider the problem of counting how many times does each $\delta$-temporal motif occur in a given temporal network. We develop a general algorithm for counting temporal network motifs defined by any number of nodes and edges that avoids enumeration over subsets of temporal edges and whose complexity depends on the structure of the static graph induced by the temporal motif. For motifs defined by a constant number of temporal edges between 2 nodes, this general algorithm is optimal up to constant factors—it runs in $O(m)$ time, where $m$ is the number of temporal edges.

Furthermore, we design fast variations of the algorithm that allow for counting certain classes of $\delta$-temporal motifs including star and triangle patterns. These algorithms are based on a common framework for managing summary counts in specified time windows. For star motifs with 3 nodes and 3 temporal edges, we again achieve a running time linear in the input, i.e., $O(m)$ time. Given a temporal graph with $\tau$ induced triangles in its induced static graph, our fast algorithm counts temporal triangle motifs with 3 temporal edges in $O(\tau^{1/2}m)$ worst-case time. In contrast, any algorithm that processes triangles individually takes $O(\tau m)$ worst-case time. In practice, our fast temporal triangle counting algorithm is up to 56 times faster than a competitive baseline and runs in just a couple of hours on a network with over two billion temporal edges.

Our algorithmic framework enables us to study the structure of several complex systems. For example, we explore the differences in human communication patterns by analyzing motif frequencies in text message, Facebook wall post, email and private online message network datasets. Temporal network motif counts reveal that text messaging and Facebook wall posting are dominated by "blocking" communication, where a user only engages with one other user at a time, whereas email is mostly characterized by "non-blocking" communication as individuals send out several emails in a row. Furthermore, private online messaging contains a mixture of blocking and non-blocking behavior.

Temporal network motifs can also be used to measure the frequency of patterns at different time scales. For example, the difference in $\delta$-temporal motif counts for $\delta = 60$ minutes and $\delta = 30$ minutes counts only the motifs that take at least 30 minutes and at most 60 minutes to form. With this type of analysis, we find that certain question-and-answer patterns on Stack Overflow need at least 30 minutes to develop. We also see that in online private messaging, star patterns constructed by outgoing messages sent by one user tend to increase in frequency from time scales of 1 to 20 minutes before peaking and then declining in frequency.

All in all, our work defines a flexible notion of motifs in temporal networks and provides efficient algorithms for counting them. It enables new analyses in a variety of scientific domains and paves a new way for modeling dynamic complex systems.

## 2. RELATED WORK

Our work builds upon the rich literature on network motifs in static graphs, where these models have proved crucial to understanding the mechanisms driving complex systems [19] and to characterizing classes of static networks [25, 29]. Furthermore, motifs are critical for understanding the higher-order organizational patterns in networks [3, 4]. On the algorithmic side, a large amount of research has been devoted simply to counting triangles in undirected static graphs [14].

Prior definitions of temporal network motifs either do not account for edge ordering [30], only have heuristic counting algorithms [7], or assume temporal edges in a motif must be consecutive events for a node [13]. In the last case, the restrictive definition permits fast counting algorithms but misses important structures. For example, many related edges occurring in a short burst at a node would not be counted together. In contrast, $\delta$-temporal motifs capture *every* occasion that edges form a particular pattern within the prescribed time window.

There are several studies on pattern formation in growing networks where one only considers the addition of edges to a static graph over time. In this context, motif-like patterns have been used to create evolution rules that govern the ways that networks develop [5, 24]. The way we consider ordering of temporal edges in our definition of $\delta$-temporal motifs is similar in spirit. There are also several analyses on the formation of triangles in a variety of social networks [9, 12, 15]. In contrast, in the temporal graphs we study here, three nodes may form a triangle several times.

## 3. PRELIMINARIES

We now provide formal definitions of temporal graphs and $\delta$-temporal motifs. In Section 4, we provide algorithms for counting the number of $\delta$-temporal motifs in a given temporal graph.

**Temporal edges and graphs.** We define a *temporal edge* to be a timestamped directed edge between an ordered pair of nodes. We call a collection of temporal edges a *temporal graph* (Fig. 1A). Formally, a temporal graph $T$ on a node set $V$ is a collection of tuples $(u_i, v_i, t_i)$, $i = 1, \ldots, m$, where each $u_i$ and $v_i$ are elements of $V$ and each $t_i$ is a timestamp in $\mathbb{R}$. We refer to a specific $(u_i, v_i, t_i)$ tuple as a *temporal edge*. There can be many temporal edges directed from $u$ to $v$, and we refer to them as *edges between $u$ and $v$*. We assume that the timestamps $t_i$ are unique so that the tuples may be strictly ordered. This assumption makes the presentation of the definitions and algorithms clearer, but our methods can easily be adapted to the case when timestamps are not unique. When it is clear from context, we refer to a temporal edge as simply an *edge*. Finally, by ignoring timestamps and duplicate edges, the temporal graph induces a standard directed graph, which we call the *static graph* $G$ of $T$ with *static edges*, i.e., $(u, v)$ is an edge in $G$ if and only if there is some temporal edge $(u, v, t)$ in $T$.

**$\delta$-temporal motifs and motif instances.** We formalize $\delta$-temporal motifs with the following definition.

**Definition.** *A $k$-node, $l$-edge, $\delta$-temporal motif is a sequence of $l$ edges, $M = (u_1, v_1, t_1), (u_2, v_2, t_2) \ldots, (u_l, v_l, t_l)$ that are time-ordered within a $\delta$ duration, i.e., $t_1 < t_2 \ldots < t_l$ and $t_l - t_1 \leq \delta$, such that the induced static graph from the edges is connected and has $k$ nodes.*

Note that with this definition, many edges between the same pair of nodes may occur in the motif $M$. Also, we note that the purpose of the timestamps is to induce an ordering on the edges. Fig. 1B illustrates a particular 3-node, 3-edge $\delta$-temporal motif.

The above definition provides a template for a particular pattern, and we are interested in how many times a given pattern occurs in a dataset. Intuitively, a collection of edges in a given temporal graph is an instance of a $\delta$-temporal motif $M$ if it matches the same edge pattern and all of the edges occur in the right order within the


| | 15s $(c,a)$ | 17s $(a,c)$ | 25s $(a,b)$ | 28s $(a,c)$ | 30s $(a,c)$ | 32s $(c,a)$ | 35s $(a,c)$ |
|---|---|---|---|---|---|---|---|
| counts$[(a,b)]$ | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| counts$[(a,c)]$ | 0 | 1 | 1 | 1 | 2 | 2 | 3 |
| counts$[(c,a)]$ | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| counts$[(a,b)(a,c)]$ | 0 | 0 | 0 | 1 | 2 | 2 | 3 |
| counts$[(c,a)(a,c)]$ | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| counts$[(a,b)(a,c)(c,a)]$ | 0 | 0 | 0 | 0 | 0 | 2 | 2 |
| start | 1 | 1 | 1 | 3 | 3 | 3 | 3 |
| end | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

**Figure 2: Example execution of Alg. 1 for counting instances of the $\delta$-temporal motif $M$ in Fig. 1. Each column shows the value of counters at the end of the for loop that processes temporal edges. Color indicates change in the variable: incremented (blue), decremented (red), incremented and decremented (purple), or no change (black). At the end of execution, counts$[(a,b)(a,c)(c,a)] = 2$ for the two instances of the temporal motif $M$ with center node $a$. Here we only show the counters needed to count $M$; in total, Alg. 1 maintains 39 total counters for this input edge sequence, 25 of which are non-zero.**

$\delta$ time window (Fig. 1C). Formally, we say that any time-ordered sequence $S = (w_1, x_1, t'_1), \ldots, (w_l, x_l, t'_l)$ of $l$ unique edges is an *instance* of the motif $M = (u_1, v_1, t_1), \ldots, (u_l, v_l, t_l)$ if

1. There exists a bijection $f$ on the vertices such that $f(w_i) = u_i$ and $f(x_i) = v_i$, $i = 1, \ldots, l$, and
2. the edges all occur within $\delta$ time, i.e., $t'_l - t'_1 \leq \delta$

A central goal of this work is to count the number of ordered subsets of edges from a temporal graph $T$ that are instances of a particular motif. In other words, given a $k$-node, $l$-edge $\delta$-temporal motif, we seek to find how many of the $l!\binom{m}{l}$ ordered length-$l$ sequences of edges in the temporal graph $T$ are instances of the motif. A naive approach to this problem would be to simply enumerate all ordered subsets and then check if it is an instance of the motif. In modern datasets, the number of edges $m$ is typically quite large (we analyze a dataset in Section 5 with over two billion edges), and this approach is impractical even for $l = 2$. In the following section, we discuss several faster algorithms for counting the number of instances of $\delta$-temporal motifs in a temporal graph.

# 4. ALGORITHMS

We now present several algorithms for exactly counting the number of instances of $\delta$-temporal motifs in a temporal graph. We first present a general counting algorithm in Section 4.1, which can count instances of any $k$-node, $l$-edge temporal motif faster than simply enumerating over all size-$l$ ordered subsets of edges. This algorithm is optimal for counting 2-node temporal motifs in the sense that it is linear in the number of edges in the temporal graph. In Section 4.2, we provide faster, specialized algorithms for counting specific types of 3-node, 3-edge temporal motifs (Fig. 3).

## 4.1 General counting framework

We begin with a general framework for counting the number of instances of a $k$-node, $l$-edge temporal motif $M$. To start, consider $H$ to be the static directed graph induced by the edges of $M$. A sequence of temporal edges $S$ is an instance of $M$ if and only if the static subgraph induced by edges in $S$ is isomorphic to $H$, the ordering of the edges in $S$ matches the order in $M$, and all the edges in $S$ span a time window of at most $\delta$ time units. This leads to the following general algorithm for counting instances of $M$ in a temporal graph $T$:

**Algorithm 1:** Algorithm for counting the number of instances of all possible $l$-edge $\delta$-temporal motifs in an ordered sequence of temporal edges. We assume the keys of counts$[\cdot]$ are accessed in order of length.

**Input**: Sequence $S'$ of edges $(e_1 = (u_1, v_1), t_1), \ldots, (e_L, t_L)$ with $t_1 < \ldots < t_L$, time window $\delta$
**Output**: Number of instances of each $l$-edge $\delta$-temporal motif $M$ contained in the sequence

```
start ← 1, counts ← Counter(default = 0)
for end = 1, ..., L :
    while t_start + δ < t_end do
        DecrementCounts(e_start), start += 1
    IncrementCounts(e_end)
return counts
Procedure DecrementCounts(e)
    counts[e] −= 1
    for suffix in counts.keys of length < l − 1 :
        counts[concat(e, suffix)] −= counts[suffix]
Procedure IncrementCounts(e)
    for prefix in counts.keys.reverse() of length < l :
        counts[concat(prefix, e)] += counts[prefix]
    counts[e] += 1
```

1. Identify all instances $H'$ of the static motif $H$ induced by $M$ within the static graph $G$ induced by the temporal graph $T$ (e.g., there are three instances of $H$ induced by $M$ in Fig. 1).
2. For each static motif instance $H'$, gather all temporal edges between pairs of nodes forming an edge in $H'$ into an ordered sequence $S' = (u_1, v_1, t_1), \ldots, (u_L, v_L, t_L)$.
3. Count the number of (potentially non-contiguous) subsequences of edges in $S'$ occurring within $\delta$ time units that correspond to instances of $M$.

The first step can use known algorithms for enumerating motifs in static graphs [27], and the second step is a simple matter of fetching the appropriate temporal edges. To perform the third step efficiently, we develop a dynamic programming approach for counting the number of subsequences (instances of motif $M$) that match a particular pattern within a larger sequence ($S'$). The key idea is that, as we stream through an input sequence of edges, the count of a given length-$l$ pattern (i.e., motif) with a given final edge is computed from the current count of the length-$(l-1)$ prefix of the pattern. Inductively, we maintain auxiliary counters of all of the prefixes of the pattern (motif). Second, we also require that all edges in the motif be at most $\delta$ time apart. Thus, we use the notion of a moving time window such that any two edges in the time window are at most $\delta$ time apart. The auxiliary counters now keep track of only the subsequences occurring within the current time window. Last, it is important to note that the algorithm only *counts* the number of instances of motifs rather than *enumerating* them.

Alg. 1 counts *all* possible $l$-edge motifs that occur in a given sequence of edges. The data structure counts$[\cdot]$ maintains auxiliary counts of all (ordered) patterns of length at most $l$. Specifically, counts$[e_1 \cdots e_r]$ is the number of times the subsequence $[e_1 \cdots e_r]$ occurs in the current time window (if $r < l$) or the number of times the subsequence has occurred within all time windows of length $\delta$ (if $r = l$). We also assume the keys of counts$[\cdot]$ are accessed in order of length. Moving the time window forward by adding a new edge into the window, all edges $(e = (u, v), t)$ farther than $\delta$ time from the new edge are removed from the window and the appropriate counts are decremented (the *DecrementCounts()* method). First,

the single edge counts ($[e]$) are updated. Based on these updates, length-2 subsequences formed with $e$ as its first edge are updated and so on, up through length-$(l-1)$ subsequences. On the other hand, when an edge $e$ is added to the window, similar updates take place, but in reverse order, from longest to shortest subsequences, in order to increment counts in subsequences where $e$ is the last edge (the *IncrementCounts()* method). Importantly, length-$l$ subsequence counts are incremented in this step but never decremented. As the time window moves from the beginning to the end of the sequence of edges, the algorithm accumulates counts of all length-$l$ subsequences in all possible time windows of length $\delta$.

Fig. 2 shows the execution of the Alg. 1 for a particular sequence of edges. Note that the figure only displays values of counts$[\cdot]$ for contiguous subsequences of the motif $M$, but the algorithm keeps counts for other subsequences as well. In general, there are $O(l^2)$ contiguous subsequences of an $l$-edge motif $M$, and there are $O(|H|^l)$ total keys in counts$[\cdot]$, where $|H|$ is the number of edges in the static subgraph $H$ induced by $M$, in order to count all $l$-edge motifs in the sequence (i.e., not just motif $M$).

We now analyze the complexity of the overall 3-step algorithm. We assume that the temporal graph $T$ has edges sorted by timestamps, which is reasonable if edges are logged in their order of occurrence, and we pre-process $T$ in linear time such that we can access the sorted list of all edges between $u$ and $v$ in $O(1)$ time. Constructing the time-sorted sequence $S'$ in step 2 of the algorithm then takes $O(\log(|H|)|S'|)$ time. Each edge inputted to Alg. 1 is processed exactly twice: once to increment counts when it enters the time window and once to decrement counts when it exits the time window. As presented in Alg. 1, each update changes $O(|H|^l)$ counters resulting in an overall complexity of $O(|H|^l|S'|)$. However, one could modify Alg. 1 to only update counts for contiguous subsequences of the sequence $M$, which would change $O(l^2)$ counters and have overall complexity $O(l^2|S'|)$. We are typically only interested in small constant values of $|H|$ and $l$ (for our experiments in Section 5, $|H| \leq 3$ and $l = 3$), in which case the running time is linear in the size of the input to the algorithm, i.e., $O(|S'|)$.

In the remainder of this section we analyze our 3-step algorithm with respect to different types of motifs (2-node, stars, and triangles) and argue benefits as well as deficiencies of the proposed framework. We show that for 2-node motifs, our general counting framework takes time linear in the total number of edges $m$. Since all the input data needs to be examined for computing exact counts, this means the algorithm is optimal for 2-node motifs. However, we also show that for star and triangle motifs the algorithm is not optimal, which then motivates us to design faster algorithms in Sec. 4.2.

**General algorithm for 2-node motifs.** We first show how to map 2-node motifs to the framework described above. Any induced graph $H$ of a 2-node $\delta$-temporal motif is either a single or a bidirectional edge. In either case, it is straightforward to enumerate over all instances of $H$ in the static graph. This leads to the following procedure: (1) for each pair of nodes $u$ and $v$ for which there is at least one edge, gather and sort the edges in either direction between $u$ and $v$; (2) call Alg. 1 with these edges. The obtain the total motif count the counts from each call to Alg. 1 are then summed together.

We only need to input each edge to Alg. 1 once, and under the assumption that we can access the sorted directed edges from one node to another in $O(1)$ time, the merging of edges into sorted order takes linear time. Therefore, the total running time is $O(2^l m)$, which is linear in the number of temporal edges $m$. We are mostly interested in small patterns, i.e., cases when $l$ is a small constant. Thus, this methodology is optimal (linear in the input, $m$) for counting 2-node $\delta$-temporal motif instances.

**General algorithm for star motifs.** Next, we consider $k$-node, $l$-edge star motifs $M$, whose induced static graph $H$ consists of a center node and $k-1$ neighbors, where edges may occur in either direction between the center node and a neighbor node. For example, in the top left corner of Fig. 3, $M_{1,1}$ is a star motif with all edges pointing toward the center node. In such motifs, the induced static graph $H$ contains at most $2(k-1) = 2k-2$ static edges—one incoming and outgoing edge from the center node to each neighbor node. We have the following method for counting the number of instances of $k$-node, $l$-edge star motifs: (1) for each node $u$ in the static graph and for each unique set of $k-1$ neighbors, gather and sort the edges in either direction between $u$ and the neighbors; (2) count the number of instances of $M$ using Alg. 1. The counts from each call to Alg. 1 are summed over all center nodes.

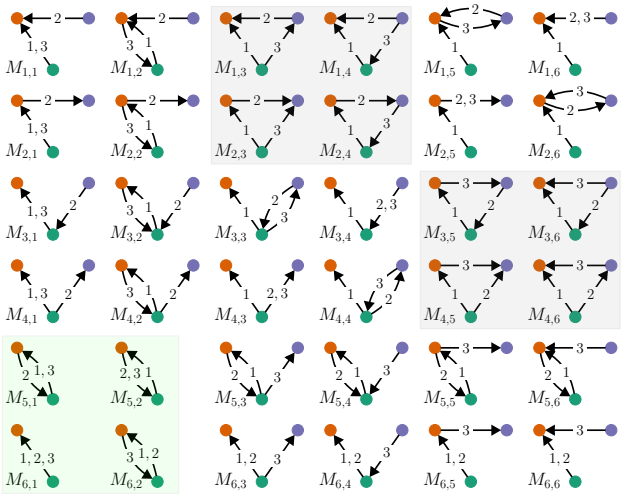


**Figure 3: All 2-node and 3-node, 3-edge $\delta$-temporal motifs. The green background highlights the four 2-node motifs (bottom left) and the grey background highlights the eight triangles. The 24 other motifs are stars. We index the 36 motifs $M_{i,j}$ by 6 rows and 6 columns. The first edge in each motif is from the green to the orange node. The second edge is the same along each row, and the third edge is the same along each column.**

The major drawback of this approach is that we have to loop over each size-$(k-1)$ neighbor set. This can be prohibitively expensive even when $k = 3$ if the center node has large degree. In Section 4.2, we shall design an algorithm that avoids this issue for the case when the star motif has $l = 3$ edges and $k = 3$.

**General algorithm for triangle motifs.** In triangle motifs, the induced graph $H$ consists of 3 nodes and at least one directed edge between any pair of nodes (see Fig. 3 for all eight of the 3-edge triangle motifs). The induced static graph $H$ of $M$ contains at least three and at most six static edges. A straightforward algorithm for counting $l$-edge triangle motifs in a temporal graph $T$ is:

1. Use a fast static graph triangle enumeration algorithm to find all triangles in the static graph $G$ induced by $T$ [14].
2. For each triangle $(u, v, w)$, merge all temporal edges from each pair of nodes to get a time-sorted list of edges. Use Alg. 1 to count the number of instances of $M$.

This approach is problematic as the edges between a pair of nodes may participate in many triangles. Fig. 4 shows a worst-case example for the motif $M = (w, u, t'_1), (w, v, t'_2), (u, v, t'_3)$ with $\delta = \infty$. In this case, the timestamps are ordered by their index. There are $m - 2n$ edges between $u$ and $v$, and each of these edges forms an instance of $M$ with every $w_i$. Thus, the overall worst-case running time of the algorithm is $O(\text{TriEnum} + m\tau)$, where TriEnum is the time to enumerate the number of triangles $\tau$ in the static graph. In

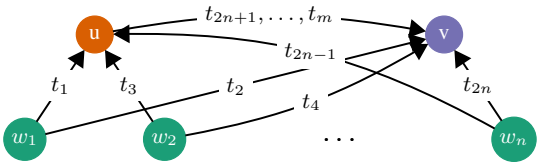



**Figure 4: Worst-case example for counting triangular motifs with Alg. 1.**

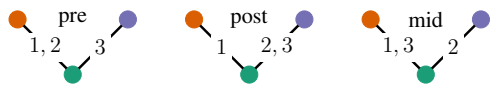
the following section, we devise an algorithm that significantly reduces the dependency on $\tau$ from linear to sub-linear (specifically, $\sqrt{\tau}$) when there are $l = 3$ edges.

## 4.2 Faster algorithms

The general counting algorithm from the previous subsection counts the number of instances of any $k$-node, $l$-edge $\delta$-temporal motif, and is also optimal for 2-node motifs. However, the computational cost may be expensive for other motifs such as stars and triangles. We now develop specialized algorithms that count certain motif classes faster. Specifically, we design faster algorithms for counting all 3-node, 3-edge star and triangle motifs (Fig. 3 illustrates these motifs). Our algorithm for stars is linear in the input size, so it is optimal up to constant factors.

**Fast algorithm for 3-node, 3-edge stars.** With 3-node, 3-edge star motifs, the key drawback of using the previous algorithmic approach would be that we would have to loop over all pairs of neighbors given a center node. Instead, we will count all instances of star motifs for a given center node in just a single pass over the edges adjacent to the center node.

We use a dynamic programming approach for counting star motifs. First, note that every temporal edge in a star with center $u$ is defined by (1) a neighbor node, (2) a direction of the edge (outward from or inward to $u$), and (3) the timestamp. With this insight we then notice that there are 3 classes of star motifs on 3 nodes and 3 edges:




where each class has $2^3 = 8$ motifs for each of the possible directions on the three edges.

Now, suppose we process the time-ordered sequence of edges containing the center node $u$. We maintain the following counters when processing an edge with timestamp $t_j$:

- pre_sum[dir1, dir2] is the number of sequentially ordered pairs of edges in $[t_j - \delta, t_j)$ where the first edge points in direction dir1 and the second edge points in direction dir2
- post_sum[dir1, dir2] is the analogous counter for the time window $(t_j, t_j + \delta]$.
- mid_sum[dir1, dir2] is the number of pairs of edges where the first edge is in direction dir1 and occurred at time $t < t_j$ and the second edge is in direction dir2 and occurred at time $t' > t_j$ such that $t' - t \leq \delta$.

If we are currently processing an edge, the "pre" class gets pre_sum[dir1, dir2] new motif instances for any choice of directions dir1 and dir2 (specifying the first two edge directions) and the current edge serves as the third edge in the motif (hence specifying the third edge direction). Similar updates are made with the post_sum$[\cdot, \cdot]$ and mid_sum$[\cdot, \cdot]$ counters, where the current edge serves as the first or second edge in the motif, respectively.

In order for our algorithm to be efficient, we must quickly update our counters. To aid in this, we introduce two additional counters:

**Algorithm 2:** Algorithmic framework for faster counting of 3-node, 3-edge star and triangle temporal motifs. The fast star counting method (Alg. 3) and triangle counting method (Alg. 4) implement different versions of the *Push()*, *Pop()*, and *ProcessCurrent()* subroutines.

**Input**: Sequence of edges $(e_1 = (u_1, v_1), t_1), \ldots, (e_L, t_L)$ with $t_1 < \ldots < t_L$, time window $\delta$

```
Initialize counters pre_nodes, post_nodes, mid_sum, pre_sum, and post_sum; start ← 1, end ← 1
for j = 1, ..., L :
    while t_start + δ < t_j do
        Pop(pre_nodes, pre_sum, e_start), start += 1
    while t_end ≤ t_j + δ do
        Push(post_nodes, post_sum, e_end), end += 1
    Pop(post_nodes, post_sum, e_j)
    ProcessCurrent(e_j)
    Push(pre_nodes, pre_sum, e_j)
```

- pre_nodes[dir, $v_i$] is the number of times node $v_i$ has appeared in an edge with $u$ with direction dir in the time window $[t_j - \delta, t_j)$
- post_nodes[dir, $v_i$] is the analogous counter but for the time window $(t_j, t_j + \delta]$.

Following the ideas of Alg. 1, it is easy to update these counters when we process a new edge. Consequently, pre_sum$[\cdot, \cdot]$, post_sum$[\cdot, \cdot]$, and mid_sum$[\cdot, \cdot]$ can be maintained when processing an edge with just a few simple primitives:

- *Push()* and *Pop()* update the counts for pre_nodes$[\cdot, \cdot]$, post_nodes$[\cdot, \cdot]$, pre_sum$[\cdot, \cdot]$ and post_sum$[\cdot, \cdot]$ when edges enter and leave the time windows $[t_j - \delta, t_j)$ and $(t_j, t_j + \delta]$.
- *ProcessCurrent()* updates motif counts involving the current edge and updates the counter mid_sum$[\cdot, \cdot]$.

We describe the general procedure in Alg. 2, which will also serve as the basis for our fast triangle counting procedure, and Alg. 3 implements the subroutines *Push()*, *Pop()*, and *ProcessCurrent()* for counting instances of 3-node, 3-edge star motifs. The count_pre$[\cdot, \cdot, \cdot]$, count_post$[\cdot, \cdot, \cdot]$, and count_mid$[\cdot, \cdot, \cdot]$ counters in Alg. 3 maintain the counts of the three different classes of stars described above.

Finally, we note that our counting scheme incorrectly includes instances of 2-node motifs such as $M = (u, v_i, t_1), (u, v_i, t_2), (u, v_i, t_3)$, but we can use the efficient 2-node motif counting algorithm to account for this. Putting everything together, we have the following procedure:

1. For each node $u$ in the temporal graph $T$, get a time-ordered list of all edges containing $u$.
2. Use Algs. 2 and 3 to count star motif instances.
3. For each neighbor $v$ of a star center $u$, subtract the 2-node motif counts using Alg. 1.

If the $m$ edges of $T$ are time-sorted, the first step can be done in linear time. The second and third steps run in linear time in the input size. Each edge is used in steps 2 and 3 exactly twice: once for each end point as the center node. Thus, the overall complexity of the algorithm is $O(m)$, which is optimal up to constant factors.

**Fast algorithm for 3-edge triangle motifs.** While our fast star counting routine relied on counting motif instances for all edges adjacent to a given *node*, our fast triangle algorithm is based on counting instances for all edges adjacent to a given *pair of nodes*. Specifically, given a pair of nodes $u$ and $v$ and a list of common neighbors $w_1, \ldots, w_d$, we count the number of motif instances for triangles $(w_i, u, v)$. Given all of the edges between these three

**Algorithm 3:** Implementation of Alg. 2 subroutines for efficiently counting instances of 3-node, 3-edge star motifs. Temporal edges are specified by a neighbor nbr, a direction dir (incoming or outgoing), and a timestamp. The ":" notation represents a selection of all indices in an array.

```
Initialize counters count_pre, count_post, count_mid
Procedure Push(node_count, sum, e = (nbr, dir))
    sum[:, dir] += node_count[:, nbr]
    node_count[dir, nbr] += 1
Procedure Pop(node_count, sum, e = (nbr, dir))
    node_count[dir, nbr] −= 1
    sum[dir, :] −= node_count[:, nbr]
Procedure ProcessCurrent(e = (nbr, dir))
    mid_sum[:, dir] −= pre_nodes[:, nbr]
    count_pre[:, :, dir] += pre_sum[:, :]
    count_post[dir, :, :] += post_sum[:, :]
    count_mid[:, dir, :] += mid_sum[:, :]
    mid_sum[dir, :] += post_nodes[:, nbr]
return count_pre, count_post, count_mid
```

nodes, the counting procedures are nearly identical to the case of stars. We use the same general counting method (Alg. 2), but the behavior of the subroutines *Push()*, *Pop()*, and *ProcessCurrent()* depends on whether or not the edge is between $u$ and $v$.

These methods are implemented in Alg. 4. The input is a list of edges adjacent to a given pair of neighbors $u$ and $v$, where each edge consists of four pieces of information: (1) a neighbor node nbr, (2) an indicator of whether or not the node nbr connects to node $u$ or node $v$, (3) the direction dir of the edge, and (4) the timestamp. The node counters (pre_nodes$[\cdot,\cdot,\cdot]$ and post_nodes$[\cdot,\cdot,\cdot]$) in Alg. 4 have an extra dimension compared to Alg. 3 to indicate whether the counts correspond to edges containing node $u$ or node $v$ (denoted by "uorv"). Similarly, the sum counters (pre_sum$[\cdot,\cdot,\cdot]$, mid_sum$[\cdot,\cdot,\cdot]$ and post_sum$[\cdot,\cdot,\cdot]$) have an extra dimension to denote if the first edge is incident on node $u$ or node $v$.

Recall that the problem with counting triangle motifs by the general framework in Alg. 1 is that a pair of nodes with many edges might have to be counted for many triangles in the graph. However, with Alg. 4, we can simultaneously count all triangles adjacent to a given pair of nodes. What remains is that we must assign each triangle in the static graph to a pair of nodes. Here, we propose to assign each triangle to the pair of nodes in that triangle containing the largest number of edges, which is sketched in Alg. 5. Alg. 5 aims to process as many triangles as possible for pairs of nodes with many edges. The following theorem says that this is faster than simply counting for each triangle (described in Section 4.1). Specifically, we reduce $O(m\tau)$ complexity to $O(m\sqrt{\tau})$.

