2SCENT: An Efficient Algorithm for Enumerating All Simple Temporal Cycles

Rohit Kumar
Université Libre de Bruxelles
Brussels, Belgium
Universitat Politécnica de Catalunya
(BarcelonaTech)
Barcelona, Spain
rohit.kumar@ulb.ac.be

Toon Calders
Universiteit Antwerpen
Antwerp, Belgium
Université Libre de Bruxelles
Brussels, Belgium
toon.calders@uantwerpen.be

ABSTRACT

In interaction networks nodes may interact continuously and repeatedly. Not only which nodes interact is important, but also the order in which interactions take place and the patterns they form. These patterns cannot be captured by solely inspecting the static network of who interacted with whom and how frequently, but also the temporal nature of the network needs to be taken into account. In this paper we focus on one such fundamental interaction pattern, namely a temporal cycle. Temporal cycles have many applications and appear naturally in communication networks. In financial networks, on the other hand, the presence of a temporal cycle could be indicative for certain types of fraud, and in biological networks, feedback loops are a prime example of this pattern type. We present 2SCENT, an efficient algorithms to find all temporal cycles in a directed interaction network. 2SCENT consist of a non-trivial temporal extension of a seminal algorithm for finding cycles in static graphs, preceded by an efficient candidate root filtering technique which can be based on Bloom filters to reduce the memory footprint. We tested 2SCENT on six real-world data sets, showing that it is up to 300 times faster than the only existing competitor and scales up to networks with millions of nodes and hundreds of millions of interactions. Results of a qualitative experiment indicate that different interaction networks may have vastly different distributions of temporal cycles, and hence temporal cycles are able to characterize an important aspect of the dynamic behavior in the networks.

PVLDB Reference Format:
Rohit Kumar and Toon Calders. 2SCENT: An Efficient Algorithm to Enumerate All Simple Temporal Cycles. *PVLDB*, 11(11): 1441-1453, 2018.
DOI: https://doi.org/10.14778/3236187.3236197

The extended version of this paper is available as [13]. Implementation of all algorithms can be found at `https://github.com/rohit13k/CycleDetection`.

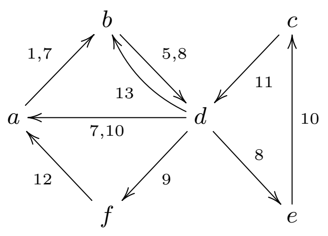
Proceedings of the VLDB Endowment, Vol. 11, No. 11
Copyright 2018 VLDB Endowment 2150-8097/18/07.
DOI: https://doi.org/10.14778/3236187.3236197

1. INTRODUCTION

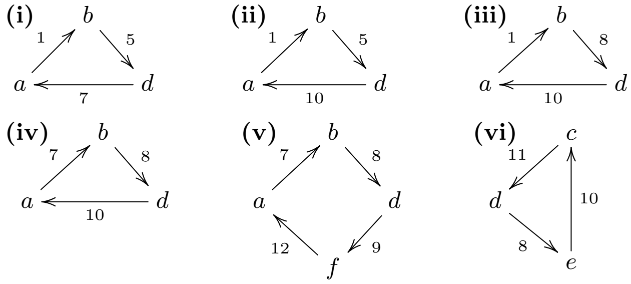
Analyzing the temporal dynamics of a network is becoming very popular. In 2011, Pan et al. [17] studied temporal paths in empirical networks of human communication and air transport, and came to the conclusion that the temporal dynamics of networks are poorly captured by their static structures: "*Nodes that appear close from the static network view may be connected via slow paths or not at all.*" This observation motivated research into temporal patterns in dynamic graphs as an addition to the abundance of works that characterize networks based on their static structures and motifs only. Recently, Paranjape et al. [18] introduced an algorithm for counting the number of occurrences of a given temporal *motif* in a temporal network. In their paper the authors show that datasets from different domains have significantly different motif counts, thus observing that temporal motifs are able to capture differences in the dynamic behavior of temporal networks. Inspired by this line of work, our paper extends it to temporal cycles of any length.

Figure 1b illustrates our notion of a temporal cycle in the temporal graph given in Figure 1a: a sequence of interactions, increasing in time, that starts and ends in the same node. Cycles appear naturally in many problem settings: (1) In stock trading, cyclic patterns could indicate attempts to artificially create high trading volumes; (2) In financial transaction data, specific types of fraud lead to cycles in the interactions [6], and recently, Giscard et al. [5] used simple cycles to evaluate balance in social networks. (3) In biological and neural networks [4], temporal cycles could indicate feedback loops. Notice that in these applications it is essential that the temporal order is respected in the cycles. Consider for instance the last example; feedback loops in a neural network. In order to identify all possible feedback loops in a neural network, a logical first step would be to identify all cycles of interactions between neurons. To have a proper feedback loop it is important that the order of the interactions is consistent with the order in the cycle.

