

LSCPM: communities in massive real-world Link Streams by Clique Percolation Method

Alexis Baudin ✉🏠

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Lionel Tabourier ✉🏠

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Clémence Magnien ✉🏠

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

Abstract

Community detection is a popular approach to understand the organization of interactions in static networks. For that purpose, the Clique Percolation Method (CPM), which involves the percolation of k -cliques, is a well-studied technique that offers several advantages. Besides, studying interactions that occur over time is useful in various contexts, which can be modeled by the link stream formalism. The Dynamic Clique Percolation Method (DCPM) has been proposed for extending CPM to temporal networks.

However, existing implementations are unable to handle massive datasets. We present a novel algorithm that adapts CPM to link streams, which has the advantage that it allows us to speed up the computation time with respect to the existing DCPM method. We evaluate it experimentally on real datasets and show that it scales to massive link streams. For example, it allows to obtain a complete set of communities in under twenty-five minutes for a dataset with thirty million links, what the state of the art fails to achieve even after a week of computation. We further show that our method provides communities similar to DCPM, but slightly more aggregated. We exhibit the relevance of the obtained communities in real world cases, and show that they provide information on the importance of vertices in the link streams.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms

Keywords and phrases Temporal network, Link stream, k -clique, Community detection, Clique Percolation Method, Real-world interactions

Acknowledgements This work is funded by the ANR (French National Agency of Research) through the ANR FiT LabCom. The authors thank the SocioPatterns¹ and Icon² communities, for making their datasets available.

1 Introduction

Detecting communities in complex networks has been a focus of interest since the early years of the field to the point that even the number of surveys on the topic is large. While the first ones aimed at giving a general view of the landscape (*e.g.*, [9]), more recent ones tend to focus on a particular issue, for instance the underlying purpose of the community detection [24] or a specific family of networks [16].

The Clique Percolation Method (CPM) proposed by Palla *et al.* [20] is a well studied technique to detect communities in a graph. It is appreciated for the advantages that its definition confers: the communities are defined locally and in a deterministic way and there is no need to use heuristics or optimization functions that are hard to interpret. Also, it allows the communities to overlap each other, by contrast with most other techniques

¹ <http://www.sociopatterns.org>

² <https://icon.colorado.edu>

which lead to a partition of the nodes. It is a desirable property in general as the frontier between communities is often difficult to decide. While the early implementations were computationally costly, it was later improved [14, 22] and the most recent one scales up to graphs with hundreds of millions of nodes and edges [2].

Considering temporal networks, where the structure evolves dynamically, a standard approach consists in examining them as a sequence of snapshots and run graph community detection algorithms on each of them. Then, it makes sense to match the communities obtained from a time step to the next to obtain consistent groups through time [1]. Another approach following the same purpose is to design communities that ensure cohesive structure continuity over a time interval [25]. This strategy has been investigated for various applications, including mobile communication networks [18] or social networks analysis [23]. However, the instability of many community detection methods makes this task hard to achieve properly [6]. In addition, some works stress the importance of achieving online community detection, in which case the communities are updated at each time step, by aggregating new information to the existing communities [7, 21]. If the method is fast enough, it is possible to achieve a streaming community analysis of the data. However, this type of method often comes at the cost of losing part of the long term meaning that a community might have.

CPM can be implemented on graph snapshots and avoid instabilities from one step to the next due to its deterministic nature; it thus appears as a good candidate for the approach described above. It was in fact proposed quite early on, in [19]; we refer to this approach as the *Dynamic Clique Percolation Method* (DCPM). Recently, Boudebza *et al.* [4] introduced a faster algorithm to do this, called *Online Clique Percolation Method* (OCPM). However, describing a temporal network as a sequence of snapshots has shortcomings. Indeed, it misses the fact that the time step of analysis chosen is frequently arbitrary, while in general the data do not exhibit an obvious timescale of analysis. This is why formalisms to describe temporal networks have been developed to circumvent these limitations [5, 11, 15]. Among those, the link stream formalism stresses the symmetric roles of structure and time in the representation of data, and aim at describing temporal networks at the intersection of graph theory and time series analysis [15]. As the notion of clique has been recently extended to this formalism and the search for cliques is implemented efficiently [3, 28], it is now possible to investigate extensions of CPM to link streams. This makes possible to design faster algorithms than the state-of-the-art DCPM implementations. In this way, we obtain a community detection technique that can process data online and maintain relevant communities which naturally spread through time.

This is the main contribution of this paper: we propose an algorithm for this goal as well as an open source implementation that scales to large link streams³. In Section 2, we give the necessary background definitions and notations. Then, we describe our method in Section 3 and derive an expression of its theoretical complexity; note that this method uses a novel k -clique enumeration algorithm in link streams. Finally, we provide in Section 4 an extensive experimental investigation which shows its efficiency on several real-world instances, compares the obtained communities to the DCPM ones, and illustrates its relevance to draw information about the data examined.

³ <https://gitlab.lip6.fr/baudin/lscpm>

2 Definitions and notations

First, we start with some basic reminders about (static) graphs. A graph is a pair $G = (V, E)$, where V is a set of vertices, and E is a set of edges of the form $\{u, v\}$, where $u, v \in V$ and $u \neq v$. A k -clique in G is a set of k vertices that are all connected pairwise by an edge. Finally, the definition of CPM communities is given by the fact that two k -cliques are adjacent when they have $k - 1$ vertices in common. Then, a CPM community of G is the set of vertices belonging to a maximal set of k -cliques that can be reached from one to another by a series of adjacent k -cliques (it forms a connected component of the graph whose vertices are the k -cliques of G and edges are defined by the adjacency relation just explained).

2.1 Cliques in link streams

In this article, we work with the link stream model, which represents interactions over time. Formally, a *link stream* is a triplet $L = (T, V, E)$ where T is a time interval, V a set of vertices and $E \subseteq T \times T \times V \times V$ a set of links (b, e, u, v) such that $e \geq b$; we call $e - b$ the duration of such a link. Throughout the paper, we consider link streams with no self-loop, *i.e.* for any link $(b, e, u, v) \in E$, then $u \neq v$. Moreover, links on the same vertices exist over disjoint time intervals, *i.e.* if $(b, e, u, v), (b', e', u, v) \in E$, with $b \neq b'$ or $e \neq e'$, then $[b, e] \cap [b', e'] = \emptyset$.

We use the definition of a clique in a link stream which follows the one in [27], with a minor difference to avoid cliques over time intervals of null length: a *clique* is a pair $(C, [t_0, t_1])$, where $C \subseteq V$, $|C| \geq 2$ and $t_0, t_1 \in T$, $t_0 < t_1$, such that for all $u, v \in C$, $u \neq v$, there is a link (b, e, u, v) in E such that $[t_0, t_1] \subseteq [b, e]$. A *k-clique* is a clique containing k vertices. Notice that if $(C, [t_0, t_1])$ is a k -clique, then $(C, [t'_0, t'_1])$ is also a k -clique for all t'_0, t'_1 such that $t_0 \leq t'_0 < t'_1 \leq t_1$. We are therefore interested in maximal k -cliques:

► **Definition 1** (maximal k -clique). *For $k \in \llbracket 2, +\infty \rrbracket$, a maximal k -clique is a clique $(C, [t_0, t_1])$ having k vertices ($|C| = k$), and such that its time interval is maximal: there is no $t'_0 < t_0$ nor $t'_1 > t_1$ such that $(C, [t'_0, t'_1])$ or $(C, [t_0, t'_1])$ is a clique.*

With this definition, we can introduce the notion of k -clique adjacency, which will allow defining a generalization of CPM communities to link streams: two maximal k -cliques $(C, [t_0, t_1])$ and $(C', [t'_0, t'_1])$ are said to be *adjacent* if they share $k - 1$ vertices and overlap over a time interval with strictly positive length, *i.e.*, $|C \cap C'| = k - 1$ and $|[t_0, t_1] \cap [t'_0, t'_1]| > 0$, where $|I|$ denotes the length of interval I .