**Theorem.** *In the worse case, Alg. 5 runs in time* $O(\text{TriEnum} + m\sqrt{\tau})$*, where* TriEnum *is the time to enumerate all triangles in the static graph* $G$*,* $m$ *is the total number of temporal edges, and* $\tau$ *is the number of static triangles in* $G$*.*

*Proof.* Let $\sigma_i$ be the number of edges between the $i$th pair of nodes with at least one edge, and let $p_i \geq 1$ be the number of times that edges on this pair are used in a call to Alg. 4 by Alg. 5. Since Alg. 4 runs in linear time in the number of edges in its input, the total running time is on the order of $\sum_i \sigma_i p_i$.

The $\sigma_i$ are fixed, and we wish to find the values of $p_i$ that maximize the summation. Without loss of generality, assume that the $\sigma_i$ are in decreasing order so that the most number of edges between a pair of nodes is $\sigma_1$. Consequently, $p_i \leq i$. Note that each triangle contributes to at most a constant repeat processing of edges for a given pair of nodes. Hence, $\sum_i p_i \leq c\tau$ for some constant $c$. The summation $\sum_i \sigma_i p_i$ is maximized when $p_1 = 1, p_2 = 2$, and so on up to some index $j = O(\sqrt{\tau})$ for which $\sum_{i=1}^{j} p_j = \sum_{i=1}^{j} i = c\tau$. Now given that the $p_j$ are fixed and the $\sigma_i$ are ordered, the summation is maximized when $\sigma_1 = \sigma_2 = \ldots = \sigma_j = m/j$. In this case, $\sum_i \sigma_i p_i = \sum_{i=1}^{j} (m/j)(1+i) = O(m\sqrt{\tau})$. □

**Algorithm 4:** Implementation of Alg. 2 subroutines for counting 3-edge triangle motifs containing a specified pair of nodes $u$ and $v$. Temporal edges are specified by a neighbor nbr, a direction dir (incoming or outgoing), an indicator "uorv" denoting if the edge connects to $u$ or $v$, and a timestamp. The ":" notation represents a selection of all indices in an array.

```
Initialize counter count
Procedure Push(node_count, sum, e = (nbr, dir, uorv))
    if nbr ∈ {u, v} then return
    sum[1-uorv, :, dir] += node_count[1-uorv, :, nbr]
    node_count[uorv, dir, nbr] += 1
Procedure Pop(node_count, sum, e = (nbr, dir, uorv))
    if nbr ∈ {u, v} then return
    node_count[uorv, dir, nbr] −= 1
    sum[uorv, dir, :] −= node_count[1-uorv, :, nbr]
Procedure ProcessCurrent(e = (nbr, dir, uorv))
    if nbr ∉ {u, v} then
        mid_sum[1-uorv, :, dir] −= pre_nodes[1-uorv, :, nbr]
        mid_sum[uorv, dir, :] += post_nodes[1-uorv, :, nbr]
    else
        utov = (nbr == u) XOR dir
        for 0 ≤ i, j, k ≤ 1 :
            count[i, j, k] += mid_sum[j XOR utov, i, k]
                  +post_sum[i XOR utov, j, 1 − k]
                  +pre_sum[k XOR utov, 1 − i, 1 − j]
/* count key map to Fig. 3:                          */
/* [0,0,0] ↦ M1,3, [0,0,1] ↦ M1,4, [0,1,0] ↦ M2,3    */
/* [0,1,1] ↦ M2,4, [1,0,0] ↦ M3,5, [1,0,1] ↦ M3,6    */
/* [1,1,0] ↦ M4,5, [1,1,1] ↦ M4,6                    */
return count
```

**Algorithm 5:** Sketch of fast algorithm for counting the number of 3-edge $\delta$-temporal triangle motifs in a temporal graph $T$.

```
Enumerate all triangles in the undirected static graph G of T
σ ← number of temporal edges on each static edge e in G
foreach static triangle Δ = (e1, e2, e3) in G do
    e_max = arg max_{e∈Δ} {σ_e}
    foreach e in Δ do Add e_max to edge set ℓ_e
    /* e' ∈ ℓ_e if e ∈ Δ and Δ assigned to e'      */
foreach temporal edge (e = (u, v), t) in time-sorted T do
    foreach e' in ℓ_e do Append (e, t) to temporal-edge list a_e'
    /* (e, t) ∈ a_e' if e ∈ Δ and Δ assigned to e' */
foreach undirected edge e in G do
    Update counts using Alg. 4 with input a_e
```

## 5. EXPERIMENTS

Next, we use our algorithms to reveal patterns in a variety of temporal network datasets. We find that the number of instances of various $\delta$-temporal motifs reveal basic mechanisms of the networks. Datasets and implementations of our algorithms are available at http://snap.stanford.edu/temporal-motifs.

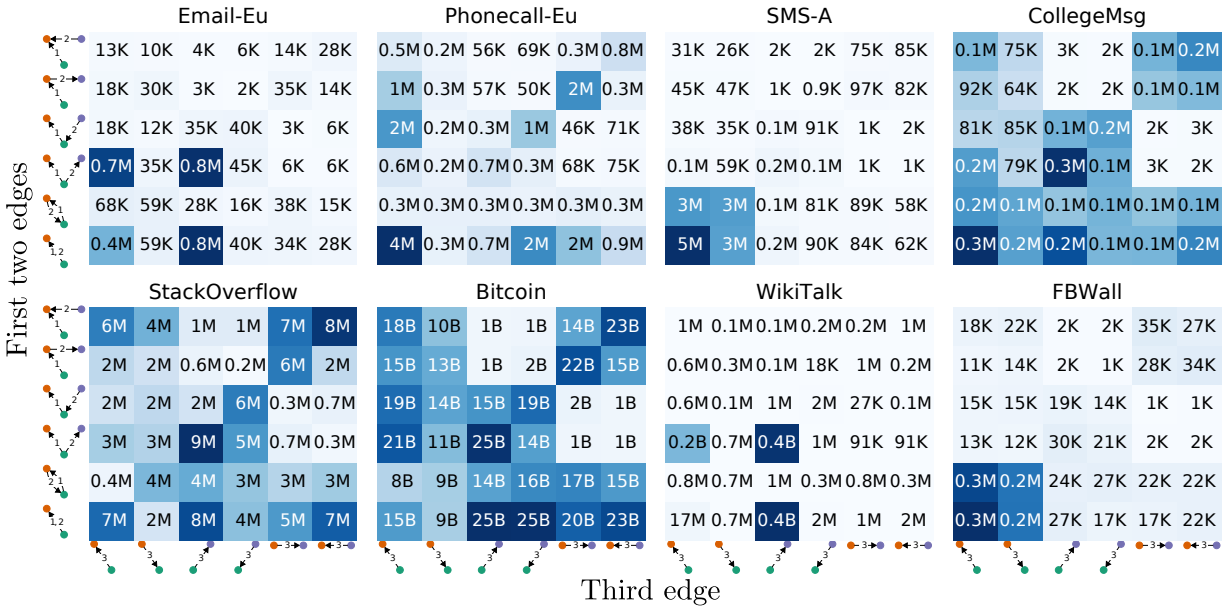



**Figure 5: Counts of instances of all 2- and 3-node, 3-edge $\delta$-temporal motifs with $\delta = 1$ hour. For each dataset, counts in the $i$th row and $j$th column is the number of instances of motif $M_{i,j}$ (see Fig. 3); this motif is the union of the two edges in the row label and the edge in the column label. For example, there are 0.7 million instances of motif $M_{4,1}$ in the EMAIL-EU dataset. The color for the count of motif $M_{i,j}$ indicates the fraction over all $M_{i,j}$ on a linear scale—darker blue means a higher count.**

**Table 1: Summary statistics of datasets.**

| dataset | # nodes | # static edges | # edges | time span (days) |
|---|---|---|---|---|
| EMAIL-EU | 986 | 2.49K | 332K | 803 |
| PHONECALL-EU | 1.05M | 2.74M | 8.55M | 7 |
| SMS-A | 44.1K | 67.2K | 545K | 338 |
| COLLEGEMSG | 1.90K | 20.3K | 59.8K | 193 |
| STACKOVERFLOW | 2.58M | 34.9M | 47.9M | 2774 |
| BITCOIN | 24.6M | 88.9M | 123M | 1811 |
| FBWALL | 45.8K | 264K | 856K | 1560 |
| WIKITALK | 1.09M | 3.13M | 6.10M | 2277 |
| PHONECALL-ME | 18.7M | 360M | 2.04B | 364 |
| SMS-ME | 6.94M | 51.5M | 800M | 89 |

## 5.1 Data

We gathered a variety of datasets in order to study the patterns of $\delta$-temporal motifs in several domains. The datasets are described below and summary statistics are in Table 1. The time resolution of the edges in all datasets is one second.

**EMAIL-EU.** This dataset is a collection of emails between members of a European research institution [17]. An edge $(u, v, t)$ signifies that person $u$ sent person $v$ an email at time $t$.

**PHONECALL-EU.** This dataset was constructed from telephone call records for a major European service provider. An edge $(u, v, t)$ signifies that person $u$ called person $v$ starting at time $t$.

**SMS-A.** Short messaging service (SMS) is a texting service provided on mobile phones. In this dataset, an edge $(u, v, t)$ means that person $u$ sent an SMS message to person $v$ at time $t$ [28].

**COLLEGEMSG.** This dataset is comprised of private messages sent on an online social network at the University of California, Irvine [21]. Users could search the network for others and then initiate conversation based on profile information. An edge $(u, v, t)$ means that user $u$ sent a private message to user $v$ at time $t$.

**STACKOVERFLOW.** On stack exchange web sites, users post questions and receive answers from other users, and users may comment on both questions and answers. We derive a temporal network by creating an edge $(u, v, t)$ if, at time $t$, user $u$: (1) posts an answer to user $v$'s question, (2) comments on user $v$'s question, or (3) comments on user $v$'s answer. We formed the temporal network from the entirety of Stack Overflow's history up to March 6, 2016.

**BITCOIN.** Bitcoin is a decentralized digital currency and payment system. This dataset consists of all payments made up to October 19, 2014 [11]. Nodes in the network correspond to Bitcoin addresses, and an individual may have several addresses. An edge $(u, v, t)$ signifies that bitcoin was transferred from address $u$ to address $v$ at time $t$.

**FBWALL.** The edges of this dataset are wall posts between users on the social network Facebook located in the New Orleans region [26]. Any friend of a given user can see all posts on that user's wall, so communication is public among friends. An edge $(u, v, t)$ means that user $u$ posted on user $v$'s wall at time $t$.

**WIKITALK.** This dataset represents edits on user talk pages on Wikipedia [16]. An edge $(u, v, t)$ signifies that user $u$ edited user $v$'s talk page at time $t$.

**PHONECALL-ME and SMS-ME.** This dataset is constructed from phone call and SMS records of a large telecommunications service provider in the Middle East. An edge $(u, v, t)$ in PHONECALL-ME means that user $u$ initiated a call to user $v$ at time $t$. An edge $(u, v, t)$ in SMS-ME means that user $u$ sent an SMS message to user $v$ at time $t$. We use these networks for scalability experiments in Section 5.3.

## 5.2 Empirical observations of motif counts

We first examine the distribution of 2- and 3-node, 3-edge motif instance counts from 8 of the datasets described in Section 5.1 with $\delta = 1$ hour (Fig. 5). We choose 1 hour for the time window as this is close to the median time for a node to take part in

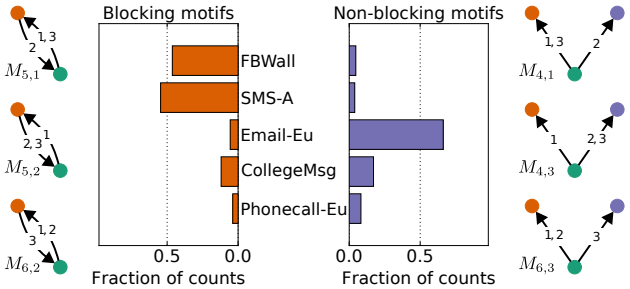



**Figure 6: Fraction of all 2 and 3-node, 3-edge $\delta$-temporal motif counts that correspond to two groups of motifs ($\delta$ = 1 hour). Motifs on the left capture "blocking" behavior, common in SMS messaging and Facebook wall posting, and motifs on the right exhibit "non-blocking" behavior, common in email.**

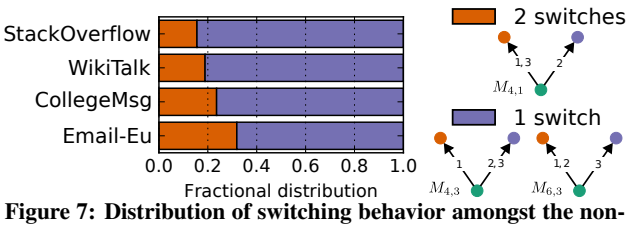



**Figure 7: Distribution of switching behavior amongst the non-blocking motifs. Switching is least common on Stack Overflow and most common in email.**

three edges in most of our datasets. We make a few empirical observations uniquely available due to temporal motifs and provide possible explanations for these observations.

**Blocking communication.** If an individual typically waits for a reply from one individual before proceeding to communicate with another individual, we consider it a *blocking* form of communication. A typical conversation between two individuals characterized by fast exchanges happening back and forth is blocking as it requires complete attention of both individuals. We capture this behavior in the "blocking motifs" $M_{5,1}$, $M_{5,2}$ and $M_{6,2}$, which contain 3 edges between two nodes with at least one edge in either direction (Fig. 6, left). However, if the reply doesn't arrive soon, we might expect the individual to communicate with others without waiting for a reply from the first individual. This is a non-blocking form of communication and is captured by the "non-blocking motifs" $M_{4,1}$, $M_{4,3}$ and $M_{6,3}$ having edges originating from the same source but directed to different destinations (Fig. 6, right)

The fractions of counts corresponding to the blocking and non-blocking motifs out of the counts for all 36 motifs in Fig. 3 uncover several interesting characteristics in communication networks ($\delta$ = 1 hour; see Fig. 6). In FBWALL and SMS-A, blocking communication is vastly more common, while in EMAIL-EU non-blocking communication is prevalent. Email is not a dynamic method of communication and replies within an hour are rare. Thus, we would expect non-blocking behavior. Interestingly, the COLLEGEMSG dataset shows both behaviors as we might expect individuals to engage in multiple conversations simultaneously. In complete contrast, the PHONECALL-EU dataset shows neither behavior. A simple explanation is that that a single edge (a phone call) captures an entire conversation and hence blocking behavior does not emerge.

**Cost of switching.** Amongst the non-blocking motifs discussed above, $M_{4,1}$ captures two consecutive switches between pairs of nodes whereas $M_{4,3}$ and $M_{6,3}$ each have a single switch (Fig. 7, right). Prevalence of $M_{4,1}$ indicates a lower cost of switching targets, whereas prevalence of the other two motifs are indicative of a higher cost. We observe in Fig. 7 that the ratio of 2-switch to 1-switch motif counts is the least in STACKOVERFLOW, followed by WIKITALK, COLLEGEMSG and then EMAIL-EU. On Stack Overflow and Wikipedia talk pages, there is a high cost to switch targets because of peer engagement and depth of discussion. On the other hand, in the COLLEGEMSG dataset there is a lesser cost to switch because it lacks depth of discussion within the time frame of $\delta$ = 1 hour. Finally, in EMAIL-EU, there is almost no peer engagement and cost of switching is negligible.

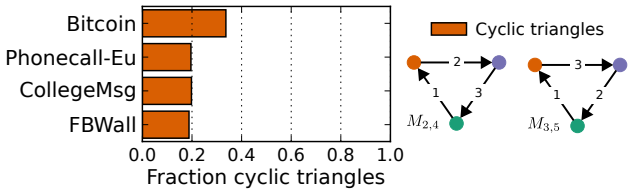



**Figure 8: Fraction of 3-edge $\delta$-temporal triangle motif counts ($\delta$ = 1 hour) corresponding to cyclic triangles (right) in the four datasets for which this fraction is the largest. BITCOIN has a much higher fraction compared to all other datasets.**

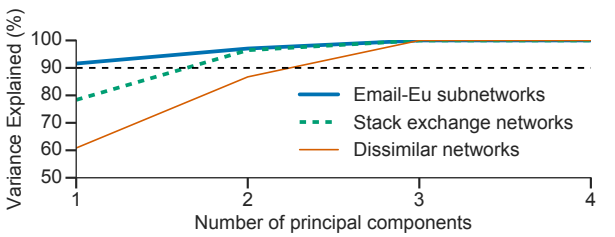



**Figure 9: Percentage of explained variance of relative counts of collections of datasets plotted as a function of the number of principal components. In datasets from the same domain, 90% of variance is explained with fewer components.**

**Cycles in BITCOIN.** Of the eight 3-edge triangle motifs, $M_{2,4}$ and $M_{3,5}$ are cyclic, i.e., the target of each edge serves as the source of another edge. We observe in Fig. 8 that the fraction of triangles that are cyclic is much higher in BITCOIN compared to any other dataset. This can be attributed to the transactional nature of BITCOIN where the total amount of bitcoin is limited. Since remittance (outgoing edges) is typically associated with earnings (incoming edges), we should expect cyclic behavior.

**Datasets from the same domain have similar counts.** Static graphs from similar domains tend to have similar motif count distributions [18, 25, 29]. Here, we find similar results in temporal networks. We formed two collections of datasets from similar domains. First, we took subsets of the EMAIL-EU dataset corresponding to email communication within four different departments at the institution. Second, we constructed temporal graphs from the stack exchange communities Math Overflow, Super User, and Ask Ubuntu to study in conjunction with the STACKOVERFLOW dataset. We form count distributions by normalizing the counts of the 36 different motifs in Fig. 5. For datasets from a similar domain, we expect that if the count distributions are similar, then most of the variance is captured by a few principal components. To compare, we use four datasets from dissimilar domains (EMAIL-EU, PHONECALL-EU, SMS-A, WIKITALK). Fig. 9 shows that to explain 90% variance, EMAIL-EU subnetworks need just one principal component, stack exchange networks need two, and the dissimilar networks need three.

**Motif counts at varying time scales.** We now explore how motif counts change at different time scales. For the STACKOVERFLOW dataset we counted the number of instances of 2- and 3-node, 3-edge $\delta$-temporal motifs for $\delta$ = 60, 300, 1800, and 3600 seconds

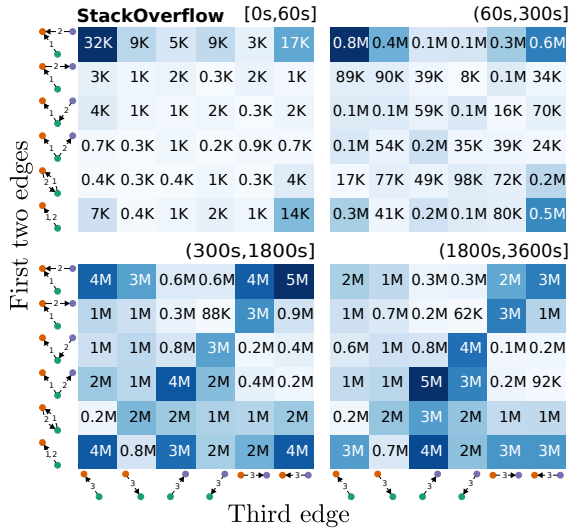



**Figure 10: Counts for all 2- and 3-node, 3-edge $\delta$-temporal motifs in four time intervals for the StackOverflow dataset. For each interval, the count in the $i$th row and $j$th column is the number of instances of motif $M_{i,j}$ (see Fig. 3).**

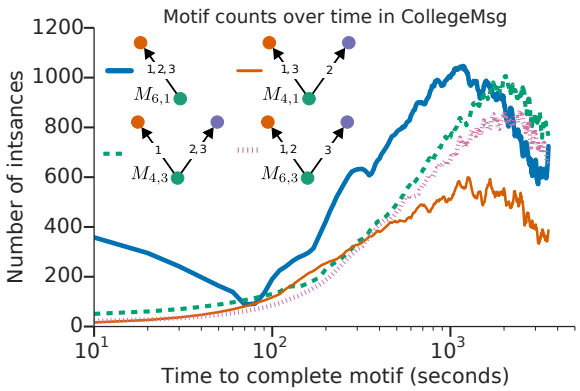



**Figure 11: Counts over various time scales for the motifs representing a node sending 3 outgoing messages to 1 or 2 neighbors in the CollegeMsg dataset.**

(Fig. 10). These counts determine the number of motifs that completed in the intervals [0, 60], (60, 300], (300, 1800s], and (1800, 3600] seconds (e.g., subtracting 60 second counts from 300 second counts gives the interval (60, 300]). Observations at smaller timescales reveal phenomenon which start to get eclipsed at larger timescales. For instance, on short time scales, motif $M_{1,1}$ (Fig. 10, top-left corner) is quite common. We suspect this arises from multiple, quick comments on the original question, so the original poster receives many incoming edges. At larger time scales, this behavior is still frequent but relatively less so. Now let us compare counts for $M_{1,5}$, $M_{1,6}$, $M_{2,5}$, $M_{2,6}$ (the four in the top right corner) with counts for $M_{3,3}$, $M_{3,4}$, $M_{4,3}$, $M_{4,4}$ (the four in the center). The former counts likely correspond to conversations with the original poster while the latter are constructed by the same user interacting with multiple questions. Between 300 and 1800 seconds (5 to 30 minutes), the former counts are relatively more common while the latter counts only become more common after 1800 seconds. A possible explanation is that the typical length of discussions on a post is about 30 minutes, and later on, users answer other questions.

Next, we examine messaging behavior in the CollegeMsg dataset at fine-grained time intervals. We counted the number of motifs consisting of a single node sending three outgoing messages

**Table 2: Time to count the eight 3-edge $\delta$-temporal triangle motifs ($\delta = 3600$) using the general counting method (Alg. 1) and the fast counting method (Alg. 5).**

| dataset | # static triangles | time, Alg. 1 (seconds) | time, Alg. 5 (seconds) | speedup |
|---|---|---|---|---|
| WikiTalk | 8.11M | 51.1 | 26.6 | 1.92x |
| Bitcoin | 73.1M | 27.3K | 483 | 56.5x |
| SMS-ME | 78.0M | 2.54K | 1.11K | 2.28x |
| StackOverflow | 114M | 783 | 606 | 1.29x |
| Phonecall-ME | 667M | 12.2K | 8.59K | 1.42x |

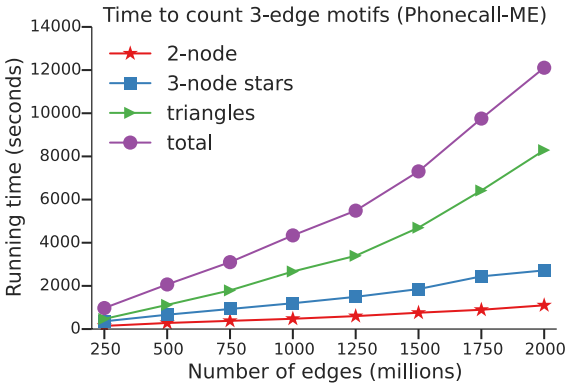



**Figure 12: Time to count 3-edge motifs on the first $k$ temporal million edges in the Phonecall-ME as a function of $k$.**

to one or two neighbors (motifs $M_{6,1}$, $M_{6,3}$, $M_{4,1}$, and $M_{4,3}$) in the time bins $[10(i-1), 10i)$ seconds, $i = 1, \ldots, 500$ (Fig. 11). We first notice that at small time scales, the motif consisting of three edges to a single neighbor ($M_{6,1}$) occurs frequently. This pattern could emerge from a succession of quick introductory messages. Overall, motif counts increase from roughly 1 minute to 20 minutes and then decline. Interestingly, after 5 minutes, counts for the three motifs with one switch in the target ($M_{6,1}$, $M_{6,3}$, and $M_{4,3}$) grow at a faster rate than the counts for the motif with two switches ($M_{4,1}$). As mentioned above, this pattern could emerge from a tendency to send several messages in one conversation before switching to a conversation with another friend.

## 5.3 Algorithm scalability

Finally, we performed scalability experiments of our algorithms. All algorithms were implemented in C++, and all experiments ran using a single thread of a 2.4GHz Intel Xeon E7-4870 processor. We did not measure the time to load datasets into memory, but our timings include all pre-processing time needed by the algorithms (e.g., the triangle counting algorithms first find triangles in the static graph). We emphasize that our implementation is single threaded, and the methods can be sped up with a parallel algorithm.

First, we used both the general counting method (Alg. 1) and the fast counting method (Alg. 5) to count the number of all eight 3-edge $\delta$-temporal triangle motifs in our datasets ($\delta = 1$ hour). Table 2 reports the running times of the algorithms for all datasets with at least one million triangles in the static graph. For all of these datasets, our fast temporal triangle counting algorithm provides significant performance gains over the general counting method, ranging between a 1.29x and a 56.5x speedup. The gains of the fast algorithm are the largest for Bitcoin, which is due to some pairs of nodes having many edges between them and also participating in many triangles.

Second, we measured the time to count various 3-edge $\delta$-temporal motifs in our largest dataset, Phonecall-ME. Specif-

ically, we measured the time to compute (1) 2-node motifs, (2) 3-node stars, and (3) triangles on the first $k$ million edges in the dataset for $k = 250, 500, \ldots, 2000$ (Fig. 12). The time to compute the 2-node, 3-edge motifs and the 3-node, 3-edge stars scales linearly, as expected from our algorithm analysis. The time to count triangle motifs grows superlinearly and becomes the dominant cost when there is a large number of edges. For practical purposes, the running times are quite modest. With two billion edges, our methods take less than 3.5 hours to complete (executing sequentially).

# 6. DISCUSSION

We have developed $\delta$-temporal network motifs as a tool for analyzing temporal networks. We introduced a general framework for counting instances of any temporal motif as well as faster algorithms for certain classes of motifs and found that motif counts reveal key structural patterns in a variety of temporal network datasets. Our work opens a number of avenues for additional research. First, our fast algorithms are designed for 3-node, 3-edge star and triangle motifs. We expect that the same general techniques can be used to count more complex temporal motifs. Next, it is important to note that our fast algorithms only *count* the number of instances of motifs rather than *enumerate* the instances. This concept has also been used to accelerate static motif counting [22]. Temporal motif enumeration algorithms provide an additional algorithmic design challenge. There is also a host of theoretical questions in this area for lower bounds on temporal motif counting. Finally, motif counts can also be measured with respect to a null model [13, 19]. Such analysis may yield additional discoveries. Importantly, our algorithms will speed up such computations, which use raw counts from many random instances of a generative null model.

**Acknowledgements.** We thank Moses Charikar for valuable discussion. This research has been supported in part by NSF IIS-1149837, ARO MURI, DARPA SIMPLEX and NGS2, Boeing, Bosch, Huawei, Lightspeed, SAP, Tencent, Volkswagen, Stanford Data Science Initiative, and a Stanford Graduate Fellowship.

# 7. REFERENCES

[1] M. Araujo, S. Papadimitriou, S. Günnemann, C. Faloutsos, P. Basu, A. Swami, E. E. Papalexakis, and D. Koutra. Com2: fast automatic discovery of temporal ('comet') communities. In *PAKDD*, 2014.

[2] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[3] A. R. Benson, D. F. Gleich, and J. Leskovec. Tensor spectral clustering for partitioning higher-order network structures. In *SDM*, 2015.

[4] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[5] M. Berlingerio, F. Bonchi, B. Bringmann, and A. Gionis. Mining graph evolution rules. In *ECML PKDD*, 2009.

[6] D. M. Dunlavy, T. G. Kolda, and E. Acar. Temporal link prediction using matrix and tensor factorizations. *TKDD*, 5(2):10, 2011.

[7] S. Gurukar, S. Ranu, and B. Ravindran. Commit: A scalable approach to mining communication motifs from dynamic networks. In *SIGMOD*, 2015.

[8] P. Holme and J. Saramäki. Temporal networks. *Physics Reports*, 519(3):97–125, 2012.

[9] H. Huang, J. Tang, S. Wu, L. Liu, et al. Mining triadic closure patterns in social networks. In *WWW*, 2014.

[10] A. Z. Jacobs, S. F. Way, J. Ugander, and A. Clauset. Assembling thefacebook: Using heterogeneity to understand online social network assembly. In *Web Science*, 2015.

[11] D. Kondor, M. Pósfai, I. Csabai, and G. Vattay. Do the rich get richer? an empirical analysis of the bitcoin transaction network. *PLOS ONE*, 9(2):e86197, 2014.

[12] G. Kossinets and D. J. Watts. Empirical analysis of an evolving social network. *Science*, 311(5757):88–90, 2006.

[13] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki. Temporal motifs in time-dependent networks. *JSTAT*, 2011(11):P11005, 2011.

[14] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1):458–473, 2008.

[15] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Signed networks in social media. In *CHI*, 2010.

[16] J. Leskovec, D. P. Huttenlocher, and J. M. Kleinberg. Governance in social media: A case study of the wikipedia promotion process. In *ICWSM*, 2010.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *TKDD*, 1(1):2, 2007.

[18] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon. Superfamilies of evolved and designed networks. *Science*, 303(5663):1538–1542, 2004.

[19] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[20] M. E. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[21] P. Panzarasa, T. Opsahl, and K. M. Carley. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *JASIST*, 2009.

[22] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. *arXiv: 1610.09411*, 2016.

[23] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe. A framework for community identification in dynamic social networks. In *KDD*, 2007.

[24] J. Ugander, L. Backstrom, and J. Kleinberg. Subgraph frequencies: Mapping the empirical and extremal geography of large graph collections. In *WWW*, 2013.