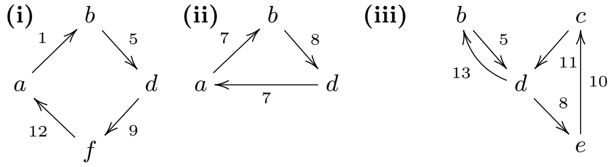
We consider temporal cycles as a essential basic pattern type for temporal networks, and in this paper we study the problem of identifying them all in huge databases of interactions. To avoid spurious cycles stretched out over time we bound the window in which a cycle has to occur to ω. Figure 1c contains some examples of cycles in the static graph which are not considered as they either (i) extend over a too long time window (we used $\omega = 10$), (ii) the interactions do not respect temporal order, or (iii) the cycle is not *simple*



(a) Temporal network with time stamped edges



(b) Instances of Simple Temporal cycles for $\omega = 10$



(c) Instances of patterns which are not Simple Temporal cycles for $\omega = 10$

Figure 1: Example temporal network

in the sense that there are repeated vertices. As we will detail in the related work section, however, the vast literature on finding cycles in static graphs does not easily extend to temporal networks. Therefore we propose a new efficient two-phase algorithm, (2SCENT), for enumerating all simple temporal cycles of bounded timespan.

In the first phase, called the *Source Detection Phase*, we gather candidate root nodes for cycles. The root node of a temporal cycle is the unique node in which the cycle starts and ends. For instance, for the simple cycle shown in Figure 1b(iv), the root node is a. Surprisingly, root nodes of cycles can be found very efficiently in one pass over the data. As a side-result we also get for each cycle its start and end time and a superset of the nodes that appear in the cycle.

In the second phase, for every quadruple of root node, start time, end time, and set of candidate nodes, we run a *constrained Depth First Search* (cDFS) algorithm. This algorithm is inspired by the seminal algorithm of *Johnson* [9]. cDFS performs a depth-first search with backtracking, starting from the root node. In order to avoid unnecessary multiple explorations of the same parts of the interaction graph, for every visited node a so-called *closing time* is maintained that allows to prune previously unsuccessful depth-first traversal paths. In this way we can output all simple cycles rooted at the given node in time $\mathcal{O}(c(n+m))$ where c is the number of cycles and n and m are respectively the number of nodes in the candidate set of the root node and the number of interactions among these nodes in the given time interval. Also this phase sometimes suffers from the peculiarities of interaction networks. To handle the special case of networks with multiple, highly repetitive activities resulting in many similar cycles only differing in a few time stamps, we introduce so-called *path bundles*. A path bundle maintains multiple temporal paths between the same nodes. The cycle finding algorithm is adapted to deal with these path bundles directly, instead of with each of the paths in the bundle individually. In this way we can reduce the number of depth-first traversal paths exponentially.

To validate the algorithm, we ran it on 6 real world data sets. The Experiments consistently show the performance improvements of the extensions and an improvement of two orders of magnitude over our only competitor, the algorithm of *Kumar and Calders* [11]. As a qualitative experiment, we study if temporal cycles can be used to quantify the dynamic behavior of interaction networks. More specifically, we monitor the distribution of frequency and size of simple temporal cycles in different kinds of interaction networks. We find that cycles of higher length are more frequent in data sets such as twitter as compared to the SMS or Facebook data sets. This observation hints that different kinds of information exchange patterns occur in open social networks where people can interact with anyone without a friendship link as compared to closed social network of friends.

2. RELATED WORK

Simple Cycles in a Static Graph. The classical problem of enumerating all simple cycles in a graph has been studied since the early 70s [20, 16, 28, 19, 31, 25, 9, 26]. One algorithm that stands out both in elegance and efficiency is that of Johnson [9]. Johnson's algorithm explores a directed graph depth-first but at the same time uses a combination of blocking and unblocking of vertices to avoid fruitless traversal of paths which will not form a cycle for the currently traversed path. For instance, if during a depth-first exploration to find cycles rooted at a, it is found that there is no path from b to a, b can be blocked such that in other depth-first explorations the paths originating from b are not explored in vain. When backtracking, however, some nodes can become unblocked again. Johnson's algorithm [9] is based upon postponing the unblocking of a node as much as possible. Using an ingenious system of cascading unblocking operations, Johnson's algorithm is able to guarantee a worst case complexity of $\mathcal{O}((n+m)(c+1))$ for enumerating all cycles in a directed graph, where n, m, and c denote respectively the number of nodes, the number of edges, and the number of simple cycles in the graph. Up to the current date, Johnson's algorithm is one of the most efficient algorithms for *directed* graphs. For *undirected* graphs, recently Ferreira et.al [2] presented a more optimal algorithm.