2.2 Communities in link streams

In a dynamical context, it is natural to define a *temporal community* as a set of *temporal vertices* of the form (u, I) , where u is a vertex, and I is a set of disjoint time intervals, which are the time intervals during which u is present in the community. Then, the notion of LSCPM community is similar to the one of CPM community in graphs, but with the notion of maximal k -clique adjacency adapted to link streams:

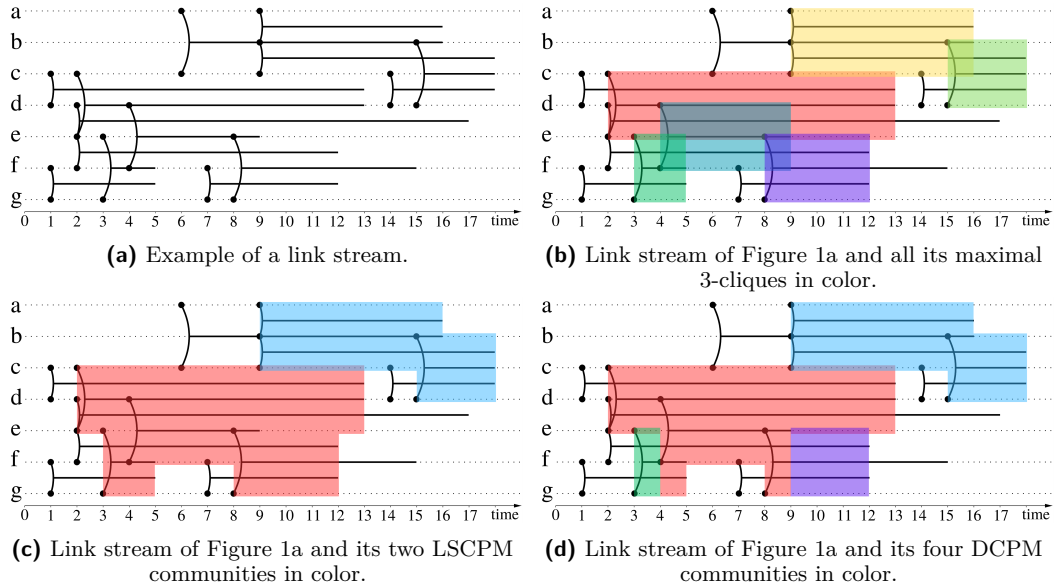
► **Definition 2** (LSCPM community). *A LSCPM community is a temporal community whose temporal vertices belong to a maximal set of k -cliques that can be reached from one to another by a series of adjacent k -cliques.*

A few observations can be made about this definition. First, as k increases, the communities may only split and/or lose temporal vertices. In other words, if $k_1 < k_2$, each

community computed with k_2 is included in a community computed with k_1 . This property is illustrated and further discussed in Section 4.5.

Second, let us recall the definition of the dynamic CPM (DCPM) communities introduced in [19]. The idea is to compute the CPM communities at each snapshot. Then, comparing the communities obtained for two consecutive snapshots allows finding whether each community evolves by gaining vertices, losing vertices, dying (disappearing or merging with a larger one), or being born (appearing or being detached from a larger one). Notice that DCPM communities can be considered as temporal communities: given two consecutive snapshots at t_i and t_{i+1} , a vertex of a CPM community at t_i belongs to the DCPM community on $[t_i, t_{i+1}[$. Most importantly, a LSCPM community is a union of DCPM communities. Indeed, the temporal vertices of a CPM community in a snapshot are all included in a same LSCPM community; and a CPM community that gains or loses vertices from one snapshot to the next remain included in the same LSCPM community. Note that two DCPM communities that are merged (resp. split) in the next snapshot belong to the same LSCPM community.

To illustrate these definitions, we show in Figure 1a an example of a link stream, with time on the X-axis and vertices on the Y-axis. Links are represented as connections between two vertices, and the horizontal line indicate their duration. For example, there is a link between vertices c and d over the time interval $[1, 13]$. In Figure 1b, we represent its maximal k -cliques in color, in Figure 1c its LSCPM communities, and in Figure 1d its DCPM communities. The background of each vertex is colored according to the time during which it belongs to its clique or community. Notice that the red LSCPM community is composed of three DCPM communities: the red one, but also the green one because $\{e, f, g\}$ is not adjacent to $\{c, d, e\}$ at time $t = 3$; and the purple one because $\{e, f, g\}$ is no longer adjacent to $\{d, e, f\}$ for $t \geq 9$.



■ **Figure 1** Example of a link stream with its maximal k -cliques for $k = 3$ and the associated LSCPM communities and DCPM communities.

3 Algorithms

Our main idea to compute efficiently LSCPM communities in link streams is to use techniques similar to those developed for enumerating maximal cliques in link streams [3, 28]. Indeed, it is possible to enumerate k -cliques in a link stream efficiently by going through each link only once and aggregate adjacent ones using a temporal Union-Find data structure. In the DCPM case however, the comparison of communities at each time step to detect their evolution is more demanding. We present an algorithm to efficiently enumerate k -cliques in Section 3.1. We then adapt the best existing method for computing CPM communities, and propose an efficient algorithm that, given as input the k -cliques of a link stream, computes its communities, in Section 3.2.

3.1 k -clique enumeration algorithm in link streams

We take inspiration from our recent work [3] which enumerates the maximal cliques to design the algorithm for enumerating k -cliques in a link stream. The key idea is to use graph k -clique enumeration, as very efficient algorithms have been designed for this task [8]. Then, we compute the starting and ending time of each induced maximal k -clique in the link stream.

■ **Algorithm 1** k -clique enumeration in link streams.

Input: Link stream $L = (T, V, E)$; $k \in \llbracket 3, +\infty \rrbracket$.
Output: All k -cliques of L without duplicates.

```

1  $G \leftarrow$  empty graph
2  $\mathcal{E} \leftarrow$  empty associative array //  $\mathcal{E}$  associates ending times to edges
3 for  $(b, e, u, v) \in E$  sorted by increasing  $b$  do
4   Add edge  $\{u, v\}$  to  $G$ 
5    $\mathcal{E}(u, v) \leftarrow e$  // Record the ending time of  $\{u, v\}$ 
6   Remove from  $G$  all edges  $\{x, y\}$  with  $\mathcal{E}(x, y) < b$ 
7    $G\text{Cliques} \leftarrow$  all  $k$ -cliques of  $G$  containing  $u$  and  $v$ 
8   for  $C \in G\text{Cliques}$  do
9      $end \leftarrow \min_{x, y \in C} (\mathcal{E}(x, y))$ 
10    output  $k$ -clique  $(C, [b, end])$ 

```

Algorithm 1 lists all the maximal k -cliques. It starts from an empty graph (Line 1), and processes the link stream in chronological order (Line 3). For each link (b, e, u, v) , it updates the graph (Lines 4 to 6). Then, it enumerates the maximal k -cliques containing u and v induced by the links seen up till then. They match the static k -cliques in G that contain u and v (Line 7). Finally, each of these maximal k -clique starts at b , because the link between u and v does not exist before b , and lasts as long as all its links exist, so its ending time is the minimum of the ending times of the edges composing it (Line 9).