[25] A. Vazquez, R. Dobrin, D. Sergi, J.-P. Eckmann, Z. Oltvai, and A.-L. Barabási. The topological relationship between the large-scale attributes and local interaction patterns of complex networks. *PNAS*, 101(52):17940–17945, 2004.

[26] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in Facebook. In *WOSN*, 2009.

[27] S. Wernicke and F. Rasche. Fanmod: a tool for fast network motif detection. *Bioinformatics*, 22(9):1152–1153, 2006.

[28] Y. Wu, C. Zhou, J. Xiao, J. Kurths, and H. J. Schellnhuber. Evidence for a bimodal distribution in human communication. *PNAS*, 107(44):18803–18808, 2010.

[29] Ö. N. Yaveroğlu, N. Malod-Dognin, D. Davis, Z. Levnajic, V. Janjic, R. Karapandza, A. Stojmirovic, and N. Pržulj. Revealing the hidden language of complex networks. *Scientific Reports*, 4, 2014.

[30] Q. Zhao, Y. Tian, Q. He, N. Oliver, R. Jin, and W.-C. Lee. Communication motifs: a tool to characterize social communications. In *CIKM*, 2010.

# ONBRA: Rigorous Estimation of the Temporal Betweenness Centrality in Temporal Networks

Diego Santoro*
Department of Information Engineering
University of Padova
Padova, Italy
diego.santoro@phd.unipd.it

Ilie Sarpe*
Department of Information Engineering
University of Padova
Padova, Italy
sarpeilie@dei.unipd.it

*"Dàme 'n' onbra de vin!"* – A Venetian asking for some wine.

## ABSTRACT

In network analysis, the betweenness centrality of a node informally captures the fraction of shortest paths visiting that node. The computation of the betweenness centrality measure is a fundamental task in the analysis of modern networks, enabling the identification of the most central nodes in such networks. Additionally to being massive, modern networks also contain information about the *time* at which their events occur. Such networks are often called *temporal networks*. The temporal information makes the study of the betweenness centrality in temporal networks (i.e., *temporal betweenness centrality*) much more challenging than in static networks (i.e., networks without temporal information). Moreover, the *exact* computation of the temporal betweenness centrality is often impractical on even moderately-sized networks, given its extremely high computational cost. A natural approach to reduce such computational cost is to obtain high-quality *estimates* of the exact values of the temporal betweenness centrality. In this work we present ONBRA, the first sampling-based approximation algorithm for estimating the temporal betweenness centrality values of the nodes in a temporal network, providing rigorous probabilistic guarantees on the quality of its output. ONBRA is able to compute the estimates of the temporal betweenness centrality values under two different optimality criteria for the shortest paths of the temporal network. In addition, ONBRA outputs high-quality estimates with sharp theoretical guarantees leveraging on the *empirical Bernstein bound*, an advanced concentration inequality. Finally, our experimental evaluation shows that ONBRA significantly reduces the computational resources required by the exact computation of the temporal betweenness centrality on several real world networks, while reporting high-quality estimates with rigorous guarantees.

## CCS CONCEPTS

• **Mathematics of computing** → **Probabilistic algorithms**; • **Theory of computation** → **Graph algorithms analysis**.

*Both authors contributed equally to this research.


*WWW '22, April 25–29, 2022, Virtual Event, Lyon, France*

ACM ISBN 978-1-4503-9096-5/22/04…$15.00
https://doi.org/10.1145/3485447.3512204

## KEYWORDS

Temporal networks, Temporal betweenness centrality, Sampling algorithm, Probabilistic analysis

**ACM Reference Format:**
Diego Santoro and Ilie Sarpe. 2022. ONBRA: Rigorous Estimation of the Temporal Betweenness Centrality in Temporal Networks. In *Proceedings of the ACM Web Conference 2022 (WWW '22), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3485447.3512204

## 1 INTRODUCTION

The study of centrality measures is a fundamental primitive in the analysis of networked datasets [3, 24], and plays a key role in social network analysis [8]. A centrality measure informally captures how important a node is for a given network according to *structural* properties of the network. Central nodes are crucial in many applications such as analyses of co-authorship networks [22, 34], biological networks [18, 33], and ontology summarization [35].

One of the most important centrality measures is the betweenness centrality [10, 11], which informally captures the fraction of *shortest paths* going through a specific node. The betweenness centrality has found applications in many scenarios such as community detection [9], link prediction [1], and network vulnerability analysis [14]. The exact computation of the betweenness centrality of each node of a network is an extremely challenging task on modern networks, both in terms of running time and memory costs. Therefore, sampling algorithms have been proposed to provide provable high-quality approximations of the betweenness centrality values, while remarkably reducing the computational costs [5, 28, 29].

Modern networks, additionally to being large, have also richer information about their edges. In particular, one of the most important and easily accessible information is the *time* at which edges occur. Such networks are often called *temporal networks* [16]. The analysis of temporal networks provides novel insights compared to the insights that would be obtained by the analysis of static networks (i.e., networks without temporal information), as, for example, in the study of subgraph patterns [19, 26], community detection [21], and network clustering [12]. As well as for static networks, the study of the temporal betweenness centrality in temporal networks aims at identifying the nodes that are visited by a high number of *optimal* paths [6, 15]. In temporal networks, the definition of optimal paths has to consider the information about the timing of the edges, making the possible definitions of optimal paths much more richer than in static networks [30].

In this work, a temporal path is valid if it is time respecting, i.e. if all the interactions within the path occur at increasing timestamps (see Figures 1b-1c). We considered two different optimality criteria for temporal paths, chosen for their relevance [15]: (i) shortest temporal path (STP) criterion, a commonly used criterion for which a path is optimal if it uses the minimum number of interactions to connect a given pair of nodes; (ii) restless temporal path (RTP) criterion, for which a path is optimal if, in addition to being shortest, all its consecutive interactions occur at most within a given user-specified time duration parameter $\delta \in \mathbb{R}$ (see Figure 1c). The RTP criterion finds application, for example, in the study of spreading processes over complex networks [25], where information about the timing of consecutive interactions is fundamental. The exact computation of the temporal betweenness centrality under the STP and RTP optimality criteria becomes impractical (both in terms of running time and memory usage) for even moderately-sized networks. Furthermore, as well as for static networks, obtaining a high-quality approximation of the temporal betweenness centrality of a node is often sufficient in many applications. Thus, we propose ONBRA, the *first* algorithm to compute rigOrous estimatioN of temporal Betweenness centRality values in temporAl networks[1], providing sharp guarantees on the quality of its output. As for many data-mining algorithms, ONBRA's output is function of two parameters: $\varepsilon \in (0, 1)$ controlling the estimates' accuracy; and $\eta \in (0, 1)$ controlling the confidence. The algorithmic problems arising from accounting for temporal information are really challenging to deal with compared to the static network scenario, although ONBRA shares a high-level sampling strategy similar to [29]. Finally, we show that in practice our algorithm ONBRA, other than providing high-quality estimates while reducing computational costs, it also enables analyses that cannot be otherwise performed with existing state-of-the-art algorithms. Our main contributions are the following:

- We propose ONBRA, the first sampling-based algorithm that outputs high-quality approximations of the temporal betweenness centrality values of the nodes of a temporal network. ONBRA leverages on an advanced data-dependent and variance-aware concentration inequality to provide sharp probabilistic guarantees on the quality of its estimates.
- We show that ONBRA is able to compute high-quality temporal betweenness estimates for two optimality criteria of the paths, i.e., STP and RTP criteria. In particular, we developed specific algorithms for ONBRA to address the computation of the estimates according to such optimality criteria.
- We perform an extensive experimental evaluation with several goals: (i) under the STP criterion, show that studying the temporal betweenness centrality provides novel insights compared to the static version; (ii) under the STP criterion, show that ONBRA provides high-quality estimates, while significantly reducing the computational costs compared to the state-of-the-art exact algorithm, and that it enables the study of large datasets that cannot practically be analyzed by the existing exact algorithm; (iii) show that ONBRA is able to estimate the temporal betweenness centrality under the RTP optimality criterion by varying $\delta$.

[1]https://vec.wikipedia.org/wiki/Onbra.

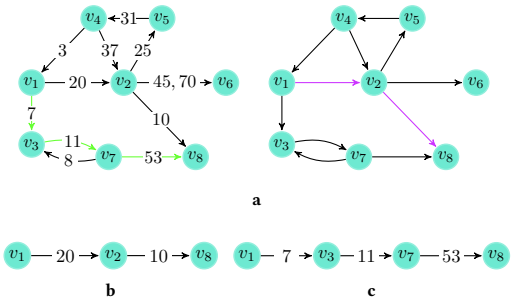



**Figure 1: (1a): (left) a temporal network $T$ with $n = 8$ nodes and $m = 12$ edges, (right) its associated static network $G_T$ obtained from $T$ by removing temporal information. A shortest *temporal* path cannot be identified by a shortest path in the static network: e.g., the shortest paths from node $v_1$ to node $v_8$, respectively coloured in green in $T$ and purple in $G_T$, are different. (1b): A path that is not time respecting. (1c): A time respecting path that is also shortest in $T$. With $\delta \geq 42$ such path is also shortest $\delta$-restless path.**

## 2 PRELIMINARIES

In this section we introduce the fundamental notions needed throughout the development of our work and formalize the problem of approximating the temporal betweenness centrality of the nodes in a temporal network.

We start by introducing temporal networks.

*Definition 2.1.* A *temporal network* $T$ is a pair $T = (V, E)$, where $V$ is a set of $n$ nodes (or vertices), and $E = \{(u, v, t) : u, v \in V, u \neq v, t \in \mathbb{R}^+\}$ is a set of $m$ directed edges[2][3].

Each edge $e = (u, v, t) \in E$ of the network represents an interaction from node $u \in V$ to node $v \in V$ at time $t$, which is the *timestamp* of the edge. Figure 1a (left) provides an example of a temporal network $T$. Next, we define *temporal paths*.

*Definition 2.2.* Given a temporal network $T$, a *temporal path* $\mathsf{P}$ is a sequence $\mathsf{P} = \langle e_1 = (u_1, v_1, t_1), e_2 = (u_2, v_2, t_2), \ldots, e_k = (u_k, v_k, t_k)\rangle$ of $k$ edges of $T$ ordered by increasing timestamps[4], i.e., $t_i < t_{i+1}, i \in \{1, \ldots, k-1\}$, such that the node $v_i$ of edge $e_i$ is equal to the node $u_{i+1}$ of the consecutive edge $e_{i+1}$, i.e., $v_i = u_{i+1}, i \in \{1, \ldots, k-1\}$, and each node $v \in V$ is visited by $\mathsf{P}$ *at most* once.

Given a temporal path $\mathsf{P}$ made of $k$ edges, we define its length as $\ell_{\mathsf{P}} = k$. An example of temporal path $\mathsf{P}$ of length $\ell_{\mathsf{P}} = 3$ is given by Figure 1c. Given a *source* node $s \in V$ and a *destination* node $z \in V, z \neq s$, a *shortest* temporal path between $s$ and $z$ is a temporal path $\mathsf{P}_{s,z}$ of length $\ell_{\mathsf{P}_{s,z}}$ such that in $T$ there is no temporal path $\mathsf{P}'_{s,z}$ connecting $s$ to $z$ of length $\ell_{\mathsf{P}'_{s,z}} < \ell_{\mathsf{P}_{s,z}}$. Given a temporal shortest path $\mathsf{P}_{s,z}$ connecting $s$ and $z$, we define $\texttt{Int}(\mathsf{P}_{s,z}) = \{w \in$

[2]ONBRA can be easily adapted to work on *undirected* temporal networks with minor modifications.

[3]W.l.o.g. we assume the edges $(u_1, v_1, t_1), \ldots, (u_m, v_m, t_m)$ to be sorted by increasing timestamps.

[4]Our work can be easily adapted to deal with non-strict ascending timestamps (i.e., with $\leq$ constraints).

$V| \exists (u,w,t) \vee (w,v,t) \in \mathsf{P}_{s,z}, w \neq s,z\} \subset V$ as the set of nodes *internal* to the path $\mathsf{P}_{s,z}$. Let $\sigma^{sh}_{s,z}$ be the number of shortest temporal paths between nodes $s$ and $z$. Given a node $v \in V$, we denote with $\sigma^{sh}_{s,z}(v)$ the number of shortest temporal paths $\mathsf{P}_{s,z}$ connecting $s$ and $z$ for which $v$ is an internal node, i.e., $\sigma^{sh}_{s,z}(v) = |\{\mathsf{P}_{s,z} | v \in \mathtt{Int}(\mathsf{P}_{s,z})\}|$. Now we introduce the *temporal betweenness centrality* of a node $v \in V$, which intuitively captures the fraction of shortest temporal paths visiting $v$.

*Definition 2.3.* We define the *temporal betweenness centrality* $b(v)$ of a node $v \in V$ as

$$b(v) = \frac{1}{n(n-1)} \sum_{s,z \in V,\ s \neq z} \frac{\sigma^{sh}_{s,z}(v)}{\sigma^{sh}_{s,z}}.$$

Let $B(T) = \{(v, b(v)) : v \in V\}$ be the set of pairs composed of a node $v \in V$ and its temporal betweenness value $b(v)$. Since the exact computation of the set $B(T)$ using state-of-the-art exact algorithms, e.g., [6, 30], is impractical on even moderately-sized temporal networks (see Section 5 for experimental evaluations), in our work we aim at providing high-quality approximations of the temporal betweenness centrality values of all the nodes of the temporal network. That is, we compute the set $\tilde{B}(T) = \{(v, \tilde{b}(v)) : v \in V\}$, where $\tilde{b}(v)$ is an accurate estimate of $b(v)$, controlled by two parameters $\varepsilon, \eta \in (0,1)$, (accuracy and confidence). We want $\tilde{B}(T)$ to be an *absolute $(\epsilon, \eta)$-approximation set* of $B(T)$, as commonly adopted in data-mining algorithms (e.g., in [29]): that is, $\tilde{B}(T)$ is an approximation set such that

$$\mathbb{P}\left[\sup_{v \in V} |\tilde{b}(v) - b(v)| \leq \varepsilon\right] \geq 1 - \eta.$$

Note that in an absolute $(\epsilon, \eta)$-approximation set, for each node $v \in V$, the estimate $\tilde{b}(v)$ of the temporal betweenness value deviates from the actual value $b(v)$ of at most $\varepsilon$, with probability at least $1-\eta$. Finally, let us state the main computational problem addressed in this work.

Problem 1. *Given a temporal network $T$ and two parameters $(\varepsilon, \eta) \in (0,1)^2$, compute the set $\tilde{B}(T)$, i.e., an absolute $(\epsilon, \eta)$-approximation set of $B(T)$.*

## 3 RELATED WORKS

Given the importance of the betweenness centrality for network analysis, many algorithms have been proposed to compute it in different scenarios. In this section we focus on those scenarios most relevant to our work, grouped as follows.

*Approximation Algorithms for Static Networks.* Recently, many algorithms to approximate the betweenness centrality in static networks have been proposed, most of them employ randomized sampling approaches [5, 28, 29]. The existing algorithms differ from each other mainly for the sampling strategy they adopt and for the probabilistic guarantees they offer. Among these works, the one that shares similar ideas to our work is [29] by Riondato and Upfal, where the authors proposed to sample pairs of nodes $(s,z) \in V^2$, compute all the shortest paths from $s$ to $z$, and update the estimates of the betweenness centrality values of the nodes internal to such paths. The authors developed a suite of algorithms to output an $(\varepsilon, \eta)$-approximation set of the set of betweenness centrality values. Their work cannot be easily adapted to temporal networks. In fact, static and temporal paths in general are not related in any way, and the temporal scenario introduces many novel challenges: (i) computing the optimal temporal paths, and (ii) updating the betweenness centrality values. Therefore, our algorithm ONBRA employs the idea of the estimator provided by [29], while using novel algorithms designed for the context of temporal networks. Furthermore, the probabilistic guarantees provided by our algorithm ONBRA leverage on the variance of the estimates, differently from [29] that used bounds based on the Rademacher averages. Our choice to use a variance-aware concentration inequality is motivated by the recent interest in providing sharp guarantees employing the *empirical variance* of the estimates [7, 27].

*Algorithms for Dynamic Networks.* In this setting the algorithm keeps track of the betweenness centrality value of each node for every timestamp $t_1, \ldots, t_m$ observed in the network [13, 20]. Note that this is extremely different from estimating the temporal betweenness centrality values in temporal networks. In the dynamic scenario the paths considered are *not* required to be time respecting. For example, in the dynamic scenario, if we consider the network in Figure 1a (left) at any time $t > 20$, the shortest path from $v_1$ to $v_8$ is the one highlighted in purple in Figure 1a (right). Instead, in the temporal setting such path is not time respecting. We think that it is very challenging to adapt the algorithms for dynamic networks to work in the context of temporal networks, which further motivates us to propose ONBRA.

*Exact Algorithms for Temporal Networks.* Several exact approaches have been proposed in the literature [2, 17, 31]. The algorithm most relevant to our work was presented in [6], where the authors extended the well-known Brandes algorithm [4] to the temporal network scenario considering the STP criterion (among several other criteria). They showed that the time complexity of their algorithm is $O(n^3(t_m - t_1)^2)$, which is often impractical on even moderately-sized networks. Recently, [30] discussed conditions on temporal paths under which the temporal betweenness centrality can be computed in polynomial time, showing a general algorithm running in $O(n^2 m (t_m - t_1)^2)$ even under the RTP criterion, which is again very far from being practical on modern networks.

We conclude by observing that, to the best of our knowledge, no approximation algorithms exist for estimating the temporal betweenness centrality in temporal networks.

## 4 METHOD, ALGORITHM, AND ANALYSIS

In this section we discuss ONBRA, our novel algorithm for computing high-quality approximations of the temporal betweenness centrality values of the nodes of a temporal network. We first discuss the sampling strategy used in ONBRA, then we present the algorithm, and finally we show the theoretical guarantees on the quality of the estimates of ONBRA.

### 4.1 ONBRA - Sampling Strategy

In this section we discuss the sampling strategy adopted by ONBRA that is independent of the optimality criterion of the paths. However, for the sake of presentation, we discuss the sampling strategy for the STP-based temporal betweenness centrality estimation.

**Algorithm 1:** ONBRA.

**Input:** Temporal network $T = (V, E)$, $\eta \in (0, 1)$, $\ell \geq 2$
**Output:** Pair $(\varepsilon', \tilde{B}(T))$ s.t. $\tilde{B}$ is an absolute $(\varepsilon', \eta)$-approximation set of $B(T)$.

1 $\mathcal{D} \leftarrow \{(u, v) \in V \times V, u \neq v\}$
2 $\tilde{B}_{v,:} \leftarrow \vec{0}_\ell, \forall v \in V$
3 **for** $i \leftarrow 1$ **to** $\ell$ **do**
4 $(s, z) \leftarrow$ `uniformRandomSample(`$\mathcal{D}$`)`
5 `SourceDestinationSTPComputation(`$T, s, z$`)`
6 **if** reached$(z)$ **then**
7 `updateSTPEstimates(`$\tilde{B}, i$`)`
8 $\tilde{B}(T) \leftarrow \{(v, 1/\ell \sum_{i=1}^{\ell} \tilde{B}_{v,i}) : v \in V\}$
9 $\varepsilon' \leftarrow \sup_{v \in V} \left\{ \sqrt{\frac{2\mathbf{V}(\tilde{B}_{v,:}) \ln(4n/\eta)}{\ell}} + \frac{7 \ln(4n/\eta)}{3(\ell - 1)} \right\}$
10 **return** $(\varepsilon', \tilde{B}(T))$

ONBRA samples *pairs* of nodes $(s, z)$ and computes all the shortest temporal paths from $s$ to $z$. More formally, let $\mathcal{D} = \{(u, v) \in V^2 : u \neq v\}$, and $\ell \in \mathbb{N}, \ell \geq 2$ be a user-specified parameter. ONBRA first collects $\ell$ pairs of nodes $(s_i, z_i)_i, i = 1, \ldots, \ell$, sampled uniformly at random from $\mathcal{D}$. Next, for each pair $(s, z)$ it computes $\mathcal{P}_{s,z} = \{\mathsf{P}_{s,z} : \mathsf{P}_{s,z} \text{ is shortest}\}$, i.e., the set of shortest temporal paths from $s$ to $z$. Then, for each node $v \in V$ s.t. $\exists \mathsf{P}_{s,z} \in \mathcal{P}_{s,z}$ with $v \in \mathtt{int}(\mathsf{P}_{s,z})$, i.e., for each node $v$ that is internal to a shortest temporal path of $\mathcal{P}_{s,z}$, ONBRA computes the estimate $\tilde{b}'(v) = \sigma^{sh}_{s,z}(v)/\sigma^{sh}_{s,z}$, which is an unbiased estimator of the temporal betweenness centrality value $b(v)$ (i.e., $\mathbb{E}[\tilde{b}'(w)] = b(v)$, see Lemma A.1 in Appendix A). Finally, after processing the $\ell$ pairs of nodes randomly selected, ONBRA computes for each node $v \in V$ the (unbiased) estimate $\tilde{b}(v)$ of the actual temporal betweenness centrality $b(v)$ by averaging $\tilde{b}'(v)$ over the $\ell$ sampling steps: $\tilde{b}(v) = 1/\ell \sum_{i=1}^{\ell} \tilde{b}'(v)_i$, where $\tilde{b}'(v)_i$ is the estimate of $b(v)$ obtained by analyzing the $i$-th sample, $i \in [1, \ell]$. We will discuss the theoretical guarantees on the quality of $\tilde{b}(v)'s$ in Section 4.4.

## 4.2 Algorithm Description

*Sampling Algorithm:* ONBRA. ONBRA is presented in Algorithm 1. In line 1 we first initialize the set $\mathcal{D}$ of objects to be sampled, where each object is a pair of distinct nodes from $V$. Next, in line 2 we initialize the matrix $\tilde{B}$ of size $|V| \cdot \ell$ to store the estimates of ONBRA for each node at the various iterations, needed to compute their empirical variance and the final estimates. Then we start the main loop (line 3) that will iterate $\ell$ times. In such loop we first select a pair $(s, z)$ sampled uniformly at randomly from $\mathcal{D}$ (line 4). We then compute all the shortest temporal paths from $s$ to $z$ by executing Algorithm 2 (line 5), which is described in detail later in this section. Such algorithm computes all the shortest temporal paths from $s$ and $z$ adopting some pruning criteria to speed-up the computation. If at least one STP between $s$ and $z$ exists (line 6), then for each node $v \in V$ internal to a path in $\mathcal{P}_{s,z}$ we update the corresponding estimate to the current iteration by computing $\tilde{b}'(v)_i$ using Algorithm 3 (line 7). While in static networks this step can be done with a simple recursive formula [29], in our scenario we need a specific algorithm to deal with the more challenging fact that a node may appear at different distances from a given source across different shortest temporal paths. We will discuss in detail such algorithm later in this section. At the end of the $\ell$ iterations of the main loop, ONBRA computes: (i) the set $\tilde{B}(T)$ of unbiased estimates (line 8); (ii) and a tight bound $\varepsilon'$ on $\sup_{v \in V} |\tilde{b}(v) - b(v)|$, which leverages the empirical variance $\mathbf{V}(\tilde{B}_{v,:})$ of the estimates (line 9). We observe that $\varepsilon'$ is such that the set $\tilde{B}(T)$ is an absolute $(\varepsilon', \eta)$-approximation set of $B(T)$. We discuss the computation of such bound in Section 4.4. Finally, ONBRA returns $(\varepsilon', \tilde{B}(T))$.

**Algorithm 2:** Source-Destination STP computation.

**Input:** $T = (V, E)$, source node $s$, destination node $z$

1 **for** $v \in V$ **do**
2 $\mathrm{dist}_v \leftarrow -1$; $\sigma_v \leftarrow 0$
3 **for** $(u, v, t) \in E$ **do**
4 $\sigma_{v,t} \leftarrow 0$; $P_{v,t} \leftarrow \emptyset$; $\mathrm{dist}_{v,t} \leftarrow -1$
5 $\mathrm{dist}_s \leftarrow 0$; $\mathrm{dist}_{s,0} \leftarrow 0$
6 $\sigma_s \leftarrow 1$; $\sigma_{s,0} \leftarrow 1$; $\mathrm{d}_z^{\min} \leftarrow \infty$
7 $Q \leftarrow$ empty queue; $Q$.enqueue$((s, 0))$
8 **while** !$Q$.empty() **do**
9 $(v, t) \leftarrow Q$.dequeue()
10 **if** $(\mathrm{dist}_{v,t} < \mathrm{d}_z^{\min})$ **then**
11 **for** $(w, t') \in \mathcal{N}^+(v, t)$, **do**
12 **if** $\mathrm{dist}_{w,t'} = -1$ **then**
13 $\mathrm{dist}_{w,t'} \leftarrow \mathrm{dist}_{v,t} + 1$
14 **if** $\mathrm{dist}_w = -1$ **then**
15 $\mathrm{dist}_w \leftarrow \mathrm{dist}_{v,t} + 1$
16 **if** $w = z$ **then**
17 $\mathrm{d}_z^{\min} \leftarrow \mathrm{dist}_w$
18 $Q$.enqueue$((w, t'))$
19 **if** $\mathrm{dist}_{w,t'} = \mathrm{dist}_{v,t} + 1$ **then**
20 $\sigma_{w,t'} \leftarrow \sigma_{w,t'} + \sigma_{v,t}$
21 $P_{w,t'} \leftarrow P_{w,t'} \cup \{(v, t)\}$
22 **if** $\mathrm{dist}_{w,t'} = \mathrm{dist}_w$ **then**
23 $\sigma_w \leftarrow \sigma_w + \sigma_{v,t}$

*Subroutines.* We now describe the subroutines employed in Algorithm 1 focusing on the STP criterion. Then, in Section 4.3, we discuss how to deal with the RTP criterion.

*Source Destination Shortest Paths Computation.* We start by introducing some definitions needed through this section. First, we say that a pair $(v, t) \in V \times \{t_1, \ldots, t_m\}$ is a *vertex appearance* (VA) if $\exists (u, v, t) \in E$. Next, given a VA $(v, t)$ we say that a VA $(w, t')$ is a *predecessor* of $(v, t)$ if $\exists (w, v, t) \in E, t' < t$. Finally, given a VA $(v, t)$ we define its set of *out-neighbouring VAs* as $\mathcal{N}^+(v, t) = \{(w, t') : \exists (v, w, t') \in E, t < t'\}$.

We now describe Algorithm 2 that computes the shortest temporal paths between a source node $s$ and a destination node $z$ (invoked in ONBRA at line 5). Such computation is optimized to prune the search space once found the destination $z$. The algorithm initializes

the data structures needed to keep track of the shortest temporal paths that, starting from $s$, reach a node in $V$, i.e., the arrays $\text{dist}[\cdot]$ and $\sigma[\cdot]$ that contain for each node $v \in V$, respectively, the minimum distance to reach $v$ and the number of shortest temporal paths reaching $v$ (line 2). In line 4 we initialize $\text{dist}[\cdot,\cdot]$ that keeps track of the minimum distance of a VA from the source $s$, $\sigma[\cdot,\cdot]$ that maintains the number of shortest temporal paths reaching a VA from $s$, and $P$ keeping the set of predecessors of a VA across the shortest temporal paths explored. After initializing the values of the data structures for the source $s$ and $\text{d}_z^{\min}$ keeping the length of the minimum distance to reach $z$ (lines 5-6), we initialize the queue $Q$ that keeps the VAs to be visited in a BFS fashion in line 7 (observe that, since the temporal paths need to be time-respecting, all the paths need to account for the time at which each node is visited). Next, the algorithm explores the network in a BFS order (line 8), extracting a VA $(v,t)$ from the queue, which corresponds to a node and the time at which such node is visited, and processing it by collecting its set $\mathcal{N}^+(v,t)$ of out-neighbouring VAs (lines 9-11). If a VA $(w,t')$ was not already explored (i.e., it holds $\text{dist}_{w,t'} = -1$), then we update the minimum distance $\text{dist}_{w,t'}$ to reach $w$ at time $t'$, the minimum distance $\text{dist}_w$ of the vertex $w$ if it was not already visited, and, if $w$ is the destination node $z$, we update $\text{d}_z^{\min}$ (lines 12-17). Observe that the distance $\text{d}_z^{\min}$ to reach $z$ is used as a *pruning criterion* in line 10 (clearly, if a VA appears at a distance greater than $\text{d}_z^{\min}$ then it cannot be on a shortest temporal path from $s$ to $z$). After updating the VAs to be visited by inserting them in $Q$ (line 18), if the current temporal path is shortest for the VA $(w,t')$ analyzed, we update the number $\sigma_{w,t'}$ of shortest temporal paths leading to it, its set $P_{w,t'}$ of predecessors, and the number $\sigma_w$ of shortest temporal paths reaching the node $w$ (lines 19-23).

*Update Estimates: STP criterion.* Now we describe Algorithm 3, which updates the temporal betweenness estimates of each node internal to a path in $\mathcal{P}_{s,z}$ already computed. With Algorithm 2 we computed for each VA $(w,t)$ the number $\sigma_{w,t}$ of shortest temporal paths from $s$ reaching $(w,t)$. Now, in Algorithm 3 we need to combine such counts to compute the total number of shortest temporal paths leading to each VA $(w,t)$ appearing in a path in $\mathcal{P}_{s,z}$, allowing us to compute the estimate of ONBRA for each node $w$.