These algorithms work very well for static graphs but cannot be used directly on interaction networks. First of all, cycles in interaction graphs need to respect the temporal order of the interactions, which leads to more complexity.In this paper we provide an extension of Johnson's algorithm for an interaction network. Furthermore, in static networks edges are never repeated while in interaction networks repetitions of interactions are very common. Not taking this aspect of interaction networks into account leads to highly inefficient solutions, a problem we handle by using *path bundles*.

Patterns in temporal graphs. Temporal graphs, also know as interaction [14, 23] or temporal networks [7], are being studied using multiple approaches. One approach is to extend global properties from static graph theory such as page rank [8, 22], shortest path [17, 24, 29], or centrality measures [1, 21] to temporal networks and to introduce efficient algorithms to compute them. Other works focus on better understanding the nature and evolution of such temporal graphs. Recent studies use temporal motifs [10, 18] and their frequency distributions to analyze and characterize temporal graphs. The algorithms in these two papers, however, cannot be used directly for our cycle detection algorithm. For the first paper by Kovanen et al. [10], motifs are considered at a higher level of abstraction. Whereas in our setting all sequences of interactions that form temporal cycles are enumerated, Kovanen et al. [10] would consider a generic temporal cycle of length k as a pattern and count the number of embeddings of this generic pattern. The second paper by Paranjape et al. [18] on the other hand, assumes the same setting as we do. Their work, however, concentrates on efficiently counting the frequency of a specific *given* pattern. In order to apply their algorithm for finding cycles, we would have to run it once for each cycle length. Whereas this is certainly possible in theory, it has a number of disadvantages, such as not knowing for which lengths we need to run the algorithm on the one hand, and the fact that the algorithm of Paranjape et al. [18] requires to first find all embeddings of the pattern in the static graph, without any temporal order or window being considered. A head-to-head comparison with our algorithm, however, would not be fair; the authors are well-aware of this deficiency and for several special cases, such as triangles Paranjape et al. propose efficient adaptations avoiding this costly first step. For cycles, however, no such optimization is described and there is no straightforward solution. The closest to our work is the work by Kumar and Calders [11], who study the same problem, and propose the idea of using simple temporal cycles and their frequency distribution to characterize the information flow in temporal networks. Kumar and Calders [11] introduce a naive algorithm which enumerates all possible temporal paths in a window to find cycles. For interaction networks with large number of temporal paths this algorithm does not scale well. In the empirical evaluation 2SCENT outperforms the algorithm of [11] by a factor of 300 in terms of time. This gain in performance is because it is much more efficient to find roots of cycles than to find the cycles themselves, and once the roots are known, many temporal paths no longer have to be considered.

3. PRELIMINARIES

Let V be a given set of nodes. An interaction is defined as a triplet (u, v, t), where $u, v \in V$, and t is a strictly positive natural number representing the time the interaction took place. Interactions are directed and could denote, for instance, the sending of a message in a communication network. Please note that multiple interactions can appear at the same time. A temporal network $G(V, \mathcal{E})$ is a set of nodes V, together with a set $\mathcal{E}$ of interactions over V. $n = |V|$ denotes the number of nodes in the temporal graph, and $m = |\mathcal{E}|$ the number of interactions.

Definition 1. A *temporal path* between two nodes $u, v \in V$ is a sequence of interactions $p = \langle(u, n_1, t_1), (n_1, n_2, t_2), .., (n_{k-1}, v, t_k)\rangle$ such that $t_1 < t_2 < .. < t_k$ and all interactions in p appear in $\mathcal{E}$. Often we will use the more compact