The complexity of Algorithm 1 is given by Theorem 3. Its proof is given in Appendix.

► **Theorem 3** (k -clique enumeration time complexity). *Algorithm 1 enumerates all k -cliques of the input link stream in time $\mathcal{O}\left(m \cdot k^3 \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$, where m is the number of links and d the maximal degree of a node, that is its maximal number of neighbors at any given time.*

The largest factor in this complexity is m . d can in theory be large with respect to m but, since it is the maximal number of neighbors of a node *at any given time*, it is small in practice. It is therefore the value of k that determines how efficient the enumeration can be, as the factors k^3 and $(\frac{d}{2})^{k-2}$ show that this method remains efficient only for small values of k .

3.2 LSCPM: CPM algorithm in link streams

To the best of our knowledge, the most efficient algorithmic implementation of CPM in graphs is [2]. In a few words, this algorithm stores each CPM community as the set of the $(k-1)$ -cliques composing its k -cliques. It builds these sets on the fly, by processing each k -clique one by one, testing to which community each of its $(k-1)$ -cliques currently belongs. Then, it merges these communities or creates a new one if needed. For this purpose, the algorithm uses a Union-Find data structure, as it is efficient to do these operations. It is a forest of trees, where each node corresponds to a $(k-1)$ -clique, and each tree corresponds to a CPM community, identified by its root. It has three intrinsic functions: `UF.Find(id)` that returns the root of the tree containing the node id , `UF.Union(p,q)` that performs the union of two trees by connecting their roots and returns the root of the new tree, and `UF.MakeSet()` which creates a new tree on a new root q , and returns this root.

We take inspiration from the algorithm above to extend the percolation of k -cliques to its definition in link streams. The procedure is given in Algorithm 2 and follows a similar logic: each LSCPM community is stored as the set of the temporal $(k-1)$ -cliques of its maximal k -cliques. A $(k-1)$ -clique of a maximal k -clique $(C_k, [t_0, t_1])$ is of the form $(C_{k-1}, [t_0, t_1])$, with $C_{k-1} \subseteq C_k$ containing $k-1$ vertices. These communities are constructed on the fly, by processing each maximal k -clique one by one, following the chronological order of their starting time, given by Algorithm 1. For each maximal k -clique $(C_k, [t_0, t_1])$ (Line 5), Algorithm 2 checks the community to which each of its $(k-1)$ -clique belongs on an interval that (strictly) intersects $[t_0, t_1]$ if it exists (Line 10), or creates a new one if needed (Line 16), then merges them (Lines 13 and 17). It also extends the membership duration of the $(k-1)$ -cliques in case they did not belong to this community until t_1 (Line 11). To do so, in addition to the Union-Find structure `UF`, we use an associative array `TimeUF`, which associates each C_{k-1} to a list of elements of the form $(id, [t_0, t_1])$, where id is the identifier of a Union-Find element and $[t_0, t_1]$ is the interval during which C_{k-1} belongs to the community of id . In these lists, the intervals are disjoint, and the pairs are sorted in ascending chronological order. Each list is initialized to $[(-1, -1, -1)]$ (Lines 3 and 4), meaning that the corresponding C_{k-1} has not yet been added to any community. Figure 2 gives an example of the update of `TimeUF` and `UF` structures, when applying Algorithm 2 to the link stream of Figure 1.

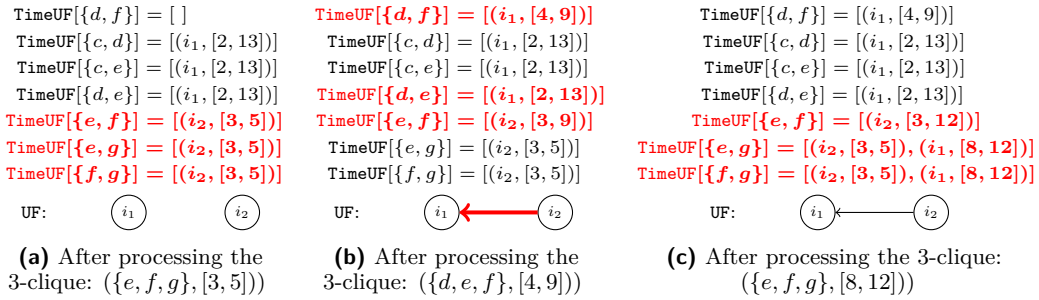
Finally, we need to transform the Union-Find structure into the adequate format to get the output as a temporal community. This is done with a loop through the elements of `TimeUF`. Each one is of the form $C_{k-1} \rightarrow I$, where C_{k-1} is a set of $k-1$ vertices, and I is a list of pairs $(id, [t_0, t_1])$ corresponding to a Union-Find element and a time interval. Each Union-Find element belongs to a single set, which is one of the LSCPM communities and that we obtain with the `Find` procedure. We then add each vertex of C_{k-1} to this community, on the time interval $[t_0, t_1]$.

The complexity of Algorithm 2 is given by Theorem 4 (demonstrated in Appendix). Note that Algorithm 2 takes as input the set of $(k-1)$ - and k -cliques of the link stream. Therefore, the total complexity given in Theorem 4 takes into account the time needed to perform their enumeration as well as the time needed to compute the communities.

■ **Algorithm 2** Clique Percolation Method in link streams (LSCPM).

Input: $(k-1)$ -cliques then k -cliques of a link stream $L = (T, V, E)$; $k \in \llbracket 3, +\infty \rrbracket$.
Output: Union-Find structure representing all LSCPM communities of L .

- 1 UF \leftarrow empty Union-Find data structure
- 2 TimeUF \leftarrow empty associative array
- 3 **for each** maximal $(k-1)$ -clique $(C_{k-1}, [t_0, t_1])$ of L **do**
- 4 TimeUF[C_{k-1}] $\leftarrow [(-1, -1, -1)]$
- 5 **for each** maximal k -clique $(C_k, [t_0, t_1])$ of L , sorted by increasing t_0 **do**
- 6 $p \leftarrow -1$
- 7 **for each** $u \in C_k$ **do**
- 8 $C_{k-1} \leftarrow C_k \setminus \{u\}$
- 9 $(id, [t'_0, t'_1]) \leftarrow$ last element of TimeUF[C_{k-1}]
- 10 **if** $t_0 < t'_1$ **then** // is in the current community
- 11 last element of TimeUF[C_{k-1}] $\leftarrow (id, [t'_0, \max(t_1, t'_1)])$
- 12 $q \leftarrow$ UF.Find(id)
- 13 $p \leftarrow$ UF.Union(p, q) // merge with other $(k-1)$ -cliques
- 14 **else** // not yet or no longer in the community
- 15 **if** $p = -1$ **then**
- 16 $p \leftarrow$ UF.MakeSet()
- 17 Append $(p, [t_0, t_1])$ to TimeUF[C_{k-1}] // add to community of p



■ **Figure 2** Example of updates of TimeUF and UF of Algorithm 2, during the processing of the second, third and fourth 3-cliques of the link stream of Figure 1. Note that all lists in TimeUF begin by a $(-1, -1, -1)$ triplet which we omit for readability. We show only the part of TimeUF relevant to the cliques under study. At each time step, the $(k-1)$ -cliques corresponding to the added clique are shown in red. In 2a, three $(k-1)$ -cliques are added to the structure. In 2b, communities of i_1 and i_2 are merged, as the k -clique contains one $(k-1)$ -clique in i_1 and another in i_2 ; also one $(k-1)$ -clique is added: $\{d, f\}$, one is extended in time: $\{e, f\}$, and one remains unchanged because it is already present in the community over a longer time interval: $\{d, e\}$. In 2c, one $(k-1)$ -clique is extended in time: $\{e, f\}$, and as the other two, $\{e, g\}$ and $\{f, g\}$, were not in the community any more, they are re-added over the time interval of the k -clique $[8, 12]$. At the end of the process, the structure matches the information represented by the red LSCPM community of Figure 1.