At the end of Algorithm 2 there are in total $|\mathcal{P}_{s,z}|$ shortest temporal paths reaching $z$ from $s$. Now we need to compute, for each node $w$ internal to a path in $\mathcal{P}_{s,z}$ and for each VA $(w,t)$, the number $\sigma^z_{w,t}$ of shortest temporal paths leading from $w$ to $z$ at a time greater that $t$. Then, the fraction of paths containing the node $w$ is computed with a simple formula, i.e., $\sum_t \sigma^z_{w,t} \cdot \sigma_{w,t}/\sigma_z$, where $\sigma_z = |\mathcal{P}_{s,z}|$. The whole procedure is described in Algorithm 3. We start by initializing $\sigma^z_{v,t}$ that stores for each VA $(v,t)$ the number of shortest temporal paths reaching $z$ at a time greater than $t$ starting from $v$, and a boolean matrix $M$ that keeps track for each VA if it has already been considered (line 2). In line 3 we initialize a queue $R$ that will be used to explore the VAs appearing along the paths in $\mathcal{P}_{s,z}$ in reverse order of distance from $s$ starting from the destination node $z$. Then we initialize $\sigma^z_{w,t'}$ for each VA reaching $z$ at a given time $t'$ (line 6), and we insert each VA in the queue only one time (line 8). The algorithm then starts its main loop exploring the VAs in decreasing order of distance starting from $z$ (line 9). We take the VA $(w,t)$ to be explored in line 10. If $w$ differs from $s$ (i.e., $w$ is an internal node), then we update its temporal betweenness estimate by adding $\sigma^z_{w,t} \cdot \sigma_{w,t}/\sigma_z$ (line 12). As we did in the initialization step, then we process each predecessor $(w',t')$ of $(w,t)$ across the paths in $\mathcal{P}_{s,z}$ (line 13), update the count $\sigma^z_{w',t'}$ of the paths from the predecessor to $z$ by summing the number $\sigma^z_{w,t}$ of paths passing through $(w,t)$ and reaching $z$ (line 14), and we enqueue the predecessor $(w',t')$ only if it was not already considered (lines 15-16). So, the algorithm terminates by having properly computed for each node $v \in V, v \neq s, z$ the estimate $\tilde{b}'(v)_i$ for each iteration $i \in [1,\ell]$.

**Algorithm 3:** Update betweenness estimates - STP.

```
   Input: B̃, i.
1  for (u, v, t) ∈ E do
2      σ^z_{v,t} ← 0; M_{v,t} ← False
3  R ← empty queue;
4  foreach t : (σ_{z,t} > 0) do
5      for (w, t') ∈ P_{z,t} do
6          σ^z_{w,t'} ← σ^z_{w,t'} + 1
7          if !M_{w,t'} then
8              R.enqueue((w, t')); M_{w,t'} ← True
9  while !R.empty() do
10     (w, t) ← R.dequeue()
11     if w ≠ s then
12         B̃_{w,i} ← B̃_{w,i} + σ^z_{w,t} · σ_{w,t}/σ_z
13         for (w', t') ∈ P_{w,t} do
14             σ^z_{w',t'} ← σ^z_{w',t'} + σ^z_{w,t}
15             if !M_{w',t'} then
16                 R.enqueue((w', t')); M_{w',t'} ← True
```

## 4.3 Restless Temporal Betweenness

In this section we present the algorithms that are used in ONBRA when considering the RTP criterion for the optimal paths to compute the temporal betweenness centrality values.

Recall that, in such scenario, a temporal path $\mathsf{P} = \langle e_1 = (u_1, v_1, t_1), e_2 = (u_2, v_2, t_2), \ldots, e_k = (u_k, v_k, t_k)\rangle$ is considered optimal if and only if P, additionally to being *shortest*, is such that, given $\delta \in \mathbb{R}^+$, it holds $t_{i+1} \leq t_i + \delta$ for $i = 1, \ldots, k-1$. Considering the RTP criteria, we need to relax the definition of shortest temporal paths and, instead, consider *shortest temporal walks*. Intuitively, a walk is a path where we drop the constraint that a node must be visited at most once. We provide an intuition of why we need such requirement in Figure 2. Given $\delta \in \mathbb{R}^+$, we refer to a shortest temporal walk as *shortest $\delta$-restless temporal walk*.

In order to properly work under the RTP criteria, ONBRA needs novel algorithms to compute the optimal walks and update the betweenness estimates. Note that to compute the shortest $\delta$-restless temporal walks we can use Algorithm 2 provided that we add the condition $t' - t \leq \delta$ in line 11.

More interestingly, the biggest computational problem arises when updating the temporal betweenness values of the various nodes on the optimal walks. Note that, to do so, we cannot use

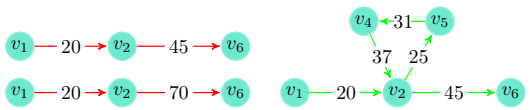



**Figure 2: Considering the temporal network in Figure 1 and $\delta = 10$, the paths from node $v_1$ to node $v_6$ on the left are not shortest $\delta$-restless since both violate the timing constraint (i.e., $45 - 20, 70 - 20 > \delta$). Instead, the walk on the right is shortest and meets the timing constraint with $\delta = 10$: so, it is a shortest $\delta$-restless walk.**

Algorithm 3 because it does not account for cycles (i.e, when vertices appear multiple times across a walk). We therefore introduce Algorithm 4 (pseudocode in Appendix B) that works in the presence of cycles. The main intuition behind Algorithm 4 is that we need to recreate backwards all the optimal walks obtained through the RTP version of Algorithm 2. For each walk we will maintain a set that keeps track of the nodes already visited up to the current point of the exploration of the walk, updating a node's estimate if and only if we see such node for the first time. This is based on the simple observation that a cycle cannot alter the value of the betweenness centrality of a node on a fixed walk, allowing us to account only once for the node's appearance along the walk.

We now describe Algorithm 4 by discussing its differences with Algorithm 3. In line 2, instead of maintaining a matrix keeping track of the presence of a VA in the queue, we now initialize a matrix $\mathsf{u}[\cdot,\cdot]$ that keeps the number of times a VA is in the queue. The queue, initialized in line 3, keeps elements of the form $\langle \cdot, \cdot \rangle$, where the first entry is a VA to be explored and the second entry is the set of nodes already visited backwards along the walk leading to such vertex appearance. While visiting backwards each walk, we check if the nodes are visited for the first time on such walk: if so, we update the betweenness values by accounting for the number of times we will visit such VA across other walks (lines 10-11). Next, we update the set of nodes visited (line 12). Finally, we update the count $\sigma^z_{w',t'}$ of the walks leading from the predecessor $(w', t')$ of the current VA $(w, t)$ to $z$ (line 14), the number $\mathsf{u}_{w',t'}$ of times such predecessor will be visited (line 15), and enqueue the predecessor $(w', t')$ to be explored, together with the additional information of the set $\mathsf{S}'$ of nodes explored up to that point.

To conclude, note that Algorithm 4 is more expensive than Algorithm 3 since it recreates all the optimal walks, while Algorithm 3 avoids such step given the absence of cycles.

### 4.4 ONBRA - Theoretical Guarantees

In order to address Problem 1, ONBRA bounds the deviation between the estimates $\tilde{b}(v)$ and the actual values $b(v)$, for every node $v \in V$. To do so, we leverage on the so called *empirical Bernstein bound*, which we adapted to ONBRA.

Given a node $v \in V$, let $\tilde{B}_{v,:} = (\tilde{b}'(v)_1, \tilde{b}'(v)_2, \ldots, \tilde{b}'(v)_\ell)$, where $\tilde{b}'(v)_i$ is the estimate of $b(v)$ by analysing the $i$-th sample, $i \in \{1, \ldots, \ell\}$. Let $\mathbf{V}(\tilde{B}_{v,:})$ be the *empirical variance* of $\tilde{B}_{v,:}$:

$$\mathbf{V}(\tilde{B}_{v,:}) = \frac{1}{\ell(\ell-1)} \sum_{1 \le < i < j \le \ell} (\tilde{b}'(v)_i - \tilde{b}'(v)_j)^2.$$

We use the *empirical Bernstein bound* to limit the deviation between $\tilde{b}(v)$'s and $b(v)$'s, which represents Corollary 5 of [23] adapted to our framework, since Corollary 5 of [23] is formulated for generic random variables taking values in $[0, 1]$ and for an arbitrary set of functions.

Theorem 4.1 (Corollary 5, [23]). *Let $\ell \ge 2$ be the number of samples, and $\eta \in (0, 1)$ be the confidence parameter. Let $\tilde{b}'(v)_i$ be the estimate of $b(v)$ by analysing the $i$-th sample, $i \in \{1, \ldots, \ell\}$ and $v \in V$. Let $\tilde{B}_{v,:} = (\tilde{b}'(v)_1, \tilde{b}'(v)_2, \ldots, \tilde{b}'(v)_\ell)$, and $\mathbf{V}(\tilde{B}_{v,:})$ be its empirical variance. With probability at least $1 - \eta$, and for every node $v \in V$, we have that*

$$|\tilde{b}(v) - b(v)| \le \sqrt{\frac{2\mathbf{V}(\tilde{B}_{v,:}) \ln(4n/\eta)}{\ell}} + \frac{7 \ln(4n/\eta)}{3(\ell - 1)}.$$

The right hand side of the inequality of the previous theorem differs from Corollary 5 of [23] by a factor of 2 in the arguments of the natural logarithms, since in [23] the bound is not stated in the symmetric form reported in Theorem 4.1. Finally, the result about the guarantees on the quality of the estimates provided by ONBRA follows.

Corollary 4.2. *Given a temporal network $T$, the pair $(\varepsilon', \tilde{B}(T))$ in output from* ONBRA *is such that, with probability $> 1 - \eta$, it holds that $\tilde{B}(T)$ is an absolute $(\varepsilon', \eta)$-approximation set of $B(T)$.*

Observe that Corollary 4.2 is independent of the structure of the optimal paths considered by ONBRA, therefore such guarantees hold for both the criteria considered in our work.

## 5 EXPERIMENTAL EVALUATION

In this section we present our experimental evaluation that has the following goals: (i) motivate the study of the temporal betweenness centrality by showing two real world temporal networks on which the temporal betweenness provides novel insights compared to the static betweenness computed on their associated static networks; (ii) assess, considering the STP criterion, the accuracy of the ONBRA's estimates, and the benefit of using ONBRA instead of the state-of-the-art exact approach [6], both in terms of running time and memory usage; (iii) finally, show how ONBRA can be used on a real world temporal network to analyze the RTP-based betweenness centrality values.

### 5.1 Setup

We implemented ONBRA in C++20 and compiled it using `gcc 9`. The code is freely available[5]. All the experiments were performed sequentially on a 72 core Intel Xeon Gold 5520 @ 2.2GHz machine with 1008GB of RAM available. The real world datasets we used are described in Table 1, which are mostly social or message networks from different domains. Such datasets are publicly available online[6]. For detailed descriptions of such datasets we refer to the links reported and [26]. To obtain the FBWall dataset we cut the last 200K edges from the original dataset [32], which has more than 800K edges. Such cut is done to allow the exact algorithm to complete its execution without exceeding the available memory.

[5] https://github.com/iliesarpe/ONBRA.
[6] http://www.sociopatterns.org/ and https://snap.stanford.edu/temporal-motifs/data.html.

**Table 1: Datasets used and their statistics.**

| Name | $n$ | $m$ | Granularity | Timespan |
|---|---|---|---|---|
| HighSchool2012 (HS) | 180 | 45K | 20 sec | 7 (days) |
| CollegeMsg | 1.9K | 59.8K | 1 sec | 193 (days) |
| EmailEu | 986 | 332K | 1 sec | 803 (days) |
| FBWall (FB) | 35.9K | 199.8K | 1 sec | 100 (days) |
| Sms | 44K | 544.8K | 1 sec | 338 (days) |
| Mathoverflow | 24.8K | 390K | 1 sec | 6.4 (years) |
| Askubuntu | 157K | 727K | 1 sec | 7.2 (years) |
| Superuser | 192K | 1.1M | 1 sec | 7.6 (years) |

## 5.2 Temporal vs Static Betweenness

In this section we assess that the temporal betweenness centrality of the nodes of a temporal network provides novel insights compared to its static version. To do so, we computed for two datasets, from different domains, the exact ranking of the various nodes according to their betweenness values. The goal of this experiment is to compare the two rankings (i.e., temporal and static) and understand if the relative orderings are preserved, i.e., verify if the most central nodes in the static network are also the most central nodes in the temporal network. To this end, given a temporal network $T = (V, E)$, let $G_T = (V, \{(u, v) : \exists (u, v, t) \in E\})$ be its associated static network. We used the following two real world networks: (i) HS, that is a temporal network representing a face-to-face interaction network among students; (ii) and FB, that is a Facebook user-activity network [32] (see Table 1 for further details).

We first computed the exact temporal and static betweenness values of the different nodes of the two networks. Then, we ranked the nodes by descending betweenness values. We now discuss how the top-$k$ ranked nodes vary from temporal to static on the two networks. We report in Table 3 (in Appendix C) the Jaccard similarity between the sets containing the top-$k$ nodes of the static and temporal networks. On HS, for $k = 25$, only 11 nodes are top ranked in both the rankings, which means that less than half of the top-25 nodes are central if only the static information is considered. The value of the intersection increases to 36 for $k = 50$, since the network has only 180 nodes. More interestingly, also on the Facebook network only few temporally central nodes can be detected by considering only static information: only 9 over the top-25 nodes and 15 over the top-50 nodes. In order to better visualize the top-$k$ ranked nodes, we show their betweenness values in Figure 3a: note that there are many top-$k$ temporally ranked nodes having small static betweenness values, and vice versa.

These experiments show the importance of studying the temporal betweenness centrality, which provides novel insights compared to the static version.

## 5.3 Accuracy and Resources of ONBRA

In this section we first assess the accuracy of the estimates $\tilde{B}(T)$ provided by ONBRA considering only the STP criterion, since for the RTP criterion no implemented exact algorithm exists. Then, we show the reduction of computational resources induced by ONBRA compared to the exact algorithm in [6].

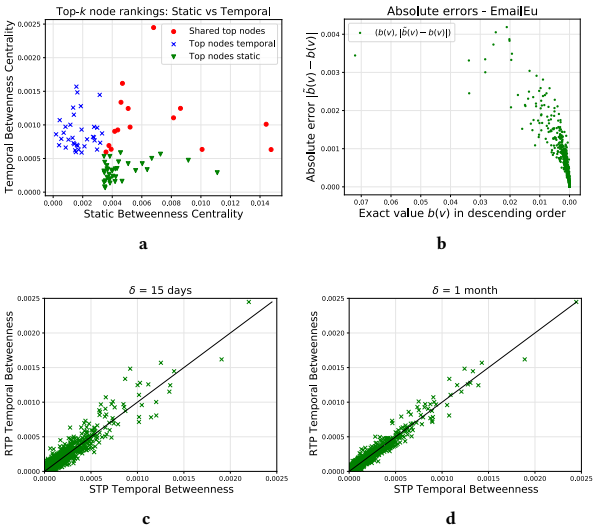



**Figure 3: (3a): static and temporal betweenness values of the top-50 ranked nodes of the dataset FB; (3b): for dataset `EmailEu`, the deviations (or absolute errors) $|\tilde{b}(v) - b(v)|$ between the estimates $\tilde{b}(v)$ and the actual values $b(v)$ of the temporal betweenness centrality, for decreasing order of $b(v)$; (3c,3d): comparison between the temporal betweenness values based on STP and RTP, for $\delta$=15 days (left) and $\delta$=1 month (right).**

To assess ONBRA's accuracy and its computational cost, we used four datasets, i.e., `CollegeMsg`, `EmailEu`, `Mathoverflow`, and `FBWall`. We first executed the exact algorithm, and then we fix $\eta = 0.1$ and $\ell$ properly for ONBRA to run within a fraction of the time required by the exact algorithm. The results we now present, which are described in detail in Table 2, are all averaged over 10 runs (except for the RAM peak, which is measured over one single execution of the algorithms).

Remarkably, even using less than 1% of the overall pairs of nodes as sample size, ONBRA is able to estimate the temporal betweenness centrality values with very small average deviations between $4 \cdot 10^{-6}$ and $5 \cdot 10^{-4}$, while obtaining a significant running time speed-up between $\approx$1.5× and $\approx$4× with respect to the exact algorithm [6]. Additionally, the amount of RAM memory used by ONBRA is significantly smaller than the exact algorithm in [6]: e.g., on the `Mathoverflow` dataset ONBRA requires only 6.8 GB of RAM peak, which is 147× less than the 1004.3 GB required by the exact state-of-the-art algorithm [6]. Furthermore, in all the experiments we found that the maximum deviation is distant at most one order of magnitude from the theoretical upper bound $\varepsilon'$ guaranteed by Corollary 4.2. Surprisingly, for two datasets (`EmailEu` and `Mathoverflow`) the maximum deviation and the upper bound $\varepsilon'$ are even of the same order of magnitude. Therefore we can conclude that the guarantees provided by Corollary 4.2 are often very sharp. In addition, ONBRA's accuracy is demonstrated by the fact that the deviation between the actual temporal betweenness centrality

**Table 2: For each dataset, the average and maximum deviation between the estimate $\tilde{b}(v)$ and the actual temporal betweenness value $b(v)$ over all nodes $v$ and 10 runs, respectively *Avg. Error* and $\sup_{v \in V} |b(v) - \tilde{b}(v)|$, the theoretical upper bound $\varepsilon'$, the *Sample rate* (%) of pairs of nodes we sampled, the running time $t_{EXC}$ and peak RAM memory $MEM_{EXC}$ required by the exact approach [6], the running time $t_{\text{ONBRA}}$ and peak RAM memory $MEM_{\text{ONBRA}}$ required by ONBRA. The symbol ✗ denotes that the exact computation of [6] is not able to conclude on our machine.**

| Dataset | Avg. Error | $\sup_{v \in V} \lvert b(v) - \tilde{b}(v) \rvert$ | $\varepsilon'$ | Sample rate (%) | $t_{\text{EXC}}$ (sec) | $t_{\text{ONBRA}}$ (sec) | $\text{MEM}_{\text{EXC}}$ (GB) | $\text{MEM}_{\text{ONBRA}}$ (GB) |
|---|---|---|---|---|---|---|---|---|
| CollegeMsg | $1.74 \cdot 10^{-4}$ | $6.38 \cdot 10^{-3}$ | $2.27 \cdot 10^{-2}$ | 0.083 | 231 | **148** | 12.0 | **0.13** |
| EmailEu | $4.69 \cdot 10^{-4}$ | $1.35 \cdot 10^{-2}$ | $6.15 \cdot 10^{-2}$ | 0.093 | 7211 | **1808** | 23.9 | **2.1** |
| Mathoverflow | $6.35 \cdot 10^{-6}$ | $2.1 \cdot 10^{-3}$ | $5.38 \cdot 10^{-3}$ | 0.005 | 79492 | **36983** | 1004.3 | **6.8** |
| FBWall | $4.25 \cdot 10^{-6}$ | $5.89 \cdot 10^{-4}$ | $2.13 \cdot 10^{-3}$ | 0.003 | 11489 | **3145** | 738.0 | **11.1** |
| Askubuntu | ✗ | ✗ | $6.92 \cdot 10^{-3}$ | 0.00006 | ✗ | **35585** | >1008 | **20.3** |
| Sms | ✗ | ✗ | $1.54 \cdot 10^{-3}$ | 0.00231 | ✗ | **13020** | >1008 | **16.2** |
| Superuser | ✗ | ✗ | $1.02 \cdot 10^{-2}$ | 0.00003 | ✗ | **41856** | >1008 | **16.7** |

value of a node and its estimate obtained using ONBRA is about one order of magnitude less than the actual value, as we show in Figure 3b and Figure 4 (in Appendix C).

Finally, we show in Table 2 that on the large datasets Askubuntu, Sms, and Superuser the exact algorithm [6] is not able to conclude the computation on our machine (denoted with ✗) since it requires more than 1008GB of RAM. Instead, ONBRA provides estimates of the temporal betweenness centrality values in less than 42K (sec) and 21 GB of RAM memory.

To conclude, ONBRA is able to estimate the temporal betweenness centrality with high accuracy providing rigorous and sharp guarantees, while significantly reducing the computational resources required by the exact algorithm in [6].

### 5.4 ONBRA on RTP-based Betweenness

In this section we discuss how ONBRA can be used to analyze real world networks by estimating the centrality values of the nodes for the temporal betweenness under the RTP criterion.

We used the FB network, on which we computed a tight approximation of the temporal betweenness values ($\varepsilon' < 10^{-4}$) of the nodes for different values of $\delta$, i.e., $\delta$=1 day, $\delta$=15 days, and $\delta$=1 month. For $\delta$=1 day, we found only 4 nodes with temporal betweenness value different from 0, which is surprising since it highlights that the information spreading across wall posts through RTPs in 2008 on Facebook required more than 1 day of time between consecutive interactions (i.e., slow spreading). We present the results for the other values of $\delta$ in Figures 3c and 3d, comparing them to the (exact) STP-based betweenness. Interestingly, 15 days are still not sufficient to capture most of the betweenness values based on STPs of the different nodes, while with $\delta$=1 month the betweenness values are much closer to the STP-based values. While this behaviour is to be expected with increasing $\delta$, finding such values of $\delta$ helps to better characterize the dynamics over the network.

To conclude, ONBRA also enables novel analyses that cannot otherwise be performed with existing tools.

## 6 DISCUSSION

In this work we presented ONBRA, the first algorithm that provides high-quality approximations of the temporal betweenness centrality values of the nodes in a temporal network, with rigorous probabilistic guarantees. ONBRA works under two different optimality criteria for the paths on which the temporal betweenness centrality is defined: shortest and restless temporal paths (STP, RTP) criteria. To the best of our knowledge, ONBRA is the first algorithm enabling a practical computation under the RTP criteria. Our experimental evaluation shows that ONBRA provides high-quality estimates with tight guarantees, while remarkably reducing the computational costs compared to the state-of-the-art in [6], enabling analyses that would not otherwise be possible to perform.

Finally, several interesting directions could be explored in the future, such as dealing with different optimality criteria for the paths, and employing sharper concentration inequalities to provide tighter guarantees on the quality of the estimates.

## ACKNOWLEDGMENTS

Part of this work was supported by the Italian Ministry of Education, University and Research (MIUR), under PRIN Project n. 20174LF3T8 "AHeAD" (efficient Algorithms for HArnessing networked Data) and the initiative "Departments of Excellence" (Law 232/2016), and by University of Padova under project "SID 2020: RATED-X".

## REFERENCES

[1] Iftikhar Ahmad, Muhammad Usman Akhtar, Salma Noor, and Ambreen Shahnaz. 2020. Missing Link Prediction using Common Neighbor and Centrality based Parameterized Algorithm. 10, 1 (jan 2020). https://doi.org/10.1038/s41598-019-57304-y

[2] Ahmad Alsayed and Desmond J. Higham. 2015. Betweenness in time dependent networks. 72 (mar 2015), 35–48. https://doi.org/10.1016/j.chaos.2014.12.009

[3] Stephen P. Borgatti and Martin G. Everett. 2006. A Graph-theoretic perspective on centrality. *Soc. Networks* 28, 4 (2006), 466–484. https://doi.org/10.1016/j.socnet.2005.11.005

[4] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. 25, 2 (jun 2001), 163–177. https://doi.org/10.1080/0022250x.2001.9990249

[5] Ulrik Brandes and Christian Pich. 2007. Centrality Estimation in Large Networks. *Int. J. Bifurc. Chaos* 17, 7 (2007), 2303–2318. https://doi.org/10.1142/S0218127407018403

[6] Sebastian Buß, Hendrik Molter, Rolf Niedermeier, and Maciej Rymar. 2020. Algorithmic Aspects of Temporal Betweenness. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 2084–2092. https://doi.org/10.1145/3394486.3403259

[7] Cyrus Cousins, Chloe Wohlgemuth, and Matteo Riondato. 2021. Bavarian: Betweenness Centrality Approximation with Variance-Aware Rademacher Averages. In *KDD '21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, Singapore, August 14-18, 2021*, Feida Zhu, Beng Chin

Ooi, and Chunyan Miao (Eds.). ACM, 196–206. https://doi.org/10.1145/3447548.3467354
[8] Kousik Das, Sovan Samanta, and Madhumangal Pal. 2018. Study on centrality measures in social networks: a survey. *Soc. Netw. Anal. Min.* 8, 1 (2018), 13. https://doi.org/10.1007/s13278-018-0493-2
[9] Santo Fortunato. 2010. Community detection in graphs. 486, 3-5 (feb 2010), 75–174. https://doi.org/10.1016/j.physrep.2009.11.002
[10] Linton C. Freeman. 1977. A Set of Measures of Centrality Based on Betweenness. 40, 1 (mar 1977), 35. https://doi.org/10.2307/3033543
[11] Linton C. Freeman. 1978. Centrality in social networks conceptual clarification. 1, 3 (jan 1978), 215–239. https://doi.org/10.1016/0378-8733(78)90021-7
[12] Dongqi Fu, Dawei Zhou, and Jingrui He. 2020. Local Motif Clustering on Time-Evolving Graphs. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 390–400. https://doi.org/10.1145/3394486.3403081
[13] Kathrin Hanauer, Monika Henzinger, and Christian Schulz. 2021. Recent Advances in Fully Dynamic Graph Algorithms. *CoRR* abs/2102.11169 (2021). arXiv:2102.11169 https://arxiv.org/abs/2102.11169
[14] Petter Holme, Beom Jun Kim, Chang No Yoon, and Seung Kee Han. 2002. Attack vulnerability of complex networks. 65, 5 (may 2002), 056109. https://doi.org/10.1103/physreve.65.056109
[15] Petter Holme and Jari Saramäki. 2012. Temporal networks. 519, 3 (oct 2012), 97–125. https://doi.org/10.1016/j.physrep.2012.03.001
[16] Petter Holme and Jari Saramäki (Eds.). 2019. *Temporal Network Theory*. Springer International Publishing. https://doi.org/10.1007/978-3-030-23495-9
[17] Hyoungshick Kim and Ross Anderson. 2012. Temporal node centrality in complex networks. 85, 2 (feb 2012), 026107. https://doi.org/10.1103/physreve.85.026107
[18] Dirk Koschützki and Falk Schreiber. 2008. Centrality Analysis Methods for Biological Networks and Their Application to Gene Regulatory Networks. 2 (jan 2008), GRSB.S702. https://doi.org/10.4137/grsb.s702
[19] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *CoRR* abs/1107.5646 (2011). arXiv:1107.5646 http://arxiv.org/abs/1107.5646
[20] Min-Joong Lee, Jungmin Lee, Jaimie Yejean Park, Ryan Hyun Choi, and Chin-Wan Chung. 2012. QUBE: a quick algorithm for updating betweenness centrality. In *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, Alain Mille, Fabien Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab (Eds.). ACM, 351–360. https://doi.org/10.1145/2187836.2187884
[21] Sune Lehmann. 2019. Fundamental Structures in Dynamic Communication Networks. *CoRR* abs/1907.09966 (2019). arXiv:1907.09966 http://arxiv.org/abs/1907.09966
[22] Xiaoming Liu, Johan Bollen, Michael L. Nelson, and Herbert Van de Sompel. 2005. Co-authorship networks in the digital library research community. *Inf. Process. Manag.* 41, 6 (2005), 1462–1480. https://doi.org/10.1016/j.ipm.2005.03.012
[23] Andreas Maurer and Massimiliano Pontil. 2009. Empirical Bernstein Bounds and Sample Variance Penalization. arXiv:0907.3740 [stat.ML]
[24] Mark Newman. 2010. *Networks*. Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199206650.001.0001
[25] Raj Kumar Pan and Jari Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E* 84, 1 (jul 2011), 016105. https://doi.org/10.1103/physreve.84.016105
[26] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. 2017. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining, WSDM 2017, Cambridge, United Kingdom, February 6-10, 2017*, Maarten de Rijke, Milad Shokouhi, Andrew Tomkins, and Min Zhang (Eds.). ACM, 601–610. https://doi.org/10.1145/3018661.3018731
[27] Leonardo Pellegrina and Fabio Vandin. 2021. SILVAN: Estimating Betweenness Centralities with Progressive Sampling and Non-uniform Rademacher Bounds. *CoRR* abs/2106.03462 (2021). arXiv:2106.03462 https://arxiv.org/abs/2106.03462
[28] Matteo Riondato and Evgenios M. Kornaropoulos. 2016. Fast approximation of betweenness centrality through sampling. *Data Min. Knowl. Discov.* 30, 2 (2016), 438–475. https://doi.org/10.1007/s10618-015-0423-0
[29] Matteo Riondato and Eli Upfal. 2018. ABRA: Approximating Betweenness Centrality in Static and Dynamic Graphs with Rademacher Averages. *ACM Trans. Knowl. Discov. Data* 12, 5 (2018), 61:1–61:38. https://doi.org/10.1145/3208351
[30] Maciej Rymar, Hendrik Molter, André Nichterlein, and Rolf Niedermeier. 2021. Towards Classifying the Polynomial-Time Solvability of Temporal Betweenness Centrality. In *Graph-Theoretic Concepts in Computer Science - 47th International Workshop, WG 2021, Warsaw, Poland, June 23-25, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12911)*, Lukasz Kowalik, Michal Pilipczuk, and Pawel Rzazewski (Eds.). Springer, 219–231. https://doi.org/10.1007/978-3-030-86838-3_17
[31] Ioanna Tsalouchidou, Ricardo Baeza-Yates, Francesco Bonchi, Kewen Liao, and Timos Sellis. 2020. Temporal betweenness centrality in dynamic graphs. *Int. J. Data Sci. Anal.* 9, 3 (2020), 257–272. https://doi.org/10.1007/s41060-019-00189-x
[32] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and P. Krishna Gummadi. 2009. On the evolution of user interaction in Facebook. In *Proceedings of the 2nd ACM Workshop on Online Social Networks, WOSN 2009, Barcelona, Spain, August 17, 2009*, Jon Crowcroft and Balachander Krishnamurthy (Eds.). ACM, 37–42. https://doi.org/10.1145/1592665.1592675
[33] Stefan Wuchty and Peter F. Stadler. 2003. Centers of complex networks. 223, 1 (jul 2003), 45–53. https://doi.org/10.1016/s0022-5193(03)00071-7
[34] Erjia Yan and Ying Ding. 2009. Applying centrality measures to impact analysis: A coauthorship network analysis. *J. Assoc. Inf. Sci. Technol.* 60, 10 (2009), 2107–2118. https://doi.org/10.1002/asi.21128
[35] Xiang Zhang, Gong Cheng, and Yuzhong Qu. 2007. Ontology summarization based on rdf sentence graph. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy (Eds.). ACM, 707–716. https://doi.org/10.1145/1242572.1242668

# A MISSING PROOFS

Lemma A.1. *Let $v \in V$, then $\tilde{b}'(v)$ is an unbiased estimator of $b(v)$.*

Proof. Let $X_{sz}$ be a Bernoulli random variable that takes value 1 if the pair of nodes $(s, z)$ is sampled, and 0 otherwise. Since $\mathbb{E}[X_{sz}] = 1/(n(n-1))$, then by the linearity of expectation,

$$\mathbb{E}[\tilde{b}'(v)] = \frac{1}{n(n-1)} \sum_{s,z \in V s \neq z} \frac{\sigma^{sh}_{s,z}(v)}{\sigma^{sh}_{s,z}} \frac{\mathbb{E}[X_{sz}]}{1/n(n-1)} = b(v).$$

□

# B MISSING ALGORITHMS

In this Section we present Algorithm 4, used to compute the temporal betweenness values estimates of the various nodes under the RTP criterion. This is discussed in details in Section 4.3.