Algorithm 1 GenerateSeeds

```
Require: Threshold $\omega$, interactions $\mathcal{E}$
Ensure: All nodes $s$, time stamps $t_s$ and $t_e$, and a set $C$
    such that there exists a loop from $s$ to $s$ using only nodes
    in $C$ starting at $t_s$ and ending at $t_e$.
 1: function GenerateSeeds($\omega$, $\mathcal{E}$)
 2:     for $(a, b, t) \in \mathcal{E}$, ordered ascending w.r.t. $t$ do
 3:         if $S(b)$ does not exist then
 4:             $S(b) \leftarrow \{\}$
 5:         $S(b) \leftarrow S(b) \cup \{(a, t)\}$
 6:         if $S(a)$ exists then
 7:             $S(a) \leftarrow S(a) \setminus \{(x, t_x) \in S(a) \mid t_x \leq t - \omega\}$
 8:             $S(b) \leftarrow S(b) \cup S(a)$
 9:             for $(b, t_b) \in S(b)$ do
10:                 $C \leftarrow \{c \mid (c, t_c) \in S(a), t_c > t_b\} \cup \{b\}$
11:                 Output $(b, [t_b, t], C)$
12:                 $S(b) \leftarrow S(b) \setminus \{(b, t_b)\}$
13:         if time to prune then
14:             for all summaries $S(x)$ do
15:                 $S(x) \leftarrow S(x) \setminus \{(y, t_y) \in S(x) \mid t_y \leq t - \omega\}$
```

notation $u \xrightarrow{t_1} n_1 \xrightarrow{t_2} n_2 \ldots \xrightarrow{t_k} v$. $dur(p) := t_k - t_1$ denotes the *duration* of the path, $len(p) := k$ its *length*.

A temporal path p is called a *simple temporal path* if no node appears more than once in p. p is *valid* for a given time window ω if $dur(p) \leq \omega$.

For example, in the temporal graph of Figure 1a, $b \xrightarrow{5} d \xrightarrow{8} e \xrightarrow{10} c \xrightarrow{11} d$ is a temporal path, but it is not a simple temporal path as node d appears more than once in the path. The duration of the path is $11 - 5 = 6$. $b \xrightarrow{5} d \xrightarrow{8} e \xrightarrow{10} c$ is a simple temporal path with duration 5.

Definition 2. A *temporal cycle* with root node u is a temporal path from u to itself. The cycle is called *simple* if each internal node in the cycle occurs exactly once. More specifically, a simple temporal cycle c with root node u consist of a simple temporal path $u \xrightarrow{t_1} n_1 \ldots \xrightarrow{t_{k-1}} v$ followed by an interaction (v, u, t_k) with $t_k > t_{k-1}$. We consider a simple temporal cycle to be *valid for time window* ω if the duration of the cycle is less than or equal to ω.

For example, the cycle in Figure 1c(i) is a simple temporal cycle but is not valid for $\omega = 10$. Please note there could be multiple cycles from the same root node of different length and duration. For example, Figure 1b (i)-(iv) represents 4 different temporal cycles with the same root node a of the same length but with different durations. The cycles in Figure 1b (ii) and (iii) have the same duration and length but still represent different cycles.

Definition 3. **Simple Cycle Enumeration (SCE)** Given a temporal network $G(V, \mathcal{E})$ and a time window ω, enumerate all simple temporal cycles C with $dur(C) \leq \omega$.

In Figure 1a, the solution of SCE with $\omega = 10$ are the cycles of Figure 1b plus $b \xrightarrow{5} d \xrightarrow{13} b$ and $b \xrightarrow{8} d \xrightarrow{13} b$.

4. SOURCE DETECTION PHASE

In this and the next two sections, we will address the problem of efficiently finding all simple temporal cycles in a given temporal network. As temporal networks are generally very large graphs, performing a DFS (Depth First Search) or BFS (Breadth First Search) scan for every node in the network would be very time consuming. Hence, we present a two-phase approach to efficiently find all simple cycles. In the first phase, we pass once over the interactions of the given temporal network to identify the root nodes and the start and end times of all cycles. We also get a set of candidate nodes which form a superset of the nodes present in the cycle. We call this phase the *Source Detection phase*. The details of this phase are given in this section. We also present a memory efficient variation of the source detection phase using Bloom Filters, which requires two passes over the data but is more memory and time efficient for particular cases in which there are many temporal paths. In the second phase, which we will discuss in Section 5, we use the identified root nodes from the first phase to find temporal cycles using a constrained DFS. The details of this phase are given in Section 5. Finally, in Section 6 we present an optimization of our two-phase algorithm for special cases with many repeated interactions.