► **Theorem 4** (LSCPM time complexity). *The time complexity of Algorithm 2 is in $\mathcal{O}((k + \alpha(n_k)) \cdot k \cdot n_k + c(k))$, where α is the inverse Ackermann function, n_k the number of k -cliques of the link stream, and $c(k)$ the complexity of enumerating k -cliques, given in Theorem 3. It is thus in $\mathcal{O}\left((k + \alpha(n_k)) \cdot m \cdot k^2 \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$.*

This complexity is expressed with the inverse Ackermann function α , which is known to

grow extremely slowly, and can be considered as a constant at the scale of our data. Thus, we see from this theorem that the time complexity is close to $\mathcal{O}(k^2 \cdot n_k + c(k))$. This shows that our algorithm is efficient in the way each k -clique is processed, once the k -cliques have been computed. Indeed, each k -clique contains $k(k-1)$ -cliques, and therefore it is not possible to process them in less than $k \cdot n_k$ operations, using an approach similar to ours.

The second part of the theorem is obtained by replacing $c(k)$ by the expression of Theorem 3 and n_k by a bound on its value. If we do not take into account the factor $\alpha(n_k)$, the complexity is in $\mathcal{O}\left(k^3 \cdot m \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$. As noted above, this complexity depends almost linearly on m and the factor d is small in practice. Nevertheless, the cubic factor in k and the $\left(\frac{d}{2}\right)^{k-2}$ make it manageable only for small values of k . We will see in Section 4 that small values of k allow for a fast building of the communities and are sufficient to observe interesting properties of the datasets.

In practice, Algorithm 2 needs to store in memory all the $(k-1)$ -cliques in the link stream. It can be limiting, for example if the input dataset contains a very large clique. For instance, if there is a clique of size 1000, and that we are looking for 6-clique communities, then there are more than 10^{15} 5-cliques to store from this large clique. Still, data from real-world interactions are known not to exhibit many large cliques, which makes the k -clique approach interesting for their study, and this memory feature has not been prohibitive during our experiments.

4 Experiments

For the experimental study, we implemented our algorithm in Python and the code is available online⁴. Throughout this section, we set $k = 3$ unless otherwise specified. We will see that this value allows for a fast computation while being sufficient to provide interesting information on the datasets. Also, we present in Section 4.5 the impact of increasing k on the community structures, which induces smaller communities and therefore allows targeting their core, with more or less strength depending on the value by which k has been increased.

4.1 Datasets

We run our experiments on real-world link streams of various sizes and types of interactions. Even though many datasets consist of links with duration, in many cases these data are registered with regular discrete time intervals, because of the practical data acquisition protocol. This is the case for instance of proximity between individuals data, usually captured using RFID tags. Therefore, currently there is a larger range of publicly available instantaneous link streams with links of the form (t, u, v) , where u and v are vertices interacting at time t . So, we transform these link streams by adding a duration Δ creating links of the form $(t, t + \Delta, u, v)$. Note that the value of Δ will impact the number and extension of cliques in the link stream. Practically, we choose uniform Δ values which are consistent with the typical time scales of the interactions considered for the datasets under study. These values, while consistent, remain arbitrary, and we use them to demonstrate the efficiency and relevance of our algorithm. Users can adjust the values according to their requirements and to the nature of the studied datasets.

The datasets on which we performed the experiments are described in Table 1. *Households* is a link stream representing contacts between members of five households in rural

⁴ <https://gitlab.lip6.fr/baudin/lscpm>

Kenya in 2012 [13]; *Highschool* corresponds to contacts between students of five classes of a preparatory school in Marseilles (France) during one week in 2012 [10] and *Infectious* consists of contacts between visitors to a museum in Dublin (Ireland) in 2009 [12]. These three datasets represent contacts between individuals, for which we have chosen to take a link duration $\Delta = 1$ hour. The *Foursquare* dataset is extracted from the eponymous application, where users check-in in venues that they visit, located in New-York City in our case [29]. It can thus be represented as a bipartite link stream between visitors and locations, where timestamps correspond to the check-in time. In this link stream, we set $\Delta = 6$ hours, then we project it on the set of locations: if a user is connected to two locations over an overlapping time interval, this will create a link between these locations during the overlap. If the time interval of two links created in this way overlap, they are merged into a single link over the union of the initial time intervals. Finally, we use the link stream *Wikipedia*, which represents links between Wikipedia pages, timestamped by the time of the link creation, over several years in the 2000s [17]. We choose a $\Delta = 1$ week duration, essentially to explore how our method scales to massive link streams.

Link stream	Δ	m	n	d	D	r
<i>Households</i>	1 hour	2,136	75	19	3 days	1 hour
<i>Highschool</i>	1 hour	5,528	180	18	8 days	20s
<i>Infectious</i>	1 hour	44,658	10,972	43	3 months	5 min
<i>Foursquare</i>	6 hours	268,472	33,153	81	10 months	15 min
<i>Wikipedia</i>	1 week	38,953,380	1,870,709	33,217	2.3 years	20s

■ **Table 1** Link stream datasets. Δ is the link duration, m the number of links, d the maximal degree, n the number of nodes, D the total duration from the first to the last link and r the time resolution, that is the smallest duration between the beginning of two links.

The link stream parameter that affects most dramatically the running time of the community detection is its number of k -cliques n_k , as the complexity depends strongly on k and n_k according to Theorem 4. In Table 2, we report the number of k -cliques for each dataset, for k ranging from 3 to 7. It allows anticipating the differences in computation time between the datasets, which are detailed in Section 4.2. We notice that for large datasets, n_k increases with k , certainly because these datasets contain some large cliques. Indeed, within a clique containing c vertices, there are $\binom{c}{k}$ cliques with k vertices, and that quantity grows with k (as long as $k \leq \frac{c}{2}$). In particular, *Foursquare* is a projection of a bipartite network, and projections are known to contain many large cliques.

Link stream	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$
<i>Households</i>	3,951	4,721	3,929	2,324	987
<i>Highschool</i>	2,468	583	97	11	1
<i>Infectious</i>	79,836	128,157	202,181	274,181	300,850
<i>Foursquare</i>	571,768	2,423,011	17,823,050	155,466,085	1,302,290,726
<i>Wikipedia</i>	3,757,877	1,148,832	1,763,386	4,545,105	11,853,134

■ **Table 2** Number of k -cliques (n_k) for each dataset and for k from 3 to 7.

4.2 LSCPM: faster and scaling to massive real-world link streams

We now compare our algorithm to the DCPM one in terms of running time. Note that the comparison focuses on the running time and not on the complexity as the complexity of the DCPM method or the existing OCPM implementation are not given by their authors. In our implementation, the k -cliques are streamed to the standard input of the LSCPM algorithm, which reads them as the enumeration proceeds. These two operations are done on two different threads; but for comparison purposes with the DCPM running time, we measure its computation time as the sum of the time spent on each of these two threads. The DCPM runtime is provided by the best implementation available [4]. We refer to this implementation as OCPM (standing for Online CPM). We performed all the experiments on a Linux machine, equipped with two processors Intel Xeon Silver 4210R with twenty cores each, at 2.40Ghz, and with 252Gb of RAM.