**Algorithm 4:** Update betweenness estimates - RTP.

**Input:** $\tilde{B}, i$.
**for** $(u, v, t) \in E$ **do**
  $\sigma^z_{v,t} \leftarrow 0$; $\mathrm{u}_{v,t} \leftarrow 0$
$R \leftarrow$ empty queue;
**foreach** $t : \sigma_{z,t} > 0$ **do**
  **for** $(w, t') \in P_{z,t}$ **do**
    $R$.enqueue($\langle (w, t'), \{z\} \rangle$);
**while** !$R$.empty() **do**
  $\langle (w, t), \mathrm{S} \rangle \leftarrow R$.dequeue()
  **if** $w \neq s$ **then**
    **if** $w \notin \mathrm{S}$ **then**
      $\tilde{B}_{w,i} \leftarrow \tilde{B}_{w,i} + \sigma^z_{w,t} \cdot \sigma_{w,t}/(\sigma_z \cdot \mathrm{u}_{w,t})$
    $\mathrm{S}' \leftarrow \mathrm{S} \cup \{w\}$
    **for** $(w', t') \in P_{w,t}$ **do**
      $\sigma^z_{w',t'} \leftarrow \sigma^z_{w',t'} + \sigma^z_{w,t}/\mathrm{u}_{w,t}$
      $\mathrm{u}_{w',t'} \leftarrow \mathrm{u}_{w',t'} + 1$
      $R$.enqueue($\langle (w', t'), \mathrm{S}' \rangle$);

# C SUPPLEMENTARY DATA

Given $T$ and $G_T$, let $S^k_T = \{v_1, \ldots, v_k\}$ be the top-$k$ nodes ranked by their temporal betweenness values and let $S^k_{G_T} = \{v'_1, \ldots, v'_k\}$

be the top-$k$ nodes ranked by their static betweenness values. We report in Table 3 the Jaccard similarity $J(k) = |S_T^k \cap S_{G_T}^k| / |S_T^k \cup S_{G_T}^k|$ for two different values of $k$.

**Table 3: Static vs temporal top-$k$ nodes Jaccard similarity $J(k)$. We also report the size of the intersection.**

| Name | $J(25)$ | $J(50)$ |
|---|---|---|
| HS | 0.28 (11) | 0.56 (36) |
| FB | 0.22 (9) | 0.18 (15) |

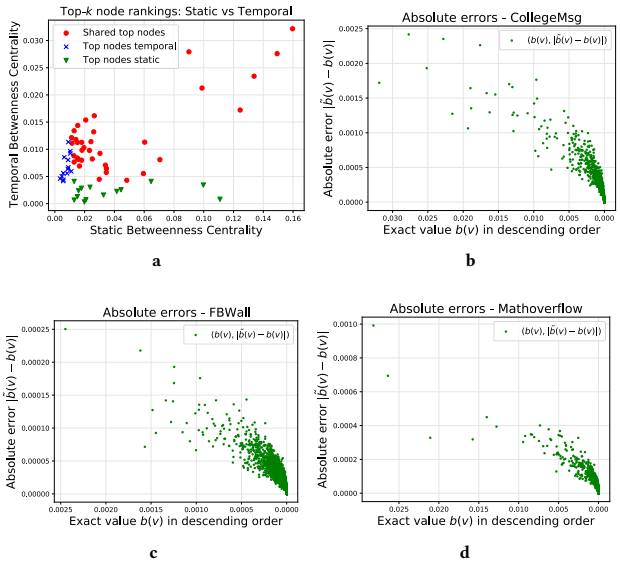



**Figure 4: (4a): static and temporal betweenness values of the top-50 ranked nodes of the dataset dataset; (4b),(4c), and (4d): respectively for datasets `CollegeMsg`, `FBWall`, and `Mathoverflow`, the deviations (or absolute errors) $|\tilde{b}(v) - b(v)|$ between the estimates $\tilde{b}(v)$ and the actual values $b(v)$ of the temporal betweenness centrality, for decreasing order of $b(v)$.**

# Pattern detection in large temporal graphs using algebraic fingerprints*

Suhas Thejaswi† Aristides Gionis‡

**Abstract**

In this paper, we study a family of pattern-detection problems in *vertex-colored temporal* graphs. In particular, given a vertex-colored temporal graph and a multi-set of colors as a query, we search for *temporal paths* in the graph that contain the colors specified in the query. These types of problems have several interesting applications, for example, recommending tours for tourists, or searching for abnormal behavior in a network of financial transactions.

For the family of pattern-detection problems we define, we establish complexity results and design an algebraic-algorithmic framework based on *constrained multilinear sieving*. We demonstrate that our solution can scale to massive graphs with up to hundred million edges, despite the problems being **NP**-hard. Our implementation, which is publicly available, exhibits practical edge-linear scalability and highly optimized. For example, in a real-world graph dataset with more than six million edges and a multi-set query with ten colors, we can extract an optimal solution in less than eight minutes on a haswell desktop with four cores.

## 1 Introduction

Pattern mining in graphs has become increasingly popular due to applications in analyzing and understanding structural properties of data originating from information networks, social networks, transportation networks, and many more. At the same time, real-world data are inherently complex. To accurately represent the heterogeneous and dynamic nature of real-world graphs, we need to enrich the basic graph model with additional features. Thus, researchers have considered *labeled graphs* [40], where vertices and/or edges are associated with additional information represented with labels, and *temporal graphs* [20], where edges are associated with timestamps that indicate when interactions between pairs of vertices took place.

In this paper we study a family of pattern-detection problems in graphs that are both *labeled* and *temporal*. In particular, we consider graphs in which each vertex is associated with one (or more) labels, to which we refer as *colors*, and each edge is associated with a timestamp. We then consider a *motif query*, which is a multi-set of colors. The problem we consider is to decide whether there exists a *temporal path* whose vertices contain exactly the colors specified in the motif query. A temporal path in a temporal graph refers to a path in which the timestamps of consecutive edges are strictly increasing. If such a path exists, we also want to find it and return it as output.

The family of problems we consider have several interesting applications. One application is in the domain of tour recommendations [11], for travelers or tourists in a city. In this case, vertices correspond to locations. The colors associated with each location represent different activities that can be enjoyed in that particular location. For example, activity types may include items such as museums, archaeological sites, restaurants, etc. Edges correspond to transportation links between different locations, and each transportation link is associated with a timestamp indicating departure time and duration. Furthermore, for each location we may have information about the amount of time recommended to spent in that location, e.g., minimum amount of time required to finish a meal or appreciate a museum. Finally, the multi-set of colors specified in the motif query represents the multi-set of activities that a user is interested in enjoying. In the tour-recommendation problem we would like to find a temporal path, from a starting location to a destination, which satisfies temporal constraints (e.g., feasible transportation links, visit times, and total duration) as well as the activity requirements of the user, i.e., what kind of places they want to visit.

Another application is in the domain of analyzing networks of financial transactions. Here, the vertices represent financial entities, the vertex colors represent features of the entities, and the temporal edges represent financial transactions between entities, annotated with the time of the transaction, amount, and possibly other features. An analyst may be interested in

*This research was supported by the Academy of Finland project "Adaptive and Intelligent Data (AIDA)" (317085), the EC H2020 RIA project "SoBigData++" (871042), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation. We acknowledge the use of computational resources funded by the project "Science-IT" at Aalto University, Finland.

†Department of Computer Science, Aalto University, Finland.

‡Department of Computer Science, KTH Royal Institute of Technology, Sweden, and Department of Computer Science, Aalto University, Finland.

finding long chains of transactions among entities that have certain characteristics, for example, searching for money-laundering activities may require querying for paths that involve public figures, companies with certain types of contracts, and banks in off-shore locations.

In this paper we study the following problems:

$k$-TempPath: find a temporal path whose length is at least $k-1$;

PathMotif: find a temporal path whose vertices contain the set of colors specified by a motif query;

RainbowPath: find a temporal path of length at least $k-1$, whose vertices have distinct colors.

All the problems we consider are **NP**-hard; thus, there is no known efficient algorithm to find exact solutions. In such cases most algorithmic solutions resort to approximation schemes. In this paper we present an (exact) *algebraic approach* based on *constrained multilinear sieving* for pattern detection in temporal graphs.

The algorithms based on constrained multilinear detection offer the theoretically best-known results for a set of combinatorial problems including $k$-path [4], Hamiltonian path [5], graph motifs [6] and many more. The implementations based on multilinear sieving are known to saturate the empirical arithmetic and memory bandwidth on modern CPU and GPU micro-architectures. Furthermore, these implementations can scale to large graphs as well as large query sizes [7, 22].

Even though these algebraic techniques have been studied extensively in the algorithms community, they are not been applied to data-mining problems. To the best of our knowledge this is the first paper that applies these approaches for data mining and exploratory graph analysis. Furthermore, this is the first work that applies these techniques for pattern detection in temporal graphs.

Our key contributions are as follows:

- We introduce a set of pattern-detection problems that originate in the *vertex-colored* and *temporal* graphs. For the problems we define we present **NP**-hardness results, while showing that they are *fixed-parameter tractable* [10], meaning that, if we restrict the size of motif query the problems are solvable in polynomial time in the size of host graph.
- We present a general algebraic-algorithmic framework based on constrained multilinear sieving. Our solution exhibits edge-linear scalability. The applications of the algorithmic approach described in this work is not limited to temporal paths, but rather it can be extended to study information cascades, temporal arborescences and temporal subgraphs.
- We engineer an implementation of the algebraic algorithm and demonstrate that the implementation can scale for graphs with up to hundred million edges.
- Open-source release: our implementation and datasets are released as open source [33].

Due to space constraints, the proofs of our technical contributions are omitted from the conference proceedings. A more detailed version of this work is available [34].

## 2 Related work

Pattern detection and pattern counting are fundamental problems in data mining. In the context of paths and trees, pattern matching problems have been extensively studied in non-temporal graphs both in theory [14, 17, 29] as well as applications [3, 21]. For many restricted variants of path problems Kowalik and Lauri presented complexity results and deterministic algorithms with probably optimal runtime bounds [29]. Most of these problems are known to be fixed-parameter tractable and the best known randomized algorithms for a subset of path and subgraph pattern detection problems is due to Björklund et al. [4, 6]. Color coding can be used to approximately count the patterns in $\mathcal{O}^*(2^k)$ time, however, these algorithms require $\mathcal{O}^*(2^k)$ memory [1].[1] A practical implementation of color coding using adaptive sampling and succint encoding was demonstrated by Bressan et al. [8] for the pattern-counting problem. However, the techniques based on color coding are mostly used to detect and count patterns in graphs with no vertex labels.

Algebraic algorithms based on multilinear and constrained multilinear sieving are due to the pioneering work of Koutis [23–25], Williams [37], Koutis and Williams [26, 27]. The approach has been extended to various combinatorial problems using a multivariate variant of the sieve by Björklund et al. [4]. Dell et al. [12] used the decision oracles introduced by Björklund et al. to approximately count the motifs. A practical implementation of multilinear sieving and its scalability to large graphs has been demonstrated by Björklund et al. [7]. Furthermore, its parallelizability to vector-parallel architectures and scalability to large multi-set sizes was shown by Kaski et al. [22].

In the recent years there has been a lot of progress with respect to mining temporal graphs. The most relevant work includes methods for efficient computation of network measures, such as centrality, connectivity, density, motifs, etc. [20, 30], as well as mining frequent subgraphs in temporal networks [32, 36]. Path problems in temporal graphs are well studied [16, 38]. Many

[1] The $*$ in the notation $\mathcal{O}^*$ hides the polynomial factor.

variants of the path problems are known to be solvable in polynomial-time [38, 39]. Surprisingly, a simple variant to check the existence of a temporal path with waiting time constraints was shown to be **NP**-complete by Casteigts et al. [9], more strongly, they proved that the problem is **W**[1]-hard. A known variant of the temporal-path problem is finding top-$k$ shortest paths. In this setting, one asks to find not only a shortest path, but also the next $k-1$ shortest paths, which may be longer than the shortest path [19]. Here by shortest path we mean that the total elapsed time of the temporal path is minimized. Note that the top-$k$ shortest path is different from the $k$-TempPath problem studied in our work.

With the availability of social media data in the recent years there has been growing interest to study pattern mining problems in temporal graphs. Paranjape et al. [32] presented efficient algorithms for counting small temporal patterns. Liu et al. [31] presented complexity results and approximation methods for counting patterns in temporal graphs. However, they mainly study temporal graphs with no vertex-labels (colors). Kovanen et al. [28] studied a general variant of the temporal subgraph problem in temporal graphs with vertex labels. Aslay et al. [2] presented methodologies for counting frequent patterns with vertex and edge labels in streaming graphs. However, most of these approaches are limited to small pattern sizes (up to 3 vertices).

To the best of our knowledge, there is no existing work related to detecting and extracting temporal patterns with vertex labels. The problems considered in this paper are closely related to variants of classical problems such as orienteering problem, TSP and Hamiltonian path [15, 35]. A motivating application for the problems can be traced to the context of tour recommendations [11, 18].

## 3 Method overview

Our method relies on the *algebraic-fingerprinting* technique [26, 37]. As this technique is not well known in the data-mining community, we provide a bird's eye view. The approach is described in more detail in Section 6.

In a nutshell, the problem is to decide the existence of a pattern, or a structure in the data. The idea is to encode the pattern-discovery problem as a polynomial over a set of variables. The variables represent entities of the problem instance (e.g., vertices or edges), and their values represent possible solutions (e.g., whether a vertex belongs to a path). The challenge is to find a polynomial encoding that has the property that a solution to our problem exists if and only if the polynomial evaluates to a non-zero term. We can then verify the existence of a solution, using *polynomial identity testing*, in particular, by evaluating random substitutions of variables: if one of them does not evaluate to zero, then the polynomial is not identically zero. Thus, the method can give false negatives, but the error probability can be brought arbitrarily close to zero.

It should also be noted that an explicit representation of the polynomial can be exponentially large. However, we do not need to represent the polynomial explicitly, since we only need to be able to evaluate the variable substitutions very fast.

## 4 Terminology

In this section we introduce the basic terminology used in the paper.

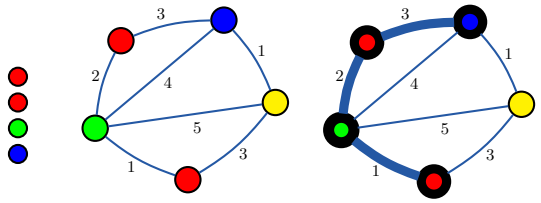
A *graph* $G$ is a tuple $(V, E)$ where $V$ is a set of *vertices* and $E$ is a set of unordered pairs of vertices called *edges*. We denote the number of vertices $|V| = n$ and the number of edges $|E| = m$. Vertices $u$ and $v$ are *adjacent* if there exists an edge $(u, v) \in E$. The set of vertices adjacent to vertex $u$ is denoted by $N(u)$. A *walk* between any two vertices is an alternating sequence of vertices and edges $u_1e_1u_2 \dots e_ku_{k+1}$ such that there exists an edge $e_i = (u_i, u_{i+1}) \in E$ for each $i \in [k]$.[2] We call the vertices $u_1$ and $u_{k+1}$ the *start* and *end* vertices of the walk, respectively. The *length* of a walk is the number of edges in the walk. A *path* is a walk with no repetition of vertices.

A *temporal graph* $G^\tau$ is a tuple $(V, E^\tau)$, where $V$ is a set of vertices and $E^\tau$ is a set of temporal edges. A *temporal edge* is a triple $(u, v, j)$ where $u, v \in V$ and $j \in Z_{\geq 0}$ is a *timestamp*. The *maximum timestamp* in $G^\tau$ is denoted by $t$. The total number of edges at time instance $j \in [t]$ is denoted by $m_j$ and the total number of edges in a temporal graph is $m = \sum_{j\in[t]} m_j$. A vertex $u$ is adjacent to vertex $v$ at timestamp $j$ if there exists an edge $(u, v, j) \in E^\tau$. The set of vertices adjacent to vertex $u$ at time step $j$ is denoted by $N_j(u)$. The set of vertices adjacent to vertex $u$ is denoted by $N(u) = \bigcup_{j\in[t]} N_j(u)$.

A *temporal walk* $W^\tau$ between any two vertices is an alternating sequence of vertices and temporal edges $u_1e_1u_2e_2 \dots e_ku_{k+1}$ such that there exists an edge $e_i = (u_i, u_{i+1}, j) \in E^\tau$ for all $i \in [k]$ and for any two edges $e_i = (u_i, u_{i+1}, j)$, $e_{i+1} = (u_{i+1}, u_{i+2}, j')$ it is $j < j'$. In other words, the timestamps of the edges should always be in strictly increasing order. The *length* of a temporal walk is the number of edges in the temporal walk. A *temporal path* is a temporal walk with no repetition of vertices.

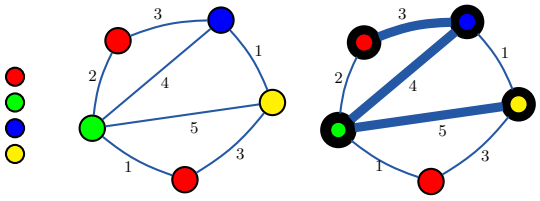
In the next section we will introduce a set of path problems in temporal graphs and an exact algorithm

[2] For convenience we represent $\{1, 2, \dots, k\}$ as $[k]$.



(a) A motif query and a temporal graph (b) A PathMotif

Figure 1: An example of the PathMotif problem in temporal graphs.



(a) A motif query and a temporal graph (b) A RainbowPath

Figure 2: An example of the RainbowPath problem in temporal graphs.

based on *multilinear sieving* is presented in Section 6.

## 5 Path problems in temporal graphs

Let us begin our discussion with the $k$-path problem for static graphs before continuing to temporal graphs.

**The $k$-path problem in static graphs.** Given a graph $G = (V, E)$ and an integer $k \leq n$ the $k$-Path problem asks to decide whether there exists a path of length at least $k-1$ in $G$. The $k$-Path problem is **NP**-complete [15, ND29]. Fortunately, the problem is *fixed-parameter tractable*. The best known fixed-parameter tractable algorithm is due to Björklund et al. [4] and has complexity $\mathcal{O}^*(1.66^k)$.

**The $k$-path problem in temporal graphs.** Given a temporal graph $G^\tau = (V, E^\tau)$ and an integer $k \leq n$ the $k$-TempPath problem asks to decide whether there exists a *temporal path* of length at least $k-1$ in $G^\tau$. For the hardness, a reduction from $k$-Path to $k$-TempPath is straightforward.

Lemma 5.1. *Problem $k$-TempPath is* **NP***-complete.*

**Path motif problem in temporal graphs.** Given a vertex-colored temporal graph $G^\tau = (V, E^\tau)$ and multi-set $M$ of colors the PathMotif problem asks to decide whether there exists a temporal path $P^\tau$ in $G^\tau$ such that the vertex colors of $P^\tau$ agrees with $M$. An example of PathMotif problem is illustrated in Figure 1.

The PathMotif problem is **NP**-complete — a reduction from $k$-TempPath is straightforward.

Lemma 5.2. *Problem* PathMotif *is* **NP***-complete.*

**Rainbow path problem in temporal graphs.** Given a temporal graph $G^\tau = (V, E^\tau)$, an integer $k \leq n$, and a coloring function $c : V \to [k]$, the RainbowPath problem asks us to decide whether there exists a temporal path $P^\tau$ of length $k-1$ such that all vertex colors of $P^\tau$ are different. An example of RainbowPath problem is illustrated in Figure 2.

The RainbowPath problem is a special case of the PathMotif problem, where all the colors in the multi-set $M$ are different, that is $M = [k]$. It is easy to see that the RainbowPath problem in static graphs can be reduced to the RainbowPath problem in temporal graphs by replacing each static edge with $k-1$ temporal edges.[3] So, the RainbowPath problem is **NP**-complete.

## 6 Algebraic algorithm for temporal paths

We now present an algorithm for the $k$-TempPath and PathMotif problems. Our algorithm relies on a *polynomial encoding* of temporal walks and the *algebraic fingerprinting* technique [6, 23, 26, 37]. The algorithm is presented in three steps: (i) we present a dynamic programming recursion to generate polynomial encoding of temporal walks; (ii) we present an algebraic algorithm to detect the existence of an multilinear monomial in the polynomial generated using the recursion in (i) — furthermore, we prove that existence of a multilinear monomial implies existence of a temporal path; (iii) finally, we extend the approach to detect temporal paths with additional color constraints using constrained multilinear detection. Let us begin with the concept of polynomial encoding of temporal walks.

Let $P$ be a multivariate polynomial such that every monomial $M$ is of the form $x_1^{d_1} x_2^{d_2} \dots x_q^{d_q} y_1^{f_1} y_2^{f_2} \dots y_r^{f_r}$. A monomial is *multilinear* if $d_i \in \{0, 1\}$ for all $i \in [q]$ and $f_j \in \{0, 1\}$ for all $j \in r$. A monomial is *$x$-multilinear* if $d_i \in \{0, 1\}$ for all $i \in q$, i.e., we do not take into account the degrees of the $y$-variables. The degree of a monomial $M$ is the sum of the degrees of all its variables. However, for a $x$-multilinear monomial the degree is the sum of degrees of $x$-variables.

**Monomial encoding of a temporal walk.** Let

[3] Given a static graph $G = (V, E)$ and a coloring function $c : V \to [k]$, the RainbowPath problem in static graphs asks us to find a path $P$ of length $k-1$ such that all vertex colors of $P$ are different. The RainbowPath problem in static graphs is known to be **NP**-complete [13].

$W^\tau = v_1 e_1 v_2 \dots e_{k-1} v_k$ be a temporal walk in a temporal graph $G^\tau = (V, E^\tau)$. Let $\{x_{v_1}, \dots, x_{v_n}\}$ be a set of variables representing vertices in $V$ and $\{y_{uv,\ell,i} : (u,v,i) \in E^\tau, \ell \in [k]\}$ be a set of variables such that $y_{uv,\ell,i}$ correspond to an edge $(u,v,i) \in E^\tau$ that appears at position $\ell$ in $W^\tau$. A monomial encoding of $W^\tau$ is represented as

$$x_{v_1} y_{v_1 v_2,1,i_1} x_{v_2} y_{v_1 v_2,2,i_2} \dots y_{v_{k-1} v_k,k-1,i_{k-1}}, x_{v_k},$$

where $i_1, \dots, i_{k-1}$ denote the timestamps on the edges $e_1, \dots, e_{k-1}$, respectively.

It can be shown that the above encoding of $W^\tau$ is $x$-multilinear if and only if $W^\tau$ is a temporal path.

LEMMA 6.1. *The monomial encoding of a temporal walk $W^\tau$ is $x$-multilinear (and multilinear) if and only if the temporal walk is a temporal path.*

**Generating polynomial for temporal walks.** We present a recursion to generate temporal walks. Let $P_{u,\ell,i}$ denote the encoding of all walks of length $\ell - 1$ ending at vertex $u$ and at latest time $i$. Let $v_1$ be a vertex such that $N_i(v_1) = \{v_2, v_3, v_4\}$. Let $P_{v_2,\ell-1,i-1}$, $P_{v_3,\ell-1,i-1}$ and $P_{v_4,\ell-1,i-1}$ represent the polynomial encoding of walks ending at vertices $v_2$, $v_3$ and $v_4$, respectively, such that all walks have length $\ell - 2$ and end at latest time $i-1$. Let $P_{v_1,\ell,i-1}$ denote polynomial encoding of all walks of length $\ell - 1$, ending at $v_1$ at latest time $i-1$. The example is illustrated in Figure 3.

The polynomial encoding to represent walks of length $\ell - 1$ ending at $v_1$ and at latest time $i$ can be written as:

$$\begin{aligned} P_{v_1,\ell,i} \quad = \quad & x_{v_1} y_{v_2 v_1,\ell,i} P_{v_2,\ell-1,i-1} + \\ & x_{v_1} y_{v_3 v_1,\ell,i} P_{v_3,\ell-1,i-1} + \\ & x_{v_1} y_{v_4 v_1,\ell,i} P_{v_4,\ell-1,i-1} + P_{v_1,\ell,i-1}. \end{aligned}$$

Intuitively, the above equation represents that we can reach vertex $v_1$ at time step $i$ if we have already reached any of its neighbors in $N_i(v_1)$ by latest timestamp $i-1$. Furthermore, the term $P_{v_1,\ell,i-1}$ is included so that if we have reached $v_1$ at latest time $i-1$ we can choose to stay at $v_1$ for timestamp $i$.

By generalizing the above idea, a generating function for $P_{u,\ell,i}$, for each $u \in V$, $\ell \in [k]$, and $i \in [t]$ is written as follows:

$$P_{u,\ell,i} = x_u \sum_{v \in N_i(u)} y_{uv,\ell,i} P_{v,\ell-1,i-1} + P_{u,\ell,i-1}. \tag{6.1}$$

Furthermore, let us form the polynomial $\mathcal{P}_{\ell,i} = \sum_{u \in V} P_{u,\ell,i}$, for each $\ell \in [k]$ and $i \in [t]$. More precisely, $\mathcal{P}_{\ell,i}$ denotes the polynomial encoding of all walks of length $\ell - 1$ ending at latest timestamp $i$. Now the problem of detecting a $k$-TEMPPATH is equivalent to finding a $x$-multilinear monomial in $\mathcal{P}_{k,t}$. From the construction of the generating function in (6.1) it is clear that $y$ variables are always distinct and detecting a $x$-multilinear monomial is equivalent to detecting a multilinear monomial.

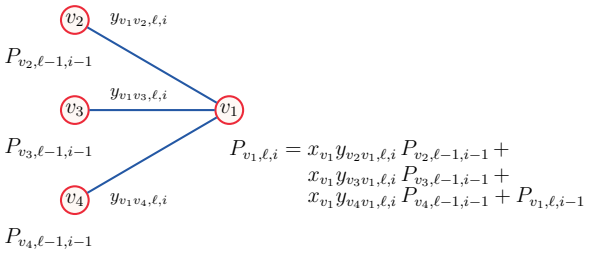



Figure 3: An illustration of the polynomial encoding of temporal walks.

LEMMA 6.2. *The polynomial encoding $P_{u,\ell,i}$ in (6.1) contains a $x$-multilinear monomial of degree $\ell$ if and only if there exists a temporal path of length $\ell-1$ ending at vertex $u$ at latest time $i$.*

**Multilinear sieving.** From Lemma 6.2 the problem of deciding the existence of a $k$-TEMPPATH in $G^\tau$ reduces to detecting the existence of a multilinear monomial term in $\mathcal{P}_{k,t}$.

Let $L$ be the set of $k$ labels and $[n]$ the set denoting vertices in $V$. For each vertex $i \in [n]$ and label $j \in L$ we introduce a new variable $z_{i,j}$. The vector of all variables of $z_{i,j}$ is denoted as $\mathbf{z}$ and the vector of all $y$-variables as $\mathbf{y}$. Now we can determine the existence of a multilinear monomial in $\mathcal{P}_{k,t}$ by making $2^k$ random substitutions of the new variables in $\mathbf{z}$ using the technique described by Björklund, Kaski and Kolwalik [5]. The algorithm is randomized and has a false negative probability of $\frac{2k-1}{2^b}$ where the arithmetic is over GF($2^b$) [6].