Table 3 presents the computation times of communities with the OCPM implementation of DCPM and our LSCPM implementation, on all the datasets of Table 1, for k from 3 to 7. These values are also plotted in Figure 3 for readability purposes. We observe that LSCPM is significantly faster, particularly on more massive datasets. For example, with $k = 3$ or $k = 4$, it takes a few seconds with LSCPM to compute the *Foursquare* communities, while it takes a few hours with OCPM. Moreover, for the massive *Wikipedia* link stream, OCPM is unable to compute the set of communities in a week, while our algorithm provides the communities in less than 30 minutes for all k values tested. Our algorithm thus allows to study a community structure in massive datasets for which the state of the art does not provide a result.

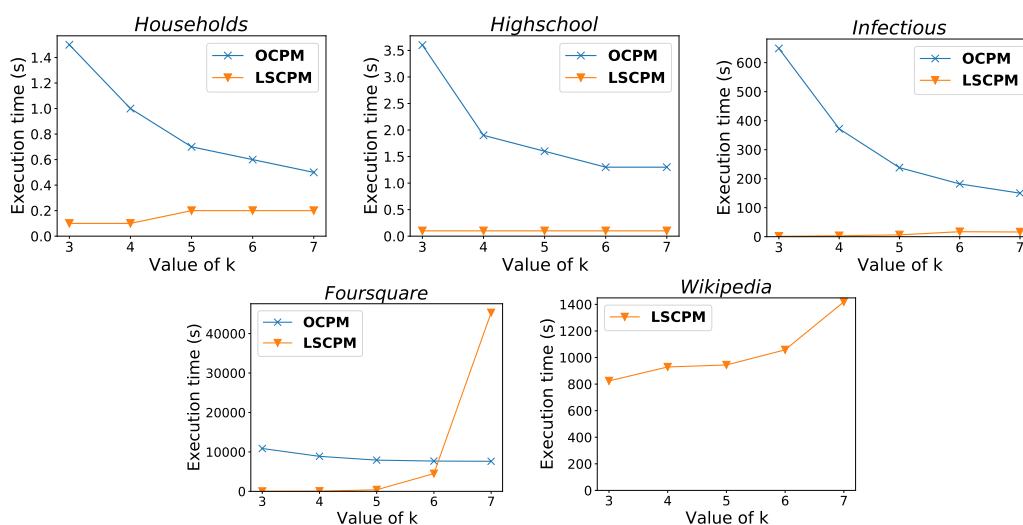
Link stream	$k = 3$		$k = 4$		$k = 5$		$k = 6$		$k = 7$	
	OCPM	LSCPM	OCPM	LSCPM	OCPM	LSCPM	OCPM	LSCPM	OCPM	LSCPM
<i>Households</i>	1.5s	0.1s	1.0s	0.1s	0.7s	0.2s	0.6s	0.2s	0.5s	0.2s
<i>Highschool</i>	3.6s	0.1s	1.9s	0.1s	1.6s	0.1s	1.3s	0.1s	1.3s	0.1s
<i>Infectious</i>	10min49s	1.4s	6min12s	3.3s	3min58s	6.2s	3min02s	17.2s	2min30s	16.2s
<i>Foursquare</i>	3h01min	9.2s	2h28min	43s	2h12min	6min39s	2h08min	1h15mins	2h07min	12h35min
<i>Wikipedia</i>	-	13min44s	-	15min29s	-	15min44s	-	17min38s	-	23min39s

■ **Table 3** Time computation of communities in seconds with both OCPM and LSCPM, for all our datasets, with k varying from 3 to 7. The symbol “-” means that the computation time exceeds one week.

Another point of interest is that the LSCPM computation time increases with k , while it decreases with OCPM. This comes from the fact that OCPM implementation obtains its results by aggregating the maximal cliques of size *at least* k , while our method enumerates k -cliques. With larger k , there are fewer maximal cliques to enumerate and process, hence the decreasing computation time. By contrast, we have seen in Section 4.1 that n_k typically increases with k for larger datasets, which implies that the computation time of LSCPM increases too according to Theorem 4. Notice however that in spite of this, there is only one instance where OCPM is faster than LSCPM: *Foursquare* link stream with $k = 7$, which as we have seen as a very large number of k -cliques.

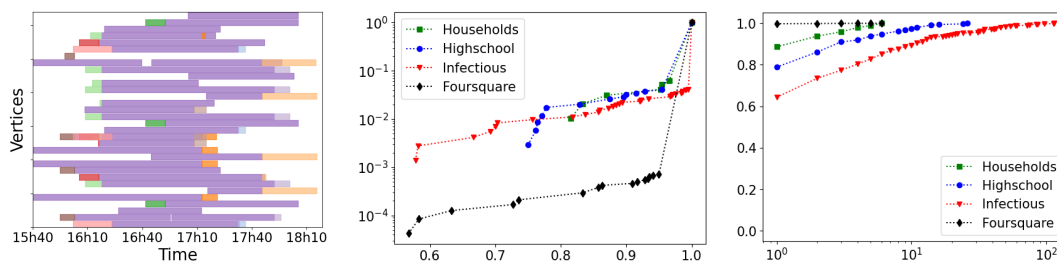
4.3 Comparison between LSCPM and DCPM communities

In what follows, we compare the communities obtained with our LSCPM algorithm to those obtained with DCPM, on the four datasets where the OCPM implementation provides a result. Note that, up to our knowledge, there is no reference method to compare overlapping



■ **Figure 3** Line charts of the running times of OCPM and LSCPM for each dataset. Values are those in Table 3.

temporal communities. So we do not use tools such as NMI or Rand index which are designed for comparing partitions of vertices in a graph and thus would require some adaptation to the context considered in this paper. We have seen in Section 2 that each DCPM community is included in a LSCPM one and, conversely, that each LSCPM community can be seen as the union of DCPM communities. This property is illustrated in Figure 4 (left), which gives an example of a LSCPM community from *Infectious* dataset, with $k = 3$ using the python package `tnetwork`⁵. Vertices are represented on the Y-axis, time on the X-axis, and each vertex belongs to the community over the period during which it is colored. Each of the DCPM communities included in this LSCPM community is represented in a different color. In what follows, we investigate to what extent DCPM communities are grouped into LSCPM communities.



■ **Figure 4** Composition of LSCPM communities in terms of DCPM communities, with $k = 3$. Left: a LSCPM community of *Infectious*, and the DCPM communities included in it (one color each). Center: cumulative distribution of the relative size (in number of vertices) of the largest DCPM community to the corresponding LSCPM community. Right: cumulative distribution of the number of DCPM community per LSCPM community.

To evaluate the similarity between a LSCPM community and the DCPM communities that it contains, we compute the relative size (in number of vertices) of the largest DCPM

⁵ <https://tnetwork.readthedocs.io/>

community that each LSCPM community contains. Figure 4 (center) reports the cumulative distribution of this value. We see a clear peak at the end, which shows that for all datasets, 90% of the LSCPM communities contain as many vertices as their largest DCPM community, which is the case of the example in Figure 4 (left). Only 1% of LSCPM communities have its largest DCPM community with less than 70% of the vertices, and none have its largest DCPM community with less than 50%.

We also observe that there are only a few DCPM communities per LSCPM community. It is illustrated in Figure 4 (right), which represents the cumulative distribution of the number of DCPM communities that each LSCPM community contains. In all cases, more than 70% of LSCPM communities contain only 1 or 2 DCPM communities, and almost never more than 10. However, there are some exceptions: in *Highschool*, 2.6% of the communities contain between 10 and 26 DCPM communities, and in *Infectious*, 1.8% of LSCPM communities contain between 50 and 115 DCPM communities.

Besides, we observe that there are fewer small LSCPM communities than DCPM ones: considering communities with 5 or fewer vertices, we observe that *Households* has 17% more DCPM communities than LSCPM, *Highschool* has twice as many and *Infectious* has six times as many, (however, the sets of LSCPM and DCPM communities are very similar for *Foursquare*). This indicates that small DCPM communities tend to be aggregated into larger LSCPM ones, as observed in the example of Figure 4 (left). These observations give the typical scheme of how LSCPM communities compare to the DCPM ones: a LSCPM community is in general composed of one large DCPM community which contains almost all the vertices, and possibly a few residual communities. It also indicates that most of the meaning conveyed by the communities obtained in both cases should be closely related, but LSCPM method allows streamlining the community analysis by aggregating the smaller, less meaningful, communities into the larger ones.

4.4 Insights on LSCPM communities

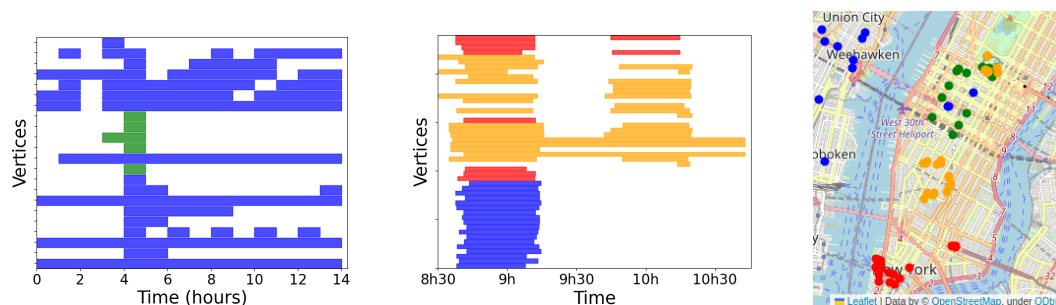
To investigate the relevance of the temporal communities obtained, we have at our disposal metadata retrieved from the dataset repositories: families of *Households*, class of *Highschool*, and GPS coordinates of locations in *Foursquare*.

In the case of the *Households* and *Highschool* datasets, which are based on person-to-person interactions, we observe that the communities are homogeneous in terms of these categories, as could be expected. Indeed, in the case of *Households* dataset, 95% of communities are composed of members of only one family, and the remaining 5% of two families. In *Highschool*, 70% of communities are composed of only one class, 23% of two classes, 6% of three classes, and 1% of four classes.

Moreover, this metadata provides interesting insight on interactions at the individual level over time, pointing out who socializes with whom and when. For instance, Figure 5 (left) is a community of *Households*, composed of 5 members of a family (in green) and 17 members of another (in blue). It highlights the existence of an at most one hour gathering between all these people except one. Similarly, Figure 5 (center) shows a community where we observe members of three different classes of *Highschool*, which are the three physics major classes of the preparatory school. We distinguish three time periods: during the first one, students from the three classes are grouped together, which suggests a period when students can meet and mix. Then, the community reduces to a few nodes of the orange class, later joined by an additional group of students, which might indicate the proximity of the students during the courses or working groups.

Concerning *Foursquare*, we can use the metadata to investigate the geographical distribu-

tion of locations which are visited by the same individuals within a Δ period. Figure 5 (right) shows a map of a part of New-York City displaying a sample of four LSCPM communities. We observe that some of them are relatively clustered geographically, such as the green one, while others are more extended, which happens when one or several persons move from one part of the city to another within a Δ period. Thus here, Δ allows tuning the geographical extension of the communities, as lower Δ should correspond to smaller geographical extensions.

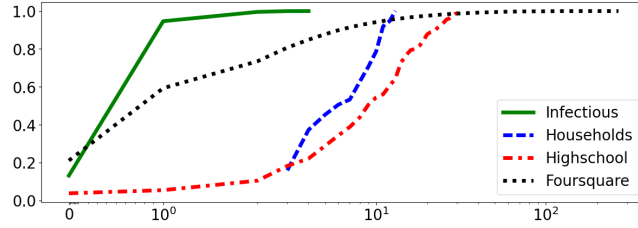


■ **Figure 5** Examples of LSCPM communities using the metadata of the datasets. Left: a community of *Households* with vertices colored according to the family. Center: a community of *Highschool* with vertices colored according to the class. Right: a map of Manhattan (NYC), where four *Foursquare* communities are represented in different colors.

It may also be relevant to evaluate the involvement of vertices in the communities that they belong to. Indeed, some vertices are active longer than others or belong to more communities, so it makes it possible to identify particularly important nodes in a group or nodes which act as bridges between groups. Figure 6 illustrates these two aspects: it is the cumulative distribution of the number of communities to which each vertex belongs. Points on the far left correspond to vertices that belong to no community at all. In this regard, the various datasets yield very different results. For example, in *Foursquare*, around 20% of the vertices do not belong to any community; it corresponds to locations where users rarely visit other places over the time period considered. We observe this for some specific locations such as medical centers, offices, playgrounds... By contrast, in *Households*, each vertex belongs to at least two communities, which is reasonable as it is a contact network between members of a same household, so no contact at all hardly seems conceivable. In *Highschool*, we see that most nodes belong to many communities, which also makes sense, as students are grouped in classes, and each day makes new LSCPM communities. Finally, vertices that belong to many communities are on the far right of the distribution. It is striking on *Foursquare*, where almost 10% of vertices belong to more than 10 communities, and a few to more than 100. These can be described as central nodes of the link stream, which interact with many other nodes throughout the period of observation. For example, the location of *Foursquare* that belongs to the most communities by far (1.5 times more than the second) is the famous Pennsylvania Station, which is the main intercity train station of New York City.

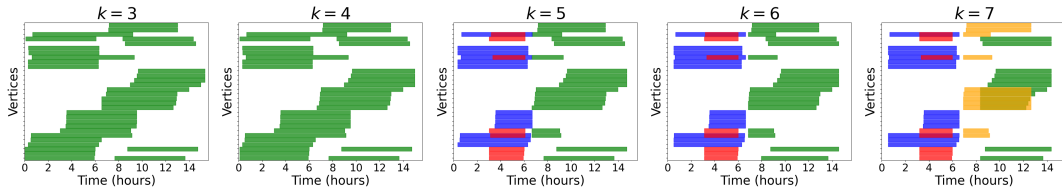
4.5 Influence of k on the community structure

The size k of the cliques at the base of the LSCPM communities is the key parameter of the algorithm. Here, we discuss the effect of increasing k on the community structure, in order to give an intuition to the user as to the choice to make for this value.



■ **Figure 6** Number of communities to which each vertex belongs, as a cumulative distribution. Note that the X-axis is in logarithmic scale, except between 0 and 1, in order to show vertices that belong to no community.

As we saw in Section 2, if $k_1 < k_2$, then each community computed with k_2 is included in a community computed with k_1 . This means that increasing k results in the communities splitting and/or losing temporal vertices. Figure 7 gives an example of this phenomenon on a community of the *Foursquare* dataset: from left to right we see the community computed with $k = 3$ and how it splits and loses nodes when increasing k up to 7. We observe that the community remains almost identical for $k = 4$, and that it splits into three communities for $k = 5$, and one of these smaller communities split again for $k = 7$. We also observe that some nodes which belong to the community for $k = 3$ at a given time do not belong to any of the resulting communities at this time for larger values of k . Thus, increasing k leads to more cohesive communities, but smaller in size and time extension. This allows to change the granularity of the dynamical communities that can be obtained on a dataset by focusing on the “core” of the interactions.