LEMMA 6.3. ([5]) *The polynomial $\mathcal{P}_{k,t}$ has at least one multilinear monomial if and only if the polynomial*

$$Q(z, \mathbf{y}) = \sum_{A \subseteq L} \mathcal{P}_{k,t}(z_1^A, \dots, z_n^A, \mathbf{y}) \tag{6.2}$$

*is not identically zero, where $z_i^A = \sum_{j \in A} z_{i,j}$ for all $i \in [n]$ and $A \subseteq L$.*

**Algorithm.** Our algorithm for $k$-TEMPPATH works as follows: $(i)$ we construct a polynomial representing all temporal walks of length $k-1$ using the recursion (6.1);

and $(ii)$ we check if there exits a $x$-multilinear monomial term in the polynomial generated from $(i)$ using Lemma 6.3. From Lemma 6.2, existence of a $x$-multilinear monomial term of size $k$ implies existence of a temporal path of length $k-1$ and vice versa.

LEMMA 6.4. *There exists an algorithm for solving the* $k$-TEMPPATH *problem in* $\mathcal{O}(2^k k(nt+m))$ *time and* $\mathcal{O}(nt)$ *space.*

LEMMA 6.5. $k$-TEMPPATH *is fixed-parameter tractable.*

**Constrained multilinear sieving.** The previous section describes an algorithm for detecting $k$-TEMPPATH. Now we discuss how to extend this approach to detect PATHMOTIF using constrained multilinear sieving technique.

If we observe carefully, to obtain a PATHMOTIF we need to find a multilinear monomial term in the polynomial $\mathcal{P}_{k,t}$ such that the vertex colors corresponding to the $x$-variables with degree one agrees to that of the multi-set $M$. This can be done by imposing additional constraints while evaluating the sieve.

Let $C$ be a set of $n$ colors and $c : [n] \to C$ a function that associates each $i \in [n]$ to a color in $C$. For each color $s \in C$ let us denote the number of occurrences of color $s$ by $\mu(s)$. A monomial $x_1^{d_1} \dots x_q^{d_q} y_1^{f_1} \dots y_r^{f_r}$ is properly colored if for all $s \in C$ it holds that $\mu(s) = \sum_{i \in c^{-1}(s)} d_i$, more precisely the number of occurrences of color $s$ is equal to the total degree of $x$-variables representing the vertices with color $s$.

For each $s \in C$, let $S_s$ be the set of $\mu(s)$ with color $s$ such that $S_s \cap S_{s'} = 0$ for all $s \neq s'$. For $i \in [n]$ and $d \in S_{c(i)}$ we introduce a new variable $v_{i,d}$. Let $L$ be a set of $K$ labels. For each $d \in \cup_{s \in C} S_s$ and each label $i \in L$ we introduce a new variable $w_{i,d}$.

LEMMA 6.6. ([6]) *The polynomial* $\mathcal{P}_{k,t}$ *has at least one monomial that is both* $x$*-multilinear and properly colored if and only if the polynomial*

$$Q(z, \mathbf{w}, \mathbf{y}) = \sum_{A \subseteq L} \mathcal{P}_{k,t}(z_1^A, \dots, z_n^A, \mathbf{y}) \tag{6.3}$$

*is not identically zero, where*

$$z_i^A = \sum_{j \in A} z_{i,j}, \quad \text{and} \quad z_{i,j} = \sum_{d \in S_{c(i)}} v_{i,d} w_{d,j}. \tag{6.4}$$

From Lemmas 6.4 and 6.6 it follows that we have an $\mathcal{O}(2^k k(nt+m))$ algorithm to solve PATHMOTIF problem in $\mathcal{O}(nt)$ space.

**Obtaining an optimal solution.** In this section we describe a procedure to obtain an optimal path. By optimal we mean that the maximum timestamp of the edges in the temporal path is minimized. For simplicity, we refer to our algorithm for the decision version as *decision oracle*. To find the minimum (optimal) timestamp $t' \in [t]$, we make at most $\mathcal{O}(\log t)$ queries to the decision oracle using binary search on range $[t]$.

**Extracting a solution.** In the previous sections we described an algebraic solution for the decision version of the PATHMOTIF problem. In many cases we need to extract a solution, if such a path exists. We use the decision oracle as a subroutine to find a solution in at most $\mathcal{O}(n)$ queries as follows: $(i)$ for each vertex $v \in V$ we remove the vertex $v$ and the edges incident to it and query the oracle. If there is a solution, then we continue to next vertex; otherwise we put back $v$ and the edges incident to it, and continue to next vertex. In this way, we can obtain a subgraph with $k$ vertices in at most $n-k$ queries to the oracle. However, the number of queries to the decision oracle can be reduced to $\mathcal{O}(k \log n)$ queries in expectation by recursively dividing the graph in to two halves [5]; $(ii)$ pick an arbitrary start vertex in the subgraph obtained from (i) and find a temporal path connecting all the $k$ vertices using temporal DFS, if such a path do not exist then continue to next vertex. Even though the worst case complexity is $\mathcal{O}(k!)$, in practice this approach works very fast. However, extracting a solution can be done using $\mathcal{O}(k)$ queries to the decision oracle using vertex-localized sieving. For the reasons of space we skip a detailed discussion of this approach.

**Path motif problem with delays.** In a real-world transport network a transition between any two locations would involve a *transition time* and a minimum *delay time* at a location before continuing the journey, for example a minimum time to visit a museum. In this section, we introduce a problem setting with transition and delay times and present generating polynomials to solve the problems.

For a temporal graph $G^\tau = (V, E^\tau)$, an edge $e \in E^\tau$ is a quadruple $(u, v, i, \Delta)$ where $u, v \in V$, $i \in \mathbb{Z}_{\geq 0}$ is a time instance and $\Delta \in \mathbb{Z}_{\geq 0}$ is transition time from $u$ to $v$. Additionally, each vertex has a delay time $\delta : V \to \mathbb{Z}_{\geq 0}$. The encoding with transition time and delay is the following:

$$P_{u,\ell,i+\Delta} = x_u \sum_{(u,v,i,\Delta) \in E} y_{uv,i} P_{v,\ell-1,i-\delta(v)} + P_{u,\ell,i-1}.$$

From Lemmas 6.2 and 6.6, it follows that existence of a multilinear monomial in the polynomial generated above would imply the existence of a PATHMOTIF.

## 7 Implementation

We use the design of Björklund et al. [7] as a starting point for our implementation, in particular we make use

Table 1: Comparison of extraction time for baseline and algebraic algorithms. All runtimes are in seconds.

| $m$ | Regular | | | Powlaw $d^{-0.5}$ | | | Powlaw $d^{-1.0}$ | | |
|---|---|---|---|---|---|---|---|---|---|
| | Base | Alg | Sp | Base | Alg | Sp | Base | Alg | Sp |
| 1040 | 0.1 | 0.1 | 1 | 0.1 | 0.1 | 1 | 0.1 | 0.1 | 1.0 |
| 10040 | 0.5 | 0.1 | 5 | 1.0 | 0.1 | 10 | 10.8 | 0.1 | **108** |
| 100040 | 5.6 | 1.1 | 5 | 30.4 | 1.1 | 27.6 | 20430.2 | 1.0 | **20430.2** |
| 1000040 | 74.0 | 12.0 | 6 | 808.2 | 11.2 | **72.1** | – | 10.1 | – |

of fast finite-field arithmetic implementation.

Our effort boils down to implementing the generating function (6.1) and evaluating the recurrence at $2^k$ random points. Specifically, we introduce a domain variable $x_v$ for each $v \in V$ and a support variable $y_{uv,\ell,i}$ for each $\ell \in [k]$ and $(u, v, i) \in E^\tau$. The values of variables $x_v$ are computed using Equation (6.4) and the values of variables $y_{uv,\ell,i}$ are assigned uniformly at random.

Our current implementation uses $\mathcal{O}(ntk)$ memory instead of $\mathcal{O}(nt)$. In order to reduce the memory access latency we arrange our memory layout as $k \times t \times n$; furthermore, we employ hardware prefetching [7, § 3.6] to saturate the memory bandwidth and a parallelization scheme [7, § 3.5] to achieve thread-level parallelism.

Our software is available as open source [33].

## 8 Experimental evaluation

In this section we discuss our experimental evaluation.

**Baseline.** For the problems considered in this paper we are not aware of any known baselines to compare. Thus, we implemented two baselines: (i) an *exhaustive-search* algorithm using temporal DFS, and (ii) a *brute-force* algorithm based on random walks. The details of these algorithms are available in an extended abstract [34]. The brute-force algorithm does not work in practice, even for small graphs ($m = 10^4$). For this reason, we experiment only with the exhaustive-search baseline. We note that the baseline is highly optimized and thread parallelized.

**Hardware.** We experiment with two configurations.
*Workstation*: A Fujitsu Esprimo E920 with 1×3.2 GHz Intel Core i5-4570 CPU, 4 cores, 16 GB memory, Ubuntu, and `gcc` v 5.4.0.
*Computenode*: A Dell PowerEdge C4130 with 2×2.5 GHz Intel Xeon 2680 V3 CPU, 24 cores, 12 cores/CPU, 128 GB memory, Red Hat, and `gcc` v 6.3.0.

Our executions make use of all cores.

**Input graphs.** We evaluate our methods using both synthetic and real-world graphs. (*i*) We use two types of *synthetic graphs*: random $d$-regular graphs; and power-law graphs. (*ii*) We use the *real-world* road transport networks from the cities of Helsinki and Madrid. A description of datasets and graph configuration models is available in an extended version of this paper [34].

Our baseline and scalability experiments are performed on RainbowPath problem instances, remember that, in RainbowPath problem every vertex matches with a multi-set color. Likewise, no trivial preprocessing step can be employed to reduce the graph size.

### 8.1 Experimental results.

We now describe our results and key findings. Recall that *decision time* is the time required to decide the existence of one solution, while *extraction time* is the time required to extract such a solution. As discussed previously, extracting a solution requires multiple calls to the decision oracle. All the experiments are executed on the workstation using all cores, with an only exception for the experiments with scalability to large graphs which is executed on the computenote.

**Baseline.** Our first set of experiments compares the *extraction* time to obtain an optimal solution using our algebraic algorithm and the exhaustive-search baseline. In Table 1, we report extraction times for: (*i*) *d-regular* random graphs with $n = 10^2, \ldots, 10^5$ and fixed values of $d = 20$, $t = 100$, $k = 5$; (*ii*) *power-law* graphs with $n = 10^2 \ldots, 10^5$, $D = 20$, $w = 100$, $k = 5$, $\alpha = -0.5$; and (*iii*) $\alpha = -1.0$. Vertex colors are assigned randomly in the range $[k]$ and the multi-set is $[k]$. Each graph instance has at least ten target instances agreeing multi-set colors with different timestamps chosen uniformly at random. For the baseline we report the *minimum* time of five independent runs, however, for the algebraic algorithm we report the *maximum*. *Speedup* (Sp) is the ratio of baseline and algebraic algorithm runtimes.

Surprisingly, the baseline can compete with the algebraic algorithm in the case of $d$-regular random graphs, however, the runtimes have high variance. On the other hand, the algebraic algorithm is very stable. For the power-law graphs with $m = 10^5$ edges and multi-set size $k = 5$, the algebraic algorithm is at least *twenty thousand* times faster than the baseline. The baseline failed to report a solution in small graphs $m = 10^3$ with large multi-set size $k = 10$.

**Scalability.** Our second set of experiments study scalability with respect to: (*i*) number of edges; (*ii*) multi-set size; (*iii*) number of timestamps; and (*iv*) vertex degree.

Figure 4 (left) reports decision and extraction times for $d$-regular random graphs with $n = 10^2, \ldots, 10^5$ and fixed values of $d = 20$, $k = 8$, $t = 100$. Figure 4 (center-left) shows decision time for $d$-regular random graphs with $k = 10, \ldots, 18$ and fixed values of $n = 10^3$, $d = 20$, $t = 100$. Vertex colors are assigned randomly in the range $[k]$ and the multi-set is $[k]$. We observe a

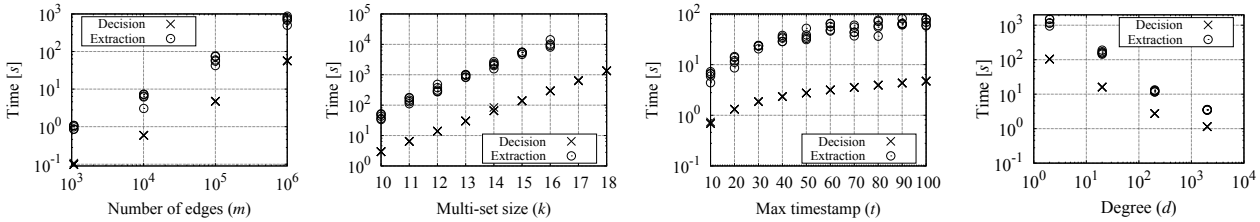



Figure 4: Scalability results. Runtime as a function of the number of edges (left); multi-set size (center-left); number of timestamps (center-right); and degree (right).

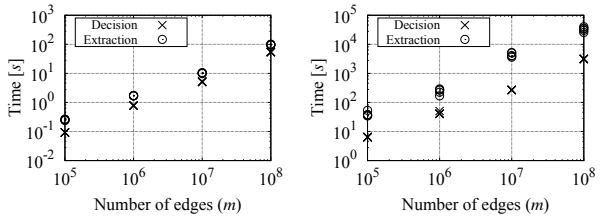



Figure 5: Scaling to large graphs. Runtime as a function of number of edges with $k = 5$ (left) and $k = 10$ (right).

Table 2: Experimental results on real-world graphs. All runtimes are in seconds. H–Helsinki, M–Madrid.

| Dataset | $n$ | $m$ | $t$ | $k=5$ | | $k=10$ | |
|---|---|---|---|---|---|---|---|
| | | | | Base | Alg | Base | Alg |
| Tram(M) | 70 | 35144 | 1265 | 1.37 | 0.24 | 1337.98 | 28.05 |
| Train(M) | 91 | 43677 | 1181 | 40.01 | 0.25 | – | 24.12 |
| Bus(M) | 4597 | 2254993 | 1440 | 6337.89 | 1.27 | – | 278.91 |
| IU-bus(M) | 7543 | 1495055 | 1440 | 744.79 | 1.30 | – | 325.51 |
| Bus(H) | 7959 | 6403785 | 1440 | – | 1.67 | – | 444.66 |

linear scaling with increasing the number of edges and exponential scaling with increasing the multi-set size, as expected by the theory. The variance in decision time is very small for different inputs, however, it is higher for extraction time. The algorithm is able to decide the existence of a solution in less than two minutes for graphs up to one million edges with multi-set size $k = 8$ and extract a solution in less than sixteen minutes.

Next we study the effect of graph density on scalability. Figure 4 (center-right) shows decision and extraction times for $d$-regular random graphs with $t = 10, \ldots, 100$ and fixed values of $n = 10^4$, $d = 20$, $k = 8$. Figure 4 (right) shows decision and extraction times for $d$-regular random graphs with $d = 2, 20, 200, 2000$ and corresponding values of $n = 10^6, \ldots, 10^3$, with fixed $m = 10^6$ and $t = 100$. We observe that the algebraic algorithm performs better for dense graphs. A possible explanation is that for sparse graphs there is not enough work to keep both the arithmetic and memory pipeline busy, simultaneously.

**Scaling to large graphs.** Next we study the scalability of the algebraic algorithm to graphs with up to hundred million edges. Figure 5 reports decision and extraction times for $d$-regular random graphs with $n = 10^3, \ldots, 10^6$, $d = 200$, $t = 100$ with $k = 5$ (left) and $k = 10$ (right). Vertex colors are assigned randomly in the range $[k]$ and the multi-set is $[k]$. In graphs with hundred million edges, the algebraic algorithm can extract an optimal solution in less than two minutes for $k = 5$ and less than two hours for $k = 10$.

**Experiments with real-world graphs.** Finally, we report decision and extraction times for the algebraic algorithm on real-world data. Table 2 reports decision and extraction time (in seconds) for the experiments on real-world datasets. For each dataset we report the maximum time among the five independent executions by choosing multi-set colors at random. For multi-set size $k = 5$, the extraction time is at most two seconds. For larger multi-set size $k = 10$, the extraction time is at most eight minutes in all the datasets. Additionally, we pre-process the graphs by removing vertices whose colors do not match with multi-set colors for both the baseline and the algebraic algorithm.

## 9 Conclusions and future work

In this paper we introduce several pattern-detection problems that arise in the context mining large temporal graphs. We present complexity results, and design algebraic algorithms based on the constrained multilinear sieving technique. Our implementation can scale to large graphs up to hundred million edges despite the problems being **NP**-hard. We present extensive experimental results that validate our scalability claims.

As a future work we would like to consider problem settings where we search for temporal arborescences and temporal subgraphs. Furthermore, we would like to explore the counting variants of these problems.

## References

[1] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, *Biomolecular network motif counting and discovery by color coding*, Bioinformatics, 24 (2008), pp. 241–249.

[2] C. Aslay, A. Nasir, G. De Francisci Morales, and A. Gionis, *Mining frequent patterns in evolving graphs*, in CIKM, 2018, pp. 923–932.

[3] A. Benson, D. Gleich, and J. Leskovec, *Higher-order organization of complex networks*, Science, 353 (2016), pp. 163–166.

[4] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto, *Narrow sieves for parameterized paths and packings*, JCSS, 87 (2017), pp. 119–139.

[5] A. Björklund, P. Kaski, and Ł. Kowalik, *Determinant sums for undirected Hamiltonicity*, SIAM J. Comput., 43 (2014), pp. 280–299.

[6] A. Björklund, P. Kaski, and L. Kowalik, *Constrained multilinear detection and generalized graph motifs*, Algorithmica, 74 (2016), pp. 947–967.

[7] A. Björklund, P. Kaski, L. Kowalik, and J. Lauri, *Engineering motif search for large graphs*, in ALENEX, 2015, pp. 104–118.

[8] M. Bressan, S. Leucci, and A. Panconesi, *Motivo: Fast motif counting via succinct color coding and adaptive sampling*, PVLDB, 12 (2019), pp. 1651–1663.

[9] A. Casteigts, A. Himmel, H. Molter, and P. Zschoche, *The computational complexity of finding temporal paths under waiting time constraints*, CoRR, abs/1909.06437 (2019).

[10] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized algorithms*, 2015.

[11] M. DeChoudhury, M. Feldman, S. Amer-Yahia, N. Golbandi, R. Lempel, and C. Yu, *Automatic construction of travel itineraries using social breadcrumbs*, in HT, 2010, pp. 35–44.

[12] H. Dell, J. Lapinskas, and K. Meeks, *Approximately counting and sampling small witnesses using a colourful decision oracle*, in SODA, 2020, pp. 2201–2211.

[13] E. Eiben, R. Ganian, and J. Lauri, *On the complexity of rainbow coloring problems*, DAM, 246 (2018), pp. 38 – 48.

[14] T. Gagie, D. Hermelin, G. M. Landau, and O. Weimann, *Binary jumbled pattern matching on trees and tree-like structures*, in ESA, 2013.

[15] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 29, W. H. Freeman and Co., 2002.

[16] B. George, S. Kim, and S. Shekhar, *Spatio-temporal network databases and routing algorithms: A summary of results*, in SSTD, 2007, pp. 460–477.

[17] E. Giaquinta and S. Grabowski, *New algorithms for binary jumbled pattern matching*, IPL, 113 (2013), pp. 538–542.

[18] A. Gionis, T. Lappas, K. Pelechrinis, and E. Terzi, *Customized tour recommendations in urban areas*, WSDM, 2014, pp. 313–322.

[19] M. Gupta, C. C. Aggarwal, and J. Han, *Finding top-k shortest path distance changes in an evolutionary network*, in SSTD, 2011, pp. 130–148.

[20] P. Holme and J. Saramäki, *Temporal networks*, Physics reports, 519 (2012), pp. 97–125.

[21] ———, *Temporal networks*, Physics reports, 519 (2012), pp. 97–125.

[22] P. Kaski, J. Lauri, and S. Thejaswi, *Engineering Motif Search for Large Motifs*, in SEA, 2018, pp. 1–19.

[23] I. Koutis, *Faster algebraic algorithms for path and packing problems*, in ICALP, 2008.

[24] ———, *The power of group algebras for constrained multilinear monomial detection*, Dagstuhl meeting 10441, (2010).

[25] ———, *Constrained multilinear detection for faster functional motif discovery*, IPL, 112 (2012), pp. 889–892.

[26] I. Koutis and R. Williams, *Limits and applications of group algebras for parameterized problems*, in ICALP (1), 2009.

[27] I. Koutis and R. Williams, *Algebraic fingerprints for faster algorithms*, Comm. of the ACM, 59 (2016), pp. 98–105.

[28] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki, *Temporal motifs in time-dependent networks*, JSM, 2011 (2011), p. P11005.

[29] L. Kowalik and J. Lauri, *On finding rainbow and colorful paths*, TCS, 628 (2016), pp. 110 – 114.

[30] M. Latapy, T. Viard, and C. Magnien, *Stream graphs and link streams for the modeling of interactions over time*, Social Network Analysis and Mining, 8 (2018).

[31] P. Liu, A. Benson, and M. Charikar, *Sampling methods for counting temporal motifs*, in WSDM, 2019, pp. 294–302.

[32] A. Paranjape, A. Benson, and J. Leskovec, *Motifs in temporal networks*, WSDM, 2017, pp. 601–610.

[33] S. Thejaswi and A. Gionis, 2019. `https://github.com/suhastheju/temporal-patterns`.

[34] ———, *Finding temporal patterns using algebraic fingerprints*, arXiv, abs/2001.07158 (2020).

[35] P. Vansteenwegen, W. Souffriau, and D. V. Oudheusden, *The orienteering problem: A survey*, EJOR, 209 (2011), pp. 1 – 10.

[36] B. Wackersreuther, P. Wackersreuther, A. Oswald, C. Böhm, and K. Borgwardt, *Frequent subgraph discovery in dynamic networks*, in MLG, 2010.

[37] R. Williams, *Finding paths of length k in $O^*(2^k)$ time*, IPL, 109 (2009).

[38] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, *Path problems in temporal graphs*, Proc. VLDB Endow., 7 (2014), pp. 721–732.

[39] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, *Efficient algorithms for temporal path computation*, TKDE, 28 (2016), pp. 2927–2942.

[40] J. Yang, J. McAuley, and J. Leskovec, *Community detection in networks with node attributes*, in ICDM, 2013, pp. 1151–1156.

# THyMe+: Temporal Hypergraph Motifs and Fast Algorithms for Exact Counting

Geon Lee
Graduate School of AI, KAIST
geonlee0325@kaist.ac.kr

Kijung Shin
Graduate School of AI and School of Electrical Engineering, KAIST
kijungs@kaist.ac.kr

2021 IEEE International Conference on Data Mining (ICDM) | 978-1-6654-2398-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDM51629.2021.00042

***Abstract*—Group interactions arise in our daily lives (email communications, on-demand ride sharing, comment interactions on online communities, to name a few), and they together form hypergraphs that evolve over time. Given such temporal hypergraphs, how can we describe their underlying design principles? If their sizes and time spans are considerably different, how can we compare their structural and temporal characteristics?**

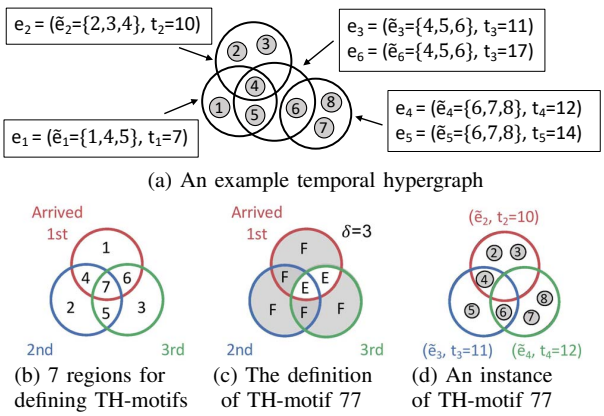
**In this work, we define 96 *temporal hypergraph motifs* (TH-motifs), and propose the relative occurrences of their instances as an answer to the above questions. TH-motifs categorize the relational and temporal dynamics among three connected hyperedges that appear within a short time. For scalable analysis, we develop THyMe$^+$, a fast and exact algorithm for counting the instances of TH-motifs in massive hypergraphs, and show that THyMe$^+$ is at most $2,163\times$ *faster* while requiring less space than baseline. Using it, we investigate 11 real-world temporal hypergraphs from various domains. We demonstrate that TH-motifs provide important information useful for downstream tasks and reveal interesting patterns, including the striking similarity between temporal hypergraphs from the same domain.**

## I. Introduction

Interactions in real-world systems are complex, and in many cases, they are beyond pairwise: email communications, on-demand ride sharing, comment interactions on online communities, to name a few. These group interactions together form a *hypergraph*, which consists of a set of nodes and a set of hyperedges (see Fig. 1(a) for an example). Each *hyperedge* is a subset of *any* number of nodes, and by naturally representing a group interaction among multiple individuals or objects, it contributes to the powerful expressiveness of hypergraphs.

Recently, several empirical studies have revealed structural and temporal properties of real-world hypergraphs. Pervasive structural patterns include (a) heavy-tailed distributions of degrees, edge sizes, and intersection sizes [1]; (b) giant connected components [2], and small diameters [2]; and (c) substantial overlaps of hyperedges with homophily [3]. Temporal properties observed commonly in various time-evolving hypergraphs include (a) significant overlaps between temporally adjacent hyperedges [4]; and (b) diminishing overlaps, densification, and shrinking diameters [1].

In addition to these macroscopic properties, local connectivity and dynamics in real-world hypergraphs have been studied. Benson et al. [5] examined the interactions among a fixed number of nodes, with a focus on their relations with the emergence of a hyperedge containing all the nodes. Lee et al. [6] inspected the overlaps between three hyperedges, which they categorize into 26 patterns called hypergraph motifs (h-motifs). Comparing the relative counts of each h-motif's instances revealed that local structures are particularly similar between hypergraphs from the same domain but different across domains. In h-motifs, however, temporal dynamics are completely ignored.




(a) An example temporal hypergraph

(b) 7 regions for defining TH-motifs

(c) The definition of TH-motif 77

(d) An instance of TH-motif 77

Fig. 1: **(a)** A temporal hypergraph with 8 nodes and 6 temporal hyperedges. **(b)** The 7 regions in the Venn diagram representation for defining TH-motifs. **(c)** The definition of TH-motif 77. 'F' and 'E' stand for 'filled' and 'empty', respectively. **(d)** The sequence $\langle e_2, e_3, e_4 \rangle$ is an instance of TH-motif 77.

This line of research has also revealed that specialized analysis tools (e.g., h-motifs [6] and multi-level decomposition [2]) are useful for extracting unique high-order information that hypergraphs convey and also for coping with additional complexity due to the flexibility in the size of hyperedges. Simply utilizing graph analysis tools (e.g., network motifs [7]) after converting hypergraphs into pairwise graphs is often limited in addressing the above challenges [6], [8].

Motivated by interesting patterns that temporal network motifs revealed in ordinary graphs [9]–[13], we define 96 *temporal hypergraph motifs* (TH-motifs) for local pattern analysis of time-evolving hypergrpahs. TH-motifs generalize the notion of static h-motifs, which completely ignore temporal information, and describe both relational and temporal dynamics among three connected hyperedges that arrive within a short time. Specifically, given three connected hyperedges $e_i$, $e_j$, and $e_k$, all of which arrive within $\delta$ time units, TH-motifs describe their connectivity based on the emptiness of the seven subsets of them shown in Fig. 1(b). In the temporal perspective, the relative arrival orders of $e_i$, $e_j$, and $e_k$ are taken into account, and thus patterns that are indistinguishable

978-1-6654-2398-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDM51629.2021.00042

using static h-motifs can be characterized using TH-motifs.

Given a temporal hypergraph, where a timestamp is attached to each hyperedge (see Fig. 1(a) for an example), we summarize its local structural and temporal characteristics using the relative occurrence of 96 TH-motifs' instances. That is, we obtain a vector of length 96 regardless of the sizes and time spans of hypergraphs, and thus local characteristics of different hypergraphs can easily be compared.

Another focus of this paper is the problem of counting TH-motifs' instances. Since the number of three connected hyperedges can be orders of magnitude larger than the number of hyperedges, directly enumerating all of them is computationally prohibitive, especially for massive hypergraphs. We develop THyME$^+$ (**T**emporal **Hy**pergraph **M**otif **Ce**nsus), which exactly counts each TH-motif's instances while avoiding direct enumeration. In our experiments, THyME$^+$ is up to $\mathbf{2,163}\times$ **faster** than the direct extension of a recent exact temporal network motif counting algorithm [9], which enumerates every static h-motif in the induced static hypergraph. THyME$^+$ makes the best use of our two findings in real-world hypergraphs that temporal hyperedges tend to be (1) repetitive and (2) temporally local. These findings about duplicated (i.e., completely overlapped) hyperedges complement the findings in [4], which focus mainly on partial overlaps.

Using TH-motifs and THyME$^+$, we investigate 11 real-world hypergraphs from 5 distinct domains. Our empirical study demonstrates that TH-motifs are informative, capturing both structural and temporal characteristics. Specifically, using the counts of incident TH-motifs' instances as features brings up to $\mathbf{25.7}\%$ **improvement in the accuracy** of a hyperedge prediction task, compared to when static h-motifs are used instead of TH-motifs. Moreover, TH-motifs reveal interesting patterns, including the striking similarity between hypergraphs from the same domain.

In summary, our contributions are as follow:

1) **New concept:** We define 96 temporal hypergraph motifs (TH-motifs) for characterizing local structures and dynamics in hypergraphs of various sizes.
2) **Fast and exact algorithms:** We develop fast algorithms for exactly counting the instances of TH-motifs, and they are up to $2,163\times$ faster than baseline.
3) **Empirical discoveries:** We demonstrate the usefulness of TH-motifs by uncovering the design principles of 11 real-world temporal hypergraphs from 5 different domains.