■ **Figure 7** Split of a *Foursquare* community as k increases from $k = 3$ to $k = 7$.

This allows to identify consistent sub-communities when metadata is available. For instance, the vertices of the *Foursquare* community of Figure 7 correspond to 6 types of locations (out of the 261 available), all sport-related: Athletic-&-Sport, Bike-Shop, Stadium, Sporting-Goods-Shop, Gym-/Fitness-Center, Motorcycle-Shop (most other communities exhibit more diverse labels). We investigate if the splitting of communities when k increases corresponds to more precise types of locations. For $k = 5$, the green community has the 6 type labels, but nodes of the blue and red ones only have 3 labels: Sporting-Goods-Shop, Gym-/Fitness-Center and Bike-Shop. Also, for $k = 7$, the green community splits into two parts, one of which focuses on two-wheeled sports: Motorcycle, Bike, Stadium, which highlights that the decomposition allows to derive the shared interests of the users.

5 Conclusion and perspectives

In this paper, we address the detection of dynamic communities in temporal networks, using the link stream formalism. Using the literature of the field, we introduced the notion of maximal k -clique of a link stream, with an algorithm to enumerate them. This leads to a new

adaptation of the Clique Percolation Method to dynamical networks, called LSCPM, which pursues the work initiated by [19]. We provided a theoretical analysis of the complexity of our algorithm, as well as an open source implementation in Python. Then, we experimented with the algorithm, comparing it to the state of the art, and showed that it allows to obtain possibly relevant information on real-world examples.

We believe that the community detection with LSCPM can scale to even more massive networks and larger values of k . For instance, as memory consumption can be a limiting factor on massive streams because of the clique storage, the work done in [2] to reduce the RAM cost of the CPM method on graphs could be adapted to link streams. Also, it could be better in some cases to percolate maximal cliques instead of k -cliques, as done in OCPM [4], using efficient maximal clique enumeration methods in link streams such as [3]. In particular, we have seen that this may be an efficient alternative for larger k values.

Moreover, it would be interesting to develop the analysis of the effect of link duration Δ and its practical implications. Indeed, when Δ increases, the k -cliques grow longer, resulting in more aggregated LSCPM communities. This is the opposite effect of what happens when we increase k . We believe that the experimental study can be extended by testing the simultaneous tuning of these two parameters, to see how the communities are structured, and if this allows targeting relevant interaction cores in particular.

Finally, we believe that it is possible to adapt the algorithm for enumerating k -cliques in link streams, into a more general framework of temporal motif listing. Indeed, the call of the algorithm to enumerate graph cliques could be replaced in principle by any other motif mining algorithm in a graph, resulting in a related temporal motif. This paves the way to novel and efficient pattern mining algorithms in temporal networks.

References

- 1 Thomas Aynaud and Jean-Loup Guillaume. Static community detection algorithms for evolving networks. In *8th international symposium on modeling and optimization in mobile, ad hoc, and wireless networks*, pages 513–519. IEEE, 2010.
- 2 Alexis Baudin, Maximilien Danisch, Sergey Kirgizov, Clémence Magnien, and Marwan Ghanem. Clique percolation method: memory efficient almost exact communities. In *Advanced Data Mining and Applications: 17th International Conference, ADMA 2021, Sydney, NSW, Australia, February 2–4, 2022, Proceedings, Part II*, pages 113–127. Springer, 2022.
- 3 Alexis Baudin, Clémence Magnien, and Lionel Tabourier. Faster maximal clique enumeration in large real-world link streams. *arXiv preprint arXiv:2302.00360*, 2023.
- 4 Souâad Boudebza, Rémy Cazabet, Faïçal Azouaou, and Omar Nouali. OL-CPM: An online framework for detecting overlapping communities in dynamic social networks. *Computer Communications*, 123:36–51, 2018. Code available at: https://figshare.com/articles/software/OLCPM_algorithm/5914846.
- 5 Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. Time-varying graphs and dynamic networks. *International Journal of Parallel, Emergent and Distributed Systems*, 27(5):387–408, 2012.
- 6 Rémy Cazabet, Giulio Rossetti, and Frédéric Amblard. Dynamic community detection, 2017.
- 7 Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. Online search of overlapping communities. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 277–288, 2013.
- 8 Maximilien Danisch, Oana Balalau, and Mauro Sozio. Listing k -cliques in sparse real-world graphs. In *Proceedings of the 2018 World Wide Web Conference*, pages 589–598, 2018.
- 9 Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

- 10 Julie Fournet and Alain Barrat. Contact patterns among high school students. *PloS one*, 9(9):e107878, 2014. Data available at: <http://www.sociopatterns.org/datasets/high-school-dynamic-contact-networks>.
- 11 Petter Holme and Jari Saramäki. Temporal networks. *Physics reports*, 519(3):97–125, 2012.
- 12 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. Data available at: <http://www.sociopatterns.org/datasets/infectious-sociopatterns/>.
- 13 Moses C Kiti, Michele Tizzoni, Timothy M Kinyanjui, Dorothy C Koech, Patrick K Munywoki, Milosch Meriac, Luca Cappa, André Panisson, Alain Barrat, Ciro Cattuto, et al. Quantifying social contacts in a household setting of rural kenya using wearable proximity sensors. *EPJ data science*, 5:1–21, 2016.
- 14 Jussi M Kumpula, Mikko Kivelä, Kimmo Kaski, and Jari Saramäki. Sequential algorithm for fast clique percolation. *Physical review E*, 78(2):026109, 2008.
- 15 Matthieu Latapy, Tiphaine Viard, and Clémence Magnien. Stream graphs and link streams for the modeling of interactions over time. *Social Network Analysis and Mining*, 8:1–29, 2018.
- 16 Matteo Magnani, Obaida Hanteer, Roberto Interdonato, Luca Rossi, and Andrea Tagarelli. Community detection in multiplex networks. *ACM Computing Surveys (CSUR)*, 54(3):1–35, 2021.
- 17 Alan Mislove. *Online Social Networks: Measurement, Analysis, and Applications to Distributed Information Systems*. PhD thesis, Rice University, Department of Computer Science, May 2009. Data available at: <https://socialnetworks.mpi-sws.org/data-wosn2008.html>.
- 18 Nam P Nguyen, Thang N Dinh, Sindhura Tokala, and My T Thai. Overlapping communities in dynamic networks: their detection and mobile applications. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 85–96, 2011.
- 19 Gergely Palla, Albert-László Barabási, and Tamás Vicsek. Quantifying social group evolution. *Nature*, 446(7136):664–667, 2007.
- 20 Gergely Palla, Imre Derényi, Illés Farkas, and Tamás Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *nature*, 435(7043):814–818, 2005.
- 21 Gang Pan, Wangsheng Zhang, Zhaohui Wu, and Shijian Li. Online community detection for large complex networks. *PloS one*, 9(7):e102799, 2014.
- 22 Fergal Reid, Aaron McDaid, and Neil Hurley. Percolation computation in complex networks. In *2012 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pages 274–281. IEEE, 2012.
- 23 Giulio Rossetti, Luca Pappalardo, Dino Pedreschi, and Fosca Giannotti. Tiles: an online algorithm for community discovery in dynamic social networks. *Machine Learning*, 106:1213–1241, 2017.
- 24 Michael T Schaub, Jean-Charles Delvenne, Martin Rosvall, and Renaud Lambiotte. The many facets of community detection in complex networks. *Applied network science*, 2(1):1–13, 2017.
- 25 Yifu Tang, Jianxin Li, Nur Al Hasan Haldar, Ziyu Guan, Jiajie Xu, and Chengfei Liu. Reliable community search in dynamic networks. *arXiv preprint arXiv:2202.01525*, 2022.
- 26 Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984.
- 27 Tiphaine Viard, Matthieu Latapy, and Clémence Magnien. Computing maximal cliques in link streams. *Theoretical Computer Science*, 609:245–252, 2016.
- 28 Tiphaine Viard, Clémence Magnien, and Matthieu Latapy. Enumerating maximal cliques in link streams with durations. *Information Processing Letters*, 133:44–48, 2018.
- 29 Dingqi Yang, Daqing Zhang, Vincent W Zheng, and Zhiyong Yu. Modeling user activity preference by leveraging user spatial temporal characteristics in lbsns. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 45(1):129–142, 2014. Data available at: <https://sites.google.com/site/yangdingqi/home/foursquare-dataset>.