**Reproducibility:** The source code and datasets used in this work are available at https://github.com/geonlee0325/THyMe.

In Section II we review preliminaries and related prior works. In Section III, we present the concept of TH-motifs. In Section IV, we develop algorithms for counting the instances of TH-motifs. In Section V, we empirically analyze real-world temporal hypergraphs through the lens of TH-motifs. Lastly, in Section VI, we offer conclusions.

## II. Preliminaries and Related Works

In this section, we first review the concept of hypergraphs. Then, we introduce hypergraph motifs (h-motifs), which is designed for static hypergraphs. Lastly, we discuss other related works. Refer to Table I for the frequently-used notations.

TABLE I: Frequently-used notations.

| Notation | Definition |
|---|---|
| $T = (V, \mathcal{E})$ | temporal hypergraph with temporal hyperedges $\mathcal{E}$ |
| $G_T = (V, E_{\mathcal{E}})$ | induced static hypergraph of the temporal hypergraph $T$ |
| $e_i = (\tilde{e}_i, t_i)$ | temporal hyperedge with nodes $\tilde{e}_i$ arrived at time $t_i$ |
| $I(\tilde{e})$ | set of temporal hyperedges whose nodes are $\tilde{e}$ |
| $h(\tilde{e}_i, \tilde{e}_j, \tilde{e}_k)$ | TH-motif corresponding to an instance $\langle e_i, e_j, e_k \rangle$ |
| $P = (V_P, E_P)$ | projected graph in THyME |
| $Q = (V_Q, E_Q, t_Q)$ | projected graph in THyME$^+$ |

### A. Basic Concepts: Static and Temporal Hypergraphs

A *hypergraph* $G = (V, E)$ consists of a set of nodes $V = \{v_1, ..., v_{|V|}\}$ and a set of hyperedges $E = \{\tilde{e}_1, ..., \tilde{e}_{|E|}\}$. Each *hyperedge* $\tilde{e} \in E$ is a non-empty set of an arbitrary number of nodes. A *temporal hypergraph* $T = (V, \mathcal{E})$ on a node set $V$ is an ordered sequence of *temporal hyperedges*. Each $i^{\text{th}}$ temporal hyperedge $e_i = (\tilde{e}_i, t_i)$ where $\tilde{e}_i \subseteq V$ is the set of nodes and $t_i$ is the time of arrival. Two distinct temporal hyperedges $e_i = (\tilde{e}_i, t_i)$ and $e_j = (\tilde{e}_j, t_j)$ are *duplicated* if they share exactly same set of nodes, i.e., $\tilde{e}_i = \tilde{e}_j$. We assume the sequence is ordered and timestamps are unique, i.e., if $i < j$, then $t_i < t_j$. We denote the set of temporal hyperedges whose nodes are $\tilde{e}$ (i.e., those *inducing* $\tilde{e}$) by $I(\tilde{e}) := \{e_i = (\tilde{e}_i, t_i) \in \mathcal{E} : \tilde{e}_i = \tilde{e}\}$. The temporal hypergraph $T$ induces a static hypergraph $G_T = (V, E_{\mathcal{E}})$ where timestamps and duplicated temporal hyperedges are ignored. That is, a hyperedge $\tilde{e} \in E_{\mathcal{E}}$ in $G_T$ exists if and only if $I(\tilde{e}) \neq \emptyset$. Notably, the number of temporal hyperedges is typically much larger than that of static hyperedges in the induced hypergraph, i.e., $|\mathcal{E}| \gg |E_{\mathcal{E}}|$.

### B. Static Hypergraph Motifs (h-motifs)

*Hypergraph motifs* (h-motifs) [6] are tools for understanding the local structural properties of static hypergraphs. Given three connected hyperedges, h-motifs describe their connectivity patterns by the emptiness of each of seven subsets: (1) $\tilde{e}_i \setminus \tilde{e}_j \setminus \tilde{e}_k$, (2) $\tilde{e}_j \setminus \tilde{e}_k \setminus \tilde{e}_i$, (3) $\tilde{e}_k \setminus \tilde{e}_i \setminus \tilde{e}_j$, (4) $\tilde{e}_i \cap \tilde{e}_j \setminus \tilde{e}_k$, (5) $\tilde{e}_j \cap \tilde{e}_k \setminus \tilde{e}_i$, (6) $\tilde{e}_k \cap \tilde{e}_i \setminus \tilde{e}_j$, and (7) $\tilde{e}_i \cap \tilde{e}_j \cap \tilde{e}_k$. While there can exist $2^7$ possible cases of emptiness, 26 cases of them are considered after excluding symmetric, duplicated, and disconnected ones. Since non-pairwise interactions among the hyperedges (such as $\tilde{e}_i \cap \tilde{e}_j \cap \tilde{e}_k$) are taken into account, h-motifs effectively captures the high-order information of the overlapping patterns of the hyperedges. It is shown empirically that their occurrences in the real-world hypergraphs are significantly different from those in randomized hypergraphs. Moreover, the relative occurrences are particularly similar between hypergraphs from the same domain, while they are distinct between hypergraphs from different domains. Note that h-motifs, which is originally designed for static hypergraphs, completely ignore temporal information.

### C. Other Related Works

In this subsection, we review prior works on network motifs and empirical analysis of hypergraphs.

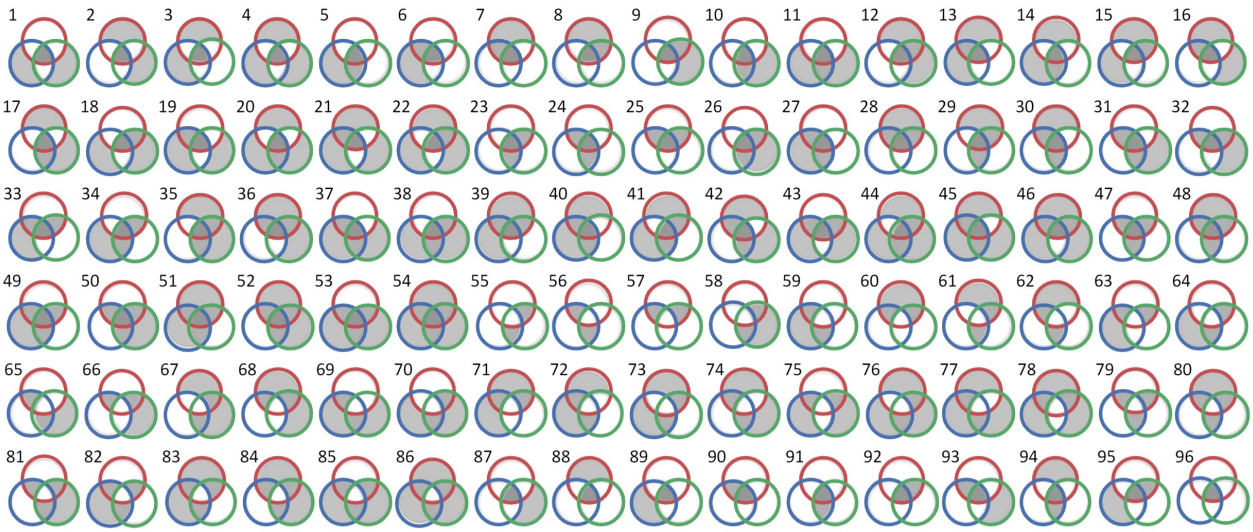


Fig. 2: **The 96 temporal hypergraph motifs (TH-motifs).** In each TH-motif, the red hyperedge arrives first followed by the blue one and then the green one. Each of the 7 distinct regions in the Venn diagram representation is colored white if it is empty, and it is colored grey if it is filled with at least one node. See Fig. 1(d) for an instance of TH-motif 77.

**Network Motifs.** Network motifs are fundamental building blocks of real-world graphs [7], [14]. Their relative occurrences in real-world graphs are significantly different from those in randomized ones [7] and unique within each domain [15]. While they were originally defined on a static graph, they have been extended to temporal [9], heterogeneous [10], [16], and bipartite [17] graphs, as well as hypergraphs [6]. Their usefulness has been demonstrated in a wide range of graph applications including community detection [18]–[22], ranking [23], and embedding [24]–[28].

**Temporal Network Motifs:** The notion of network motifs has been extended to temporal networks to describe patterns in sequences of temporal edges. Several definitions of temporal motifs have been used, and most of them consider the *temporal connectivity* between the edges. In [11] and [12], they consider $\delta$-adjacency between temporal edges. That is, every consecutive edges should share a node and arrive within in $\delta$ time units. Several counting algorithms for such patterns have been proposed [11]–[13]. Another definition of temporal motifs describes patterns of sequences of temporal edges where all edges arrive within $\delta$ time units [9] while taking their relative arrival orders into consideration. In this work, we define TH-motifs based on the notion of temporal motifs defined in [9] due to its simplicity and effectiveness.

**Empirical Analysis of Real-world Hypergraphs:** Empirical analysis of global [2], [3] and local [5], [6] structural patterns and temporal patterns [1], [4], [5] of real-world hypergraphs has been performed, as discussed in detail in Section I.

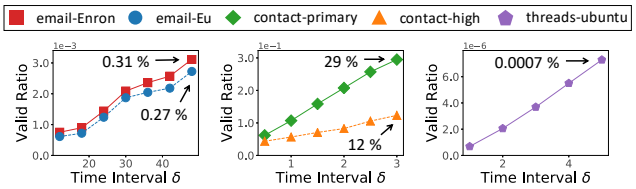
## III. Proposed Concepts

In this section, we propose *temporal hypergraph motifs* (TH-motifs), which are tools for understanding the local structural and temporal characteristics of temporal hypergraphs. We introduce the definition and their relevant concepts.

**Definition:** TH-motifs describe structural and temporal patterns in sequences of three connected temporal hyperedges that are close in time. Note that three hyperedges are *connected* if and only if one among them overlaps with the others. Specifically, given three connected temporal hyperedges $\langle e_i = (\tilde{e}_i, t_i),\ e_j = (\tilde{e}_j, t_j),\ e_k = (\tilde{e}_k, t_k)\rangle$ where $t_i < t_j < t_k$ and $t_k - t_i \leq \delta$ (i.e., they arrive within a predefined time interval $\delta$), TH-motifs describe the emptiness of the 7 subsets: (1) $\tilde{e}_i \setminus \tilde{e}_j \setminus \tilde{e}_k$, (2) $\tilde{e}_j \setminus \tilde{e}_k \setminus \tilde{e}_i$, (3) $\tilde{e}_k \setminus \tilde{e}_i \setminus \tilde{e}_j$, (4) $\tilde{e}_i \cap \tilde{e}_j \setminus \tilde{e}_k$, (5) $\tilde{e}_j \cap \tilde{e}_k \setminus \tilde{e}_i$, (6) $\tilde{e}_k \cap \tilde{e}_i \setminus \tilde{e}_j$, and (7) $\tilde{e}_i \cap \tilde{e}_j \cap \tilde{e}_k$. That is, in the structural aspect, TH-motif describes the emptiness of the seven distinct regions in the Venn diagram representation (see Fig. 1(b)), effectively capturing the high-order connectivity among three hyperedges. In the temporal aspects, TH-motifs take the relative arrival orders of three hyperedges and their time interval into consideration. While there can exist $2^7$ possible cases of emptiness, we consider 96 cases of them, which are called TH-motif 1 to TH-motif 96, after excluding those describing disconnected hyperedges. We visualize the 96 TH-motifs in Fig. 2. Recall that static h-motifs completely ignore temporal information, and also assume that every hyperedge is unique, while TH-motifs also describe the patterns among duplicated temporal hyperedges. Thus, while static h-motifs distinguish only 26 different patterns, TH-motifs distinguish 96 different patterns by considering temporal dynamics in addition to connectivity.

**Instance of TH-motifs:** A sequence $\langle e_i, e_j, e_k\rangle$ of three temporal hyperedges is an *instance* of TH-motif $t$ if their relational and temporal dynamics are described by TH-motif $t$ (see Fig. 1(d) for an example). For each instance $\langle e_i, e_j, e_k\rangle$, we denote its corresponding TH-motif by $h(\tilde{e}_i, \tilde{e}_j, \tilde{e}_k)$.

**Triple, Pair, and Single Inducing TH-motifs:** The 96 TH-motifs can be categorized into three types based on the number of underlying static hyperedges. A TH-motif is *triple-inducing*




(a) email (b) contact (c) threads

Fig. 3: Only a small fraction of static h-motifs' instances in the induced static hypergraphs are induced by any valid instance of TH-motifs. Results in small datasets where the instances of static h-motifs can be exactly counted are reported.

if underlying hyperedges in its instance $\langle e_i, e_j, e_k \rangle$ are distinct (i.e., $\tilde{e}_i \neq \tilde{e}_j$, $\tilde{e}_j \neq \tilde{e}_k$, and $\tilde{e}_k \neq \tilde{e}_i$), as in TH-motifs 1-86. If two are duplicated while the remaining one is different, as in TH-motifs 87-95, it is *pair-inducing*. If all three hyperedges are duplicated, as in TH-motif 96, it is *single-inducing*.

## IV. Counting Algorithms

In this section, we describe methodologies for exactly counting the instances of each TH-motifs in the input temporal hypergraph. We first present DP, which extends a recent exact counting algorithm [9] for temporal network motifs. Then, we describe THyMe, a preliminary version of our proposed algorithm THyMe$^+$. Lastly, we propose THyMe$^+$ (**T**emporal **Hy**pergraph **M**otif C**e**nsus), a fast and efficient algorithm that addresses the limitations of the previous ones.

**Remarks:** The problem of counting TH-motifs has additional technical challenges while it bears some similarity with counting static h-motifs or temporal network motifs. First, the number of temporal hyperedges is typically much larger than that of hyperedges in the underlying static hypergraph. For example, the 11 considered real-world temporal hypergraphs (see Section V-A) have up to $1.2 - 22.0\times$ more hyperedges than the underlying static ones. This incurs significant bottlenecks of enumeration methods, and thus fast algorithms are demanded. Temporal network motifs are defined only by pairwise interactions among a fixed number of nodes and their timestamps. However, TH-motifs are defined not just by pairwise interactions but also by non-pairwise interactions among three hyperedges, in addition to their timestamps.

### A. Dynamic Programming (DP): *Extension of [9]*

We present Dynamic Programming (DP), which is a baseline approach for counting the instances of each TH-motif in the input temporal hypergraph $T$.

**Counting DP:** Given an input temporal hypergraph $T = (V, \mathcal{E})$, DP enumerates the instances of static h-motifs in the induced static hypergraph $G_T = (V, E_{\mathcal{E}})$. This step can be processed by using an existing algorithm provided in [6]. For each instance $\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\}$ of static h-motif in $G_T$, DP counts the instances of each TH-motifs whose temporal hyperedges (a) induce the static h-motif instance and (b) arrive within $\delta$ time. To this end, we adapt the dynamic programming scheme provided by [9], as described in detail in Appendix A.

**Algorithm 1:** THyMe: Preliminary Algorithm

**Input :** (1) temporal hypergraph: $T = (V, \mathcal{E})$
(2) time interval $\delta$
**Output:** # of each temporal h-motif $t$'s instances: $M[t]$

```
M ← map initialized to 0
P = (V_P = ∅, E_P = ∅)
w_s ← 1
for each temporal hyperedge e_i = (ẽ_i, t_i) ∈ ℰ do
    insert(e_i)
    while t_{w_s} + δ < t_i do
        remove(e_{w_s})
        w_s ← w_s + 1
    S ← set of 3 connected temporal hyperedges including e_i
    for each instance ⟨e_j, e_k, e_i⟩ ∈ S do
        M[h(ẽ_j, ẽ_k, ẽ_i)] += 1
return M
Procedure insert(e_i = (ẽ_i, t_i))
    V_P ← V_P ∪ {e_i}
    N_{e_i} ← {e : e ∈ V_P \ {e_i} and ẽ_i ∩ ẽ ≠ ∅}
    E_P ← E_P ∪ {(e_i, e) : e ∈ N_{e_i}}
Procedure remove(e_i = (ẽ_i, t_i))
    V_P ← V_P \ {e_i}
    N_{e_i} ← {e : e ∈ V_P and ẽ_i ∩ ẽ ≠ ∅}
    E_P ← E_P \ {(e_i, e) : e ∈ N_{e_i}}
```

**Limitations of DP:** Using dynamic programming, DP avoids enumerating over all instances of TH-motifs. However, it still enumerates all instances of static h-motifs in the induced hypergraph $G_T$, most of which however are not induced by any valid instance of TH-motifs, as seen in Fig. 3. For example, in threads-ubuntu, only $0.0007\%$ of the static h-motifs instances are induced by any valid instance of TH-motifs when $\delta$ is 5 hours. That is, DP enumerates every three connected hyperedges in $G_T$, ignoring any temporal information, while we are interested only in three connected temporal hyperedges that arrive within in a short period of time.

### B. THyMe: *Preliminary Version of the Proposed Algorithm*

To address the limitations of DP, we present THyMe, a preliminary version of our proposed algorithm THyMe$^+$. THyMe directly enumerates each instance of TH-motifs, instead of those of static h-motifs, to avoid unnecessary search. To this end, THyMe concisely considers the temporal hyperedges that occur in the $\delta$-sized temporal window. In response to the arrival of a new temporal hyperedge $e_i$ at time $t_i$, the temporal window moves to $[t_i - \delta, t_i]$. It maintains only a succinct projected graph $P = (V_P, E_P)$ that represents the connectivity between the temporal hyperedges that occur within the current temporal window. As the window moves, the projected graph $P$ is incrementally updated, reflecting the changes of the current temporal hyperedges. Using $P$, THyMe exhaustively enumerates the instances of TH-motifs.

**Projected Graph in THyMe:** The projected graph $P = (V_P, E_P)$ is a graph where each node is a temporal hyperedge and two nodes are connected as an edge if their corresponding temporal hyperedges share any nodes. In THyMe, $P$ is maintained on the fly, with response to the temporal hyperedges

that either enter or exit the sliding time window. The update schemes are described as `insert` and `remove`, respectively, in Algorithm 1. In `insert`, a temporal hyperedge $e_i$ is added as a node (line 14) and its neighbors (i.e., those in $V_P$ that are adjacent to $e_i$) are joined by edges (lines 15-16). In `remove`, a temporal hyperedge $e_i$, as well as its incident edges are removed from $V_P$ and $E_P$, respectively (lines 18-20).

**Counting in THyME:** The counting procedure of THyME is described in Algorithm 1. The sets of nodes and edges of the projected graph $P$ are initialized to empty maps, i.e., $V_P = \varnothing$ and $E_P = \varnothing$ (line 2). Once a temporal hyperedge $e_i = (\tilde{e}_i, t_i) \in \mathcal{E}$ arrives, the temporal window is moved to $[t_i - \delta, t_i]$ and the projected graph $P$ is updated accordingly, as described above. Then, it enumerates the instances of three connected nodes in $P$, which corresponds to the instances of TH-motifs of $T$ containing $e_i$ (line 9). For each instance $\langle e_j, e_k, e_i \rangle$ of TH-motif $t$, the corresponding count $M[t]$ is incremented (line 11).

**Limitations of THyME:** Though THyME avoids redundant search in the induced static hypergraph $G_T$, it directly enumerates every instance of TH-motifs in $T$. Since the size of the temporal hypergraph is much larger than that of induced static hypergraph, counting the instances in temporal hypergraph can be more computationally challenging, especially when time interval $\delta$ is large. Each temporal hyperedge within the temporal window corresponds to a unique node in the projected graph $P$ even when many temporal hyperedges are highly duplicated, as in real-world hypergraphs (see Section V-E).

### C. THyME+: Advanced Version of the Proposed Algorithm

We present THyME+, our proposed algorithm for exactly counting the instances of TH-motifs. THyME+ is faster and more efficient than DP and THyME, as shown empirically in Section V, by addressing their limitations as follows.

- DP enumerates all instances of static h-motifs in the induced hypergraph $G_T$, where most of them are redundant, not induced by any instance of TH-motifs of the temporal hypergraph $T$. THyME+ selectively enumerates the h-motif instances and thus reduces the redundancy.
- THyME exhaustively enumerates all instances of TH-motifs. THyME+ reduces the enumeration by introducing an effective counting scheme.
- The projected graph $P$ maintained by THyME can be large since each temporal hyperedge is represented as a unique node. THyME+ maintains a projected graph $Q$ that is typically smaller than $P$. In $Q$, the same node can be shared by multiple temporal hyperedges. The motivation behind $Q$ is empirically demonstrated in Section V-E.

**Projected Graph in THyME+:** THyME+ maintains a projected graph $Q = (V_Q, E_Q, t_Q)$ composed of a set of nodes $V_Q$, a set of edges $E_Q$, and a map $t_Q$. Each node and edge represent a static hyperedge and a pair of static hyperedges that share any nodes, respectively. In addition, $t_Q$ maps a set of timestamps of temporal hyperedges inducing a particular static hyperedge. Notably, while each node in the projected graph $P$ used in THyME is a unique temporal hyperedge, $Q$ represents the connectivity between hyperedges in the induced static hypergraph $G_T$. That is, duplicated temporal hyperedges can share the same node in $Q$, and thus the size of the graph can be much smaller than $P$, i.e., $|E_Q| < |E_P|$.

**Algorithm 2:** THyME+: Proposed Algorithm

**Input :** (1) temporal hypergraph: $T = (V, \mathcal{E})$
(2) time interval $\delta$
**Output:** # of each temporal h-motif $t$'s instances: $M[t]$

1 $M \leftarrow$ map initialized to 0
2 $Q = (V_Q = \varnothing, E_Q = \varnothing, t_Q = \varnothing)$
3 $w_s \leftarrow 1$
4 **for each** temporal hyperedge $e_i = (\tilde{e}_i, t_i) \in \mathcal{E}$ **do**
5  `insert` $(e_i)$
6  **while** $t_{w_s} + \delta < t_i$ **do**
7  `remove` $(e_{w_s})$
8  $w_s \leftarrow w_s + 1$
9  $S \leftarrow$ set of 3 connected static hyperedges including $\tilde{e}_i$
10  **for each** instance $\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\} \in S$ **do**
11  `comb3` $(\tilde{e}_i, \tilde{e}_j, \tilde{e}_k)$
12  **for each** pair $(\tilde{e}_i, \tilde{e}_j) \in N_{\tilde{e}_i}$ **do**
13  `comb2` $(\tilde{e}_i, \tilde{e}_j)$
14  `comb1` $(\tilde{e}_i)$
15 **return** *M*
16 **Procedure** `insert` $(e_i = (\tilde{e}_i, t_i))$
17  **if** $\tilde{e}_i \notin V_Q$ **then**
18  $V_Q \leftarrow V_Q \cup \{\tilde{e}_i\}$
19  $N_{\tilde{e}_i} \leftarrow \{\tilde{e} : \tilde{e} \in V_Q \setminus \{\tilde{e}_i\}$ and $\tilde{e}_i \cap \tilde{e} \neq \varnothing\}$
20  $E_Q \leftarrow E_Q \cup \{(\tilde{e}_i, \tilde{e}) : \tilde{e} \in N_{\tilde{e}_i}\}$
21  $t_Q(\tilde{e}_i) \leftarrow t(\tilde{e}_i) \cup \{t_i\}$
22 **Procedure** `remove` $(e_i = (\tilde{e}_i, t_i))$
23  $t(\tilde{e}_i) \leftarrow t(\tilde{e}_i) \setminus \{t_i\}$
24  **if** $t_Q(\tilde{e}_i) = \varnothing$ **then**
25  $V_Q \leftarrow V_Q \setminus \{\tilde{e}_i\}$
26  $N_{\tilde{e}_i} \leftarrow \{\tilde{e} : \tilde{e} \in V_Q$ and $e_i \cap e \neq \varnothing\}$
27  $E_Q \leftarrow E_Q \setminus \{(\tilde{e}_i, \tilde{e}) : \tilde{e} \in N_{\tilde{e}_i}\}$
28 **Procedure** `comb3` $(\tilde{e}_i, \tilde{e}_j, \tilde{e}_k)$
29  $M[h(\tilde{e}_j, \tilde{e}_k, \tilde{e}_i)]$ += $\sum_{t \in t_Q(\tilde{e}_j), t' \in t_Q(\tilde{e}_k)} \mathbb{1}[t < t']$
30  $M[h(\tilde{e}_k, \tilde{e}_j, \tilde{e}_i)]$ += $\sum_{t \in t_Q(\tilde{e}_j), t' \in t_Q(\tilde{e}_k)} \mathbb{1}[t' < t]$
31 **Procedure** `comb2` $(\tilde{e}_i, \tilde{e}_j)$
32  $M[h(\tilde{e}_i, \tilde{e}_j, \tilde{e}_i)]$ += $\sum_{t \in t_Q(\tilde{e}_i) \setminus \{t_i\}, t' \in t_Q(\tilde{e}_j)} \mathbb{1}[t < t']$
33  $M[h(\tilde{e}_j, \tilde{e}_i, \tilde{e}_i)]$ += $\sum_{t \in t_Q(\tilde{e}_i) \setminus \{t_i\}, t' \in t_Q(\tilde{e}_j)} \mathbb{1}[t' < t]$
34  $M[h(\tilde{e}_j, \tilde{e}_j, \tilde{e}_i)]$ += $\binom{|t_Q(\tilde{e}_j)|}{2}$
35 **Procedure** `comb1` $(\tilde{e}_i)$
36  $M[h(\tilde{e}_i, \tilde{e}_i, \tilde{e}_i)]$ += $\binom{|t_Q(\tilde{e}_i) - \{t_i\}|}{2}$

The update schemes of $Q$, `insert` and `remove` in Algorithm 2 add or delete nodes and their adjacent edges, respectively. More specifically, in `insert`, given a new temporal hyperedge $e_i = (\tilde{e}_i, t_i)$, its set of nodes $\tilde{e}_i$ is inserted as a new node, only if there do not exist any temporal hyperedges in the current temporal window whose nodes are $\tilde{e}_i$ (line 18). Once the new node is inserted, their incident edges are created as well (lines 19-20). Finally, the timestamp $t_i$ is added in $t_Q(\tilde{e}_i)$ (line 21). In `remove`, given a temporal hyperedge $e_i$ to be

removed, it first deletes its timestamp $t_i$ from $t_Q(\tilde{e}_i)$ (line 23). If the $e_i$ is the only temporal hyperedge in the current window whose node set is $\tilde{e}_i$, then $\tilde{e}_i$ and its incident edges are removed from $V_Q$ and $E_Q$, respectively (lines 25-27).

**Counting in THyME$^+$:** The counting procedure of THyME$^+$ is described in Algorithm 2. The sets of nodes and edges of the projected graph $Q$ are initialized to empty maps, i.e., $V_Q = \varnothing$ and $E_Q = \varnothing$ (line 2). For each temporal hyperedge $e_i = (\tilde{e}_i, t_i)$, it moves the temporal window to $[t_i - \delta, t_i]$ and accordingly as described above. Once $Q$ is updated, THyME$^+$ counts the instances of TH-motifs that contains $e_i$ and the previous temporal hyperedges. To minimize enumerations, THyME$^+$ adapts effective counting schemes, `comb3`, `comb2`, and `comb1`, which compute the number of instances of triple-inducing, pair-inducing, and single-inducing TH-motifs, respectively, as follows:

- **Triple-inducing TH-motifs (lines 9-11):** THyME$^+$ first enumerates the instances of three connected hyperedges in $Q$ such that contains $\tilde{e}_i$ (line 9). For each set $\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\}$ of three connected hyperedges, the number of instances of TH-motifs that contains $e_i$ is counted by timestamp combinations using `comb3` method. That is, since $e_i$ is the latest temporal hyperedge, the set $\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\}$ can be induced by sequences of either $\langle e_x, e_y, e_i\rangle$ or $\langle e_y, e_x, e_i\rangle$ where $\tilde{e}_x = \tilde{e}_j$ and $\tilde{e}_y = \tilde{e}_k$. Since $t_x \in t_Q(\tilde{e}_j)$ and $t_y \in t_Q(\tilde{e}_k)$, the number of such instances can be computed by the number of timestamp combinations of $t_Q(\tilde{e}_j)$ and $t_Q(\tilde{e}_k)$ (lines 29-30).
- **Pair-inducing TH-motifs (lines 12-13):** THyME$^+$ enumerates each edge $(\tilde{e}_i, \tilde{e}_j)$ in $Q$ that are adjacent to $\tilde{e}_i$, which can be induced by three different orders of sequences, $\langle e_x, e_y, e_i\rangle$, $\langle e_y, e_x, e_i\rangle$, and $\langle e_y, e_y, e_i\rangle$ where $\tilde{e}_x = \tilde{e}_i$ and $\tilde{e}_y = \tilde{e}_j$. Since $t_x \in t_Q(\tilde{e}_i) \setminus \{t_i\}$ and $t_y \in t_Q(\tilde{e}_j)$, the number of the sequences can be computed by the number of combinations of the set of these timestamps (lines 32-33).
- **Single-inducing TH-motifs (line 14):** Single-inducing TH-motif, which consists of three duplicated temporal hyperedges, can be immediately counted using `comb1`. That is, a sequence $\langle e_x, e_y, e_i\rangle$ where $\tilde{e}_x = \tilde{e}_i$ and $\tilde{e}_y = \tilde{e}_i$ can be an instance of single-inducing TH-motif. Since $t_x \in t_Q(\tilde{e}_i) \setminus \{t_i\}$, $t_y \in t_Q(\tilde{e}_i) \setminus \{t_i\}$, and $t_x < t_y$, the number of such instances is computed immediately (line 36).