A Proofs for Section 3 (Algorithms)

► **Theorem 3** (*k*-clique enumeration time complexity). *Algorithm 1 enumerates all k-cliques of the input link stream in time $\mathcal{O}\left(m \cdot k^3 \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$, where m is the number of links and d the maximal degree of a node, that is its maximal number of neighbors at any given time.*

Proof. First, we show that the update of G from Lines 4 to 6 is done in $\mathcal{O}(m \cdot \log(m))$ in the whole loop of Line 3. G is stored as an associative array, associating to each vertex the set of its neighbors, so the addition of an edge at Line 4 is done in constant time. Line 5 is also done in constant time. Finding all edges that have an ending time smaller than b (Line 6) can be done by maintaining a sorted list of the end times of links in G , which can be done in $\mathcal{O}(\log(m))$ for each new link. Then, the deletion itself is done in constant time by going through this list. The global complexity is therefore in $\mathcal{O}(m \cdot \log(m) + m) = \mathcal{O}(m \cdot \log(m))$.

Consider an iteration of the loop starting at Line 3 and (b, e, u, v) its associated link. We need to compute all k -cliques of G containing u and v . This is equivalent to computing the $(k-2)$ -cliques of the graph induced by the common neighbors of u and v , that we note $G(N(u) \cap N(v))$. Computing this induced subgraph is done in $\mathcal{O}(d^2)$. Then, the overall complexity of these operations over all iterations is in $\mathcal{O}(m \cdot d^2)$.

Enumerating the $(k-2)$ -cliques of $G(N(u) \cap N(v))$ depends on the value of k . If $k = 3$ it consists in enumerating vertices, which is in $\mathcal{O}(d)$. If $k = 4$, it is enumerating the edges, in $\mathcal{O}(d^2)$. If $k \geq 4$, then we use the k -clique enumeration algorithm in graphs described in [8]. In that paper, Theorem 5.7 gives the complexity of enumeration in $\mathcal{O}\left(k \cdot m \cdot \left(\frac{d}{2}\right)^{k-2}\right)$. Thus, the $(k-2)$ -clique enumeration is in $\mathcal{O}\left((k-2) \cdot d^2 \cdot \left(\frac{d}{2}\right)^{k-4}\right)$ and the overall complexity of Line 7 is in $\mathcal{O}\left(k \cdot \left(\frac{d}{2}\right)^{k-2}\right)$, for any value of k . Note that this value sets an upper bound on the number of k -cliques enumerated by the loop iteration. Each of these k -cliques is then processed by the loop at Line 8, in $\mathcal{O}(k \cdot (k-1)) = \mathcal{O}(k^2)$. Then, the total complexity of these operations in the iteration is in $\mathcal{O}\left(k^3 \cdot \left(\frac{d}{2}\right)^{k-2}\right)$, it is thus in $\mathcal{O}\left(m \cdot k^3 \cdot \left(\frac{d}{2}\right)^{k-2}\right)$ for the entire loop.

Combining the cost of the above operations finally gives the result. Note that this result can be slightly refined by keeping $k \cdot (k-1) \cdot (k-2)$ instead of k^3 , which may have a significant impact, since k values are usually small (typically ≤ 10). ◀

► **Theorem 4** (LSCPM time complexity). *The time complexity of Algorithm 2 is in $\mathcal{O}\left((k + \alpha(n_k)) \cdot k \cdot n_k + c(k)\right)$, where α is the inverse Ackermann function, n_k the number of k -cliques of the link stream, and $c(k)$ the complexity of enumerating k -cliques, given in Theorem 3. It is thus in $\mathcal{O}\left((k + \alpha(n_k)) \cdot m \cdot k^2 \cdot \left(\frac{d}{2}\right)^{k-2} + m \cdot d^2 + m \cdot \log(m)\right)$.*

Proof. With a Union-Find data structure, it is known that the amortized cost of **Union** and **Find** functions is in $\mathcal{O}(\alpha(N))$, if N is the number of elements in the structure (see for example [26]). Here, the Union-Find structure contains at most 1 element per maximal k -clique, since there cannot be more than one call to **MakeSet** (Line 16) in each iteration of the loop starting at Line 5. Indeed, if a **MakeSet** is performed, then p is no longer equal to -1 and no other is performed until the end of this loop. So the complexity of each call to **Union** and **Find** functions in the procedure is in $\mathcal{O}(\alpha(n_k))$.

Now, consider a maximal k -clique $(C_k, [t_0, t_1])$ corresponding to an iteration of the loop starting at Line 5. Line 7 performs one iteration per vertex of C_k , that is k iterations. The operation at Line 9 to find the last element of **TimeUF** $[C_{k-1}]$ is in $\mathcal{O}(k-1)$. During this

loop, there is also at most one call to **Union** and **Find**, and other operations in constant time. So, it runs in $\mathcal{O}(k + \alpha(n_k))$. Thus, in total, the loop starting at Line 5 runs in $\mathcal{O}(n_k \cdot k \cdot (k + \alpha(n_k)))$.

In addition, we have to take into account the complexity of the enumeration of k -cliques and $(k - 1)$ -cliques given as input to the algorithm. However, Theorem 3 indicates that the complexity of enumerating $(k - 1)$ -cliques is included in the one of enumerating k -cliques, denoted $c(k)$. We thus obtain an overall complexity of Algorithm 2 in $\mathcal{O}((k + \alpha(n_k)) \cdot k \cdot n_k + c(k))$.

Finally, we saw in the proof of Theorem 3 above that at each iteration of the loop at Line 3 of Algorithm 1, the number of k -cliques listed is in $\mathcal{O}\left(k \cdot \left(\frac{d}{2}\right)^{k-2}\right)$. Since there are m iterations, we get that the number of k -cliques n_k is in $\mathcal{O}\left(m \cdot k \cdot \left(\frac{d}{2}\right)^{k-2}\right)$. Hence the second part of the theorem by combining the above bound on n_k and Theorem 3. ◀

Note that the output of Algorithm 2 is the Union-Find structure and thus a post-processing is required to produce the actual communities. This post-processing is done by going through the set of elements $(C_{k-1}, [t_0, t_1])$ in **TimeUF**, that are at most $k \cdot n_k$ (at most k per maximal k -clique). Then, it performs a **Find** operation on them and adds each of its $k - 1$ associated vertices to its community during the corresponding time interval. Adding a vertex to its community can be done in constant time if the nodes of the Union-Find are browsed in chronological order, which is possible by storing their creation time order at Line 16. So the complexity of the post-processing is also in $\mathcal{O}(k \cdot n_k \cdot (k + \alpha(n_k)))$.