In Section V-E, we share empirical observations supporting the intuition behind THyME$^+$. In addition, we provide the complexity analysis of THyME$^+$ in the supplementary document [29].

## V. Empirical Studies

In this section, we review experiments to answer Q1-Q4.

Q1. **Discoveries:** Which findings do TH-motifs bring?
Q2. **Comparison with Static H-motifs:** Are TH-motifs more informative than static hypergraph motifs [6]?
Q3. **Speed & Efficiency:** How fast and efficient is THyME$^+$?
Q4. **Further Analysis:** Why is THyME$^+$ fast and efficient?

TABLE II: Statistics of the 11 real-world hypergraphs from 5 different domains: the number of nodes $|V|$, the number of temporal hyperedges $|\mathcal{E}|$, the number of induced static hyperedges $|E_{\mathcal{E}}|$, and the maximum hyperedge size $\max_{e\in\mathcal{E}}|e|$.

| Dataset | $\lvert V\rvert$ | $\lvert\mathcal{E}\rvert$ | $\lvert E_{\mathcal{E}}\rvert$ | $\max_{e\in\mathcal{E}}\lvert e\rvert$ |
|---|---|---|---|---|
| email-Enron | 143 | 10,885 | 1,514 | 37 |
| email-Eu | 986 | 235,263 | 25,148 | 40 |
| contact-primary | 242 | 106,879 | 12,704 | 5 |
| contact-high | 327 | 172,035 | 7,818 | 5 |
| threads-ubuntu | 90,054 | 192,947 | 166,999 | 14 |
| threads-math | 153,806 | 719,792 | 595,749 | 21 |
| tags-ubuntu | 3,021 | 271,233 | 147,222 | 5 |
| tags-math | 1,627 | 822,059 | 170,476 | 5 |
| coauth-DBLP | 1,836,596 | 3,700,681 | 2,467,389 | 280 |
| coauth-Geology | 1,091,979 | 1,591,166 | 1,204,704 | 284 |
| coauth-History | 503,868 | 1,813,147 | 896,062 | 925 |

We first describe the settings where the experiments are conducted. Then, we provide some empirical observations using the proposed concepts and algorithms. Next, we test the scalability of the methods. Finally, we provide possible reasons why THyME$^+$ is efficient based on the observations on real-world temporal hypergraphs.

### A. Experimental Settings

**Machines:** We conducted all the experiments on a machine with i9-10900K CPU and 64GB RAM.

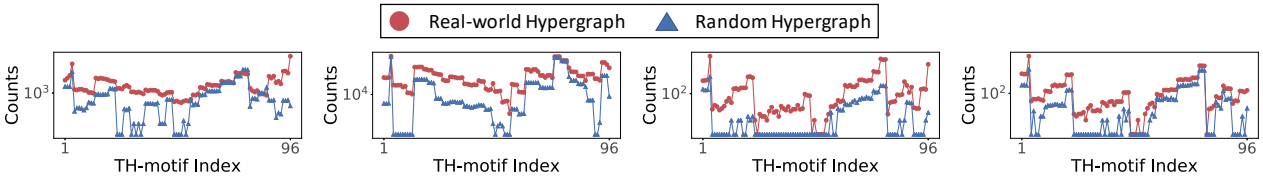
**Implementation:** We implemented DP, THyME, and THyME$^+$ commonly in C++.

**Datasets:** We use eleven real-world temporal hypergraphs from five different domains. Refer to Table II for the summarized statistics of the hypergraphs. We provide the details of each dataset in Appendix B. While we assume that timestamps of temporal hyperedges are unique, in some dataset, this may not hold. In such cases, we randomly order the temporal hyperedges whose timestamps are identical.

### B. Q1. Discoveries

In this subsection, we present several observations that TH-motifs reveal in the 11 real-world hypergraphs. TH-motifs provide a new perspective in analyzing temporal hypergraphs.

**Obs 1. Real hypergraphs are not 'random':** For an accurate characterization, we compare the number of instances of TH-motifs in real-world temporal hypergraphs against that in randomized ones. To this end, we randomize the real-world temporal hypergraph using HyperCL [3], a random hypergraph generator which preserves node degrees and hyperedge sizes. Once the randomized hypergraph is generated, we randomly assign the timestamps of its temporal hyperedges. In Fig. 4, we compare the distribution of the number of instances of each TH-motif in real-world temporal hypergraphs and those in randomized ones. The distributions are clearly different, and the total number of instances is greater in real-world hypergraphs than in random hypergraphs. Specifically, the total number of TH-motifs' instances in real-world hypergraphs are $6.42\times$, $1.44\times$, $46.69\times$, $4.30\times$ of that in randomized hypergraphs




(a) email-Eu (b) contact-primary (c) threads-math (d) tags-ubuntu

Fig. 4: The distribution of the number of TH-motifs' instances in real-world temporal hypergraphs and that in randomized temporal hypergraphs are significantly different. We set $\delta$ to 1 hour. We do not directly compare the distributions from the coauthorship datasets since their timestamp units are years. We provide the distributions in [29].

in email-Eu, contact-primary, threads-math, and tags-ubuntu, respectively.

**Obs 2. TH-motifs distinguish domains:** Network motifs have demonstrated their power to distinguish graphs based on their domains. In addition, the count distributions of h-motifs in static hypergraphs are particularly similar between domains but different across domains. To confirm that temporal h-motifs also possess such distinguishing power, we obtain the characteristic profile (CP) of each hypergraph, a normalized 96 dimensional vector of concatenation of relative significance of each temporal h-motif, as suggested in [6]. As seen in Fig. 5, CPs accurately capture patterns of real-world temporal hypergraphs. That is, while CPs of the temporal hypergraphs from the same domain are similar, they are different across domains. These results support that TH-motifs play a key role in capturing structural and temporal patterns of real-world temporal hypergraphs.

**Obs 3. Orders of hyperedges matter:** TH-motifs are asymmetric with respect to the arrival order of the temporal hyperedges, and thus instances that are indistinguishable with static h-motifs can be categorized as different TH-motifs. We are interested in how the orders of the hyperedges affect the occurrences of TH-motifs, and to this end, we statistically investigate nine pair-inducing ones, ranging from TH-motif 87 to 95. TH-motifs in each triple, TH-motifs $87-89$, TH-motifs $90-92$, and TH-motifs $93-95$ share the same structural pattern and are distinguished by the orders of the hyperedges. Consider an instance $\langle e_i, e_j, e_k \rangle$ of the pair-inducing TH-motif. The pair-inducing TH-motifs, by definition, consist of a pair of duplicated hyperedges and thus enables three different orderings **O1**: $\tilde{e}_i = \tilde{e}_j \neq \tilde{e}_k$, **O2**: $\tilde{e}_i \neq \tilde{e}_j = \tilde{e}_k$, and **O3**: $\tilde{e}_i \neq \tilde{e}_j \neq \tilde{e}_k$, $\tilde{e}_i = \tilde{e}_k$,. In **O1** and **O2**, duplicated temporal hyperedges occur consecutively, whereas in **O3**, the first and last hyperedges are duplicated. TH-motifs 87, 90, and 93 are **O1**, TH-motifs 88, 91, and 94 are **O2**, and TH-motifs 89, 92, and 95 are **O3**. As seen in Fig. 6, this difference indeed affect the occurrences of the TH-motifs in real-world temporal hypergraphs. The ratio of the TH-motifs whose ordering is **O3** are significantly small, compared to that of **O1** and **O2**. That is, duplicated temporal hyperedges tend to occur in a short time and thus affect the count distributions of TH-motifs.

### C. Q2. Comparison with Static H-motifs

In this subsection, we demonstrate the usefulness of TH-motifs. We compare TH-motifs and static h-motifs as inputs features for a hyperedge prediction task.

**Obs 4. TH-motifs help predict future hyperedges:** To verify the usefulness of temporal h-motifs, we consider the problem of hyperedge prediction, a binary classification problem of predicting whether the given hyperedge is true or not. Given a temporal hypergraph $T = (V, \mathcal{E})$, we generate a set $\mathcal{E}'$ of fake hyperedges, whose size is equal to the true one (i.e., $|\mathcal{E}| = |\mathcal{E}'|$), using HyperCL [3], which preserves the degrees of the nodes and the sizes are equal to the true ones. The timestamps of the fake hyperedges are randomly assigned. We sort the entire temporal hyperedges $\mathcal{E} \cup \mathcal{E}'$ based on their timestamps and split into train and test sets in a ratio 8:2. Then we train a logistic regression classifier using the train set with following three different features of each temporal hyperedge:

- **THM96** ($\in \mathbb{R}^{96}$)**:** Each dimension represents the number of instances of TH-motifs that contain the hyperedge.
- **THM26** ($\in \mathbb{R}^{26}$)**:** The 26 TH-motifs whose occurrences have the highest variance are selected.
- **SHM26** ($\in \mathbb{R}^{26}$)**:** Each dimension represent the number of instances of static h-motifs that contain the hyperedge. Temporal information is ignored.

As seen in Fig. 7, THM96 and THM26, which are based on the TH-motifs counts, are more accurate than STM26. While h-motifs only represent structural patterns, TH-motifs incorporate temporal information in addition to them, and thus they are more informative.

### D. Q3. Speed and Efficiency

We evaluate the speed and efficiency of the proposed algorithms DP, THyMe, and THyMe$^+$. As seen in Fig. 8, while DP and THyMe run out of memory in some datasets or with particular $\delta$ values, THyMe$^+$ is fast and space efficient enough in all considered settings. Specifically, THyMe$^+$ is up to $2,163\times$ faster than DP and $16\times$ faster than THyMe. As described in Section IV, THyMe$^+$ maintains a small projected graph $Q$ and thus reduces enumeration over the instances in $Q$. In the next subsection, we provide empirical findings that support the effectiveness of THyMe$^+$.

### E. Q4. Further Analysis

Why is THyMe$^+$ faster and more space efficient compared to DP and THyMe? What properties of real-world temporal hypergraphs make THyMe$^+$ efficient? To answer these questions, we examine structural and temporal patterns of temporal

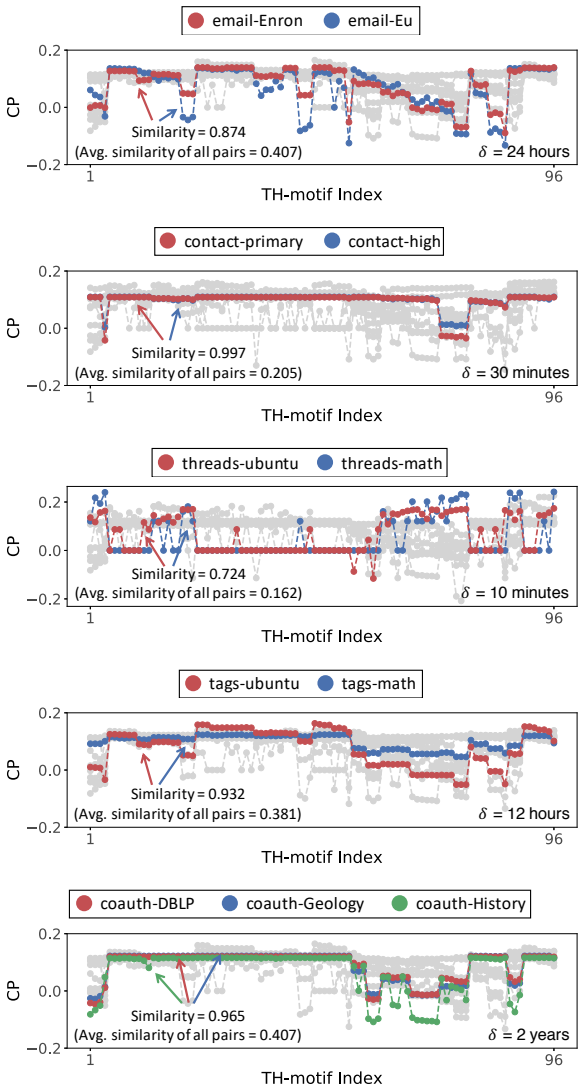
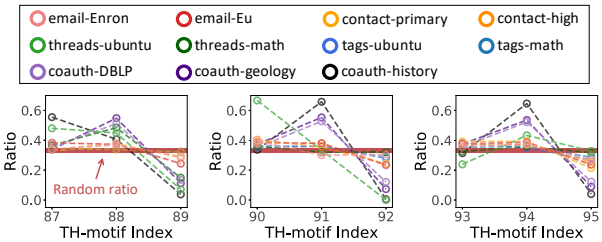



Fig. 5: Characteristic profiles (CPs) (i.e., normalized significance of each TH-motif) accurately capture the patterns of real-world temporal hypergraphs. The CPs of the temporal hypergraphs from the same domain are similar in terms of the Pearson correlation coefficients, which are the reported numbers, while they are different across domains. Grey lines indicate CPs of the temporal hypergraphs from other domains.

hyperedges in real-world temporal hypergraphs and summarize common properties observed as follows.

- **(Obs. 5) Repetitive behavior:** Duplicated temporal hyperedges tend to appear repeatedly, and the distribution of the numbers of repetitions is heavy-tailed.
- **(Obs. 6) Temporally locality:** Future temporal hyperedges are more likely to repeat recent hyperedges than older ones.

**Obs. 5. Repetitive behavior:** We first investigate the repeating patterns (i.e., duplication) of temporal hyperedges in real-world temporal hypergraphs. As seen in Table II, the




(a) TH-motifs 87-89 (b) TH-motifs 90-92 (c) TH-motifs 93-95

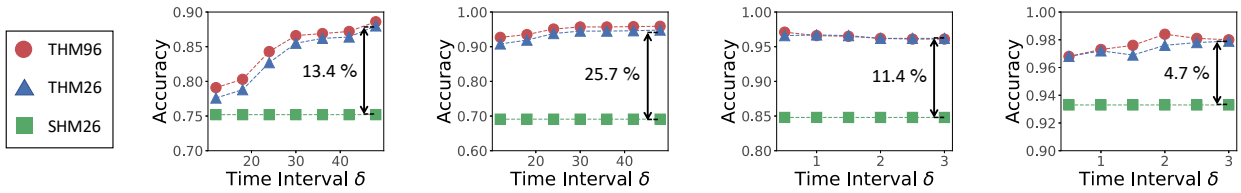
Fig. 6: The number of instances of nine pair-inducing TH-motifs depend on the ordering of the hyperedges. The ratio of the occurrences of TH-motifs 89, 92, and 95 are significantly low compared to the other TH-motifs with same structures.

number of induced hyperedges ($|E_{\mathcal{E}}|$) is significantly smaller than that of temporal hyperedges ($|\mathcal{E}|$), implying that temporal hyperedges are frequently repeated. Surprisingly, in `contact-high` dataset, the number of induced hyperedges is only 4.5% of that of temporal hyperedges, implying that most temporal hyperedges consist of predefined set of nodes. Note that due to the flexibility of hyperedge sizes, a hyperedge can be generated from $O(2^{|V|})$, and thus is extremely unlikely to repeat the exact set of nodes. In addition, we discover that the distributions of hyperedge repetitions in real-world temporal hypergraphs are generally heavy-tailed and close to power-law distributions, as seen in Fig. 9. We support this claim by fitting the distributions to representative heavy-tailed distributions in [29].

**Obs. 6. Temporal locality:** Now that we have observed the structural behaviors of the temporal hyperedges, we turn our attention to the temporal aspect. The temporal locality of temporal hyperedges is the tendency that recent hyperedges are more likely to be repeated in the near future than the older ones. To show the temporal locality, we investigate the time intervals of the $N$ consecutive identical temporal hyperedges, i.e., the time it takes for a hyperedge to be repeated $N$ times. Fig. 10 shows the average time intervals of all the hyperedges in the real-world hypergraphs and randomly shuffled hypergraphs, where timestamps of the hyperedges are randomly shuffled while preserving the underlying structure. In every dataset, the time intervals within $N$ consecutive hyperedges are shorter in real-world hypergraphs than in randomized ones. That is, future hyperedges are more likely to repeat the recent hyperedges than older ones.

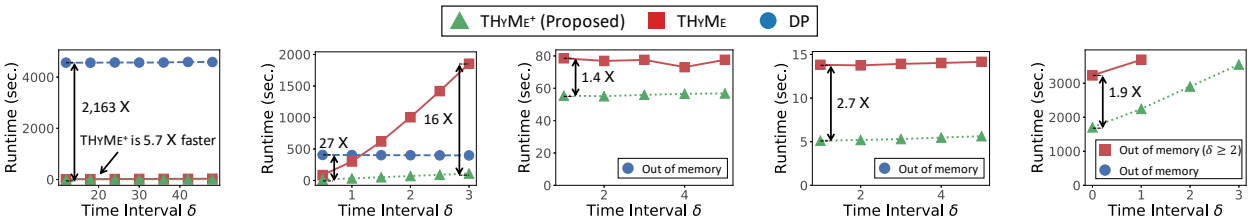
**Intuition behind THyMe$^+$:** How do these properties of real-world temporal hypergraphs provide efficiency to THyMe$^+$? Here, we provide some reasons why we expect THyMe$^+$ to be faster and more space-efficient than THyMe and DP.

- **Connection to Obs. 5:** Each node in the projected graph $P$ used in THyMe represents a unique temporal hyperedge, and its size heavily depends on $\delta$. On the other hand, the nodes in the projected graph $Q$ maintained in THyMe$^+$ represent induced hyperedges, and several temporal hyperedges can share the same node. Thus, more repetitions of temporal hyperedges provide higher efficiency of THyMe$^+$,



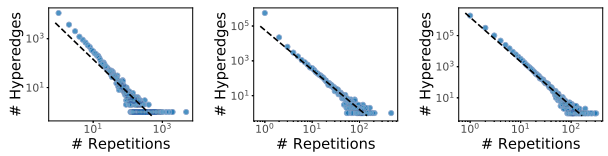

(a) email-Enron (b) email-Eu (c) contact-primary (d) contact-high

Fig. 7: TH-motifs provide informative features of temporal hyperedges. THM96 and THM26, which use the counts of TH-motifs' instance as features, are more accurate than SHM26, which uses the counts of static h-motifs' instances, in predicting future temporal hyperedges. Results in small datasets where the instances of static h-motifs can be exactly counted are reported.



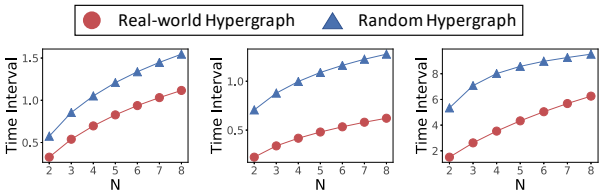

(a) email-Eu (b) contact-primary (c) threads-math (d) tags-ubuntu (e) coauth-DBLP

Fig. 8: THyMe+ is faster and more space efficient than DP and THyMe. We provide the full results in [29].




(a) email-Eu (b) threads-math (c) coauth-DBLP

Fig. 9: Temporal hyperedges in real-world hypergraphs are repetitive. Temporal hyperedges appear repetitively and the number of repetitions follow a near power-law distribution. This tendency is found consistently across all datasets [29].




(a) email-Eu (b) threads-math (c) coauth-DBLP

Fig. 10: Temporal hyperedges in real-world hypergraphs are temporally local. The time intervals of $N$ consecutive duplicated temporal hyperedges is shorter in real-world temporal hypergraphs than in randomized hypergraphs. The units of time intervals in coauth-DBLP is years, and the others are hours. We provide the full results in [29].

as observed in real-world temporal hypergraphs.

- **Connection to Obs. 6:** The benefits of temporal locality of temporal hyperedges are two-fold: (1) The tendency of temporal hyperedges to repeat within a short period of time indicates that duplicated temporal hyperedges are more likely to co-appear in the temporal window in THyMe+, which reduces the size of the projected graph $Q$. (2) If duplicated temporal hyperedges reappear within the temporal window, insertion/deletion of nodes and edges of $Q$ are skipped, which is beneficial in terms of speed.

## VI. Conclusion

In this work, we propose (a) temporal hypergraph motifs (TH-motifs), which are tools for analyzing design principles of time-evolving hypergraphs, and (b) THyMe+, which is a fast algorithm for exactly counting TH-motifs' instances. Using them, we investigate 11 real-world hypergraphs from 5 domains. Our contributions are summarized as follows.

- **New concept:** We define 96 temporal hypergraph motifs (TH-motifs) that describe local relational and temporal dynamics in time-evolving hypergraphs.
- **Fast and exact algorithm:** We develop THyMe+, a fast and exact algorithm for counting the instances of TH-motifs. It is at most $2,163\times$ faster than the baseline approach.
- **Empirical discoveries:** TH-motifs reveal interesting structural and temporal patterns in real-world hypergraphs. TH-motifs also provide informative features that are useful in predicting future hyperedges.

**Reproducibility:** The source code and datasets used in this work are available at https://github.com/geonlee0325/THyMe.

**Acknowledgements:** This work was supported by National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2020R1C1C1008296) and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2019-0-00075, Artificial Intelligence Graduate School Program (KAIST)).

## References

[1] Y. Kook, J. Ko, and K. Shin, "Evolution of real-world hypergraphs: Patterns and models without oracles," in *ICDM*, 2020.
[2] M. T. Do, S.-e. Yoon, B. Hooi, and K. Shin, "Structural patterns and generative models of real-world hypergraphs," in *KDD*, 2020.
[3] G. Lee, M. Choe, and K. Shin, "How do hyperedges overlap in real-world hypergraphs?–patterns, measures, and generators," in *WWW*, 2021.
[4] A. R. Benson, R. Kumar, and A. Tomkins, "Sequences of sets," in *KDD*, 2018.

[5] A. R. Benson, R. Abebe, M. T. Schaub, A. Jadbabaie, and J. Kleinberg, "Simplicial closure and higher-order link prediction," *Proceedings of the National Academy of Sciences*, vol. 115, no. 48, pp. E11 221–E11 230, 2018.
[6] G. Lee, J. Ko, and K. Shin, "Hypergraph motifs: concepts, algorithms, and discoveries," *PVLDB*, vol. 13, pp. 2256–2269, 2020.
[7] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, "Network motifs: simple building blocks of complex networks," *Science*, vol. 298, no. 5594, pp. 824–827, 2002.
[8] S.-e. Yoon, H. Song, K. Shin, and Y. Yi, "How much and when do we need higher-order information in hypergraphs? a case study on hyperedge prediction," in *WWW*, 2020.
[9] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *WSDM*, 2017.
[10] Y. Li, Z. Lou, Y. Shi, and J. Han, "Temporal motifs in heterogeneous information networks," in *MLG Workshop*, 2018.
[11] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki, "Temporal motifs in time-dependent networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, no. 11, p. P11005, 2011.
[12] S. Gurukar, S. Ranu, and B. Ravindran, "Commit: A scalable approach to mining communication motifs from dynamic networks," in *SIGMOD*, 2015.
[13] U. Redmond and P. Cunningham, "Temporal subgraph isomorphism," in *ASONAM*, 2013.
[14] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon, "Network motifs in the transcriptional regulation network of escherichia coli," *Nature Genetics*, vol. 31, no. 1, pp. 64–68, 2002.
[15] R. Milo, S. Itzkovitz, N. Kashtan, R. Levitt, S. Shen-Orr, I. Ayzenshtat, M. Sheffer, and U. Alon, "Superfamilies of evolved and designed networks," *Science*, vol. 303, no. 5663, pp. 1538–1542, 2004.
[16] R. A. Rossi, N. K. Ahmed, A. Carranza, D. Arbour, A. Rao, S. Kim, and E. Koh, "Heterogeneous graphlets," *ACM TKDD*, vol. 15, no. 1, pp. 1–43, 2020.
[17] S. P. Borgatti and M. G. Everett, "Network analysis of 2-mode data," *Social networks*, vol. 19, no. 3, pp. 243–269, 1997.
[18] A. R. Benson, D. F. Gleich, and J. Leskovec, "Higher-order organization of complex networks," *Science*, vol. 353, no. 6295, pp. 163–166, 2016.
[19] P.-Z. Li, L. Huang, C.-D. Wang, and J.-H. Lai, "Edmot: An edge enhancement approach for motif-aware community detection," in *KDD*, 2019.
[20] C. E. Tsourakakis, J. Pachocki, and M. Mitzenmacher, "Scalable motif-aware graph clustering," in *WWW*, 2017.
[21] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *KDD*, 2017.
[22] A. Arenas, A. Fernandez, S. Fortunato, and S. Gomez, "Motif-based communities in complex networks," *Journal of Physics A: Math Theor.*, vol. 41, no. 22, p. 224001, 2008.
[23] H. Zhao, X. Xu, Y. Song, D. L. Lee, Z. Chen, and H. Gao, "Ranking users in social networks with higher-order structures," in *AAAI*, 2018.
[24] Y. Yu, Z. Lu, J. Liu, G. Zhao, and J.-r. Wen, "Rum: Network representation learning using motifs," in *ICDE*, 2019.
[25] R. A. Rossi, N. K. Ahmed, and E. Koh, "Higher-order network representation learning," in *WWW Companion*, 2018.
[26] R. A. Rossi, N. K. Ahmed, E. Koh, S. Kim, A. Rao, and Y. Abbasi-Yadkori, "A structural graph representation learning framework," in *WSDM*, 2020.
[27] J. B. Lee, R. A. Rossi, X. Kong, S. Kim, E. Koh, and A. Rao, "Graph convolutional networks with motif-based attention," in *CIKM*, 2019.
[28] R. A. Rossi, R. Zhou, and N. K. Ahmed, "Deep inductive graph representation learning," *IEEE TKDE*, vol. 32, no. 3, pp. 438–452, 2018.
[29] "Online appendix," 2021. [Online]. Available: https://github.com/geonlee0325/THyMe/blob/main/supplements.pdf

## APPENDIX

### A. Details of DYNAMIC PROGRAMMING (DP)

The procedure `count` (lines 9-19) counts the instances of TH-motifs that induce a set of $\ell$ connected static hyperedges. That is, given a set $s = \{\tilde{e}_1, \ldots, \tilde{e}_\ell\}$ of $\ell$ connected static hyperedges, `count` first constructs a time-sorted sequence $e(s)$ of temporal hyperedges whose nodes is one of $s$ (line 10). It also introduces a map $C$ that maintains the counts of ordered hyperedges of length at most $\ell$. Then `count` scans through the temporal hyperedges in $e(s)$ and tracks the subsequences that occur within the temporal window that spans temporal hyperedges within $\delta$ time units. As the temporal window slides through the temporal hyperedges $e(s)$, the count of the sequences are computed based on the subsequences counted in $C$. Refer to [9] for more intuition behind this dynamic programming formulation.

**Algorithm 3:** DP: Preliminary Algorithm for Exact Counting of TH-motifs' Instances

**Input :** (1) temporal hypergraph: $T = (V, \mathcal{E})$
(2) time interval $\delta$
**Output:** # of each temporal h-motif $t$'s instances: $M[t]$

1 $S \leftarrow$ set of instances of static h-motifs in $G_T$
2 **for each** instance $\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\} \in S$ **do**
3  `count` $(\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\})$
4 **for each** pair of overlapping hyperedges $\{\tilde{e}_i, \tilde{e}_j\} \in \wedge_{\mathcal{E}}$ **do**
5  `count` $(\{\tilde{e}_i, \tilde{e}_j\})$
6 **for each** hyperedge $\tilde{e}_i \in E_{\mathcal{E}}$ **do**
7  `count` $(\{\tilde{e}_i\})$
8 **return** $M$
9 **Procedure** `count` $(s = \{\tilde{e}_1, \ldots, \tilde{e}_\ell\})$
10  $e(s) \leftarrow$ sorted$(I(\tilde{e}_1) \cup \cdots \cup I(\tilde{e}_\ell))$
11  $w_s \leftarrow 1$
12  $C \leftarrow$ map initialized to 0
13  **for each** temporal hyperedge $e_i = (\tilde{e}_i, t_i) \in e(s)$ **do**
14  **while** $t_{w_s} + \delta < t_i$ **do**
15  `decrement` $(\tilde{e}_{w_s})$
16  $w_s \leftarrow w_s + 1$
17  `increment` $(\tilde{e}_i)$
18  **for each** $\langle e_i, e_j, e_k \rangle \in$ permutations$(\{\tilde{e}_i, \tilde{e}_j, \tilde{e}_k\})$ **do**
19  $M[h(\tilde{e}_i, \tilde{e}_j, \tilde{e}_k)]$ += $C[$concat$(\tilde{e}_i, \tilde{e}_j, \tilde{e}_k)]$
20 **Procedure** `increment` $(\tilde{e})$
21  **for each** prefix **in** $C$.keys.reverse of length $< \ell$ **do**
22  $C[$concat(prefix, $\tilde{e})]$ += $C[$prefix$]$
23  $C[\tilde{e}] \leftarrow C[\tilde{e}] + 1$
24 **Procedure** `decrement` $(\tilde{e})$
25  $C[\tilde{e}] \leftarrow C[\tilde{e}] - 1$
26  **for each** suffix **in** $C$.keys of length $< \ell - 1$ **do**
27  $C[$concat$(\tilde{e},$ suffix$)]$ -= $C[$suffix$]$

### B. Details of Datasets

We provide the details of the eleven real-world temporal hypergraphs from the following five distinct domains:

- **email**: Each node is an email account and each hyperedge is the set of sender and receivers of the email.
- **contact**: Each node is a person and each hyperedge is a group interaction among people.
- **threads**: Each node is a user and each hyperedge is a group of users working in a thread.
- **tags**: Each node is a tag and each hyperedge is a set of tags attached to the question.
- **coauthorship**: Each node is an author and each hyperedge is a set of authors of the publication.