4.1 Reverse Reachability Summary

We find the source node and candidate sets by maintaining a so-called *reverse-reachability summary* $S(u)$ for all u in V. The reverse reachability summary of u at time t, denoted $S_t(u)$, is defined as the set of pairs (x, t_x) such that there is a temporal path p from x to u starting at time t_x and with $t_x \geq t - \omega$ within the set of interactions up to time stamp t. Maintaining the summary is straightforward; whenever an interaction $a \xrightarrow{t} b$ is processed we add (a, t) to $S(b)$ as it captures the path of length 1 due to this new interaction. Also, every path to a is now extended to b, hence we add all pairs in $S(a)$ to $S(b)$. We remove paths which are older than ω; that is, pairs (x, t_x) such that $t_x < t - \omega$. We call this *old path pruning*. Whenever there is a path from b to b after processing the new interaction $a \xrightarrow{t} b$; that is, there is a pair $(b, t_b) \in S(a)$, we know there is a cycle with b as source node, that starts at t_b and ends at t. Furthermore, every node x in this cycle which was completed by $a \xrightarrow{t} b$ is connected to a and hence there must be a pair $(x, t_x) \in S(a)$. In this way we can also construct a candidate set $\{x \mid \exists (x, t_x) \in S(a) : t_b < t_x < t\}$.

Example 1. Consider the interaction in the example Figure 1a. Before processing the interaction $(d, a, 8)$, the summaries of nodes a and d are $S(a) = \{\}$ and $S(d) = \{(a, 1), (b, 5)\}$ respectively. While processing $(d, a, 8)$ the summary of a is updated to $S(a) = \{(b, 5), (d, 8)\}$ and as there is $(a, 1)$ in the summary of d it generates a seed candidate as $(a, [1, 8], \{b, d\})$. This seed candidate actually corresponds to the simple cycle in Figure 1b(i).

The details of the algorithm are given in Algorithm 1. One detail that still needs clarification is the *inactive node pruning* (steps 13-15). In this step, at regular time instances all pairs (x, t_x) such that $t_x \leq t - \omega$ is removed from the memory. In this way we ensure that memory does not get filled with summaries of nodes which are no longer active. In all our experiments we noticed that the overhead of this step was negligible because when executed regularly, only nodes which were active within the past window of size ω will have a summary, but the memory saving were huge.

Theorem 1. *Let $m = |\mathcal{E}|$, $n = |V|$, W be the number of interactions in a window of size ω, and c the number of valid temporal cycles. Algorithm 1 generates one tuple (a, t_s, t_e, C) for each cycle c that starts and ends in a with respectively an interaction at time t_s and one at time t_e. All nodes of the cycle are in C. Furthermore, for each tuple (a, t_s, t_e, C) output by the algorithm, a corresponding cycle exists. The time complexity for handling one interaction is bounded by $\mathcal{O}((m + c)W)$, and the memory complexity is $\mathcal{O}(\min(n, W)W)$.*

4.2 Improvements using Bloom Filters

Despite the regular pruning, the summaries may still grow very large for large window lengths or large networks, causing out-of-memory problems. This problem occurs for instance when there are many long temporal paths within the window of length ω. Therefore, for such extreme cases, we further refine the source detection phase by using a Bloom filter [3] as summary. A Bloom filter is a compact data structure for representing sets which allows for membership queries. It consists of an array B of q bits and uses k independent hash functions $h_1, \ldots, h_k$ that hash the elements to be stored in the set uniformly over the set of valid indices $1 \ldots q$ for B. Initially all bits in the bitmap index are 0. Whenever a new element a arrives, all bits $h_1(a), \ldots, h_k(a)$ are set to 1. Whenever we need to know if an element x is in the set represented by B, we test if all entries $h_1(x), \ldots, h_k(x)$ are 1. If x was added to the Bloom filter at some point, for sure these bits must all be 1. Notice that there may be false positives if the combined bits set to 1 by the other elements in the set cover all the bits for x. False negatives, however, are impossible. For the exact details on the Bloom filter and how to select optimal values for q and k in function of the number of elements to store in the set and the false positive probability, we refer to [3]. If we have two Bloom filters representing sets S_1 and S_2, we can construct the Bloom filter for their union by taking the bitwise OR of the two Bloom filters. Taking the intersection of two Bloom filters can be done by taking the bitwise AND. In contrast to the union, however, the Bloom filter for the intersection cannot be constructed exactly with this construction. We will denote the bitwise AND (respectively OR) of two Bloom filters B_1 and B_2 with $B_1 \cap B_2$ (respectively $B_1 \cup B_2$).

$S(a)$ will hence be replaced by a Bloom filter $B(a)$, that represents the set of all nodes that can reach a. Whenever an interaction $a \xrightarrow{t} b$ is processed, we test if b is a hit for the Bloom filter of a. If so, b will be listed as a potential cycle source node. Then we union the Bloom filter of $B(a)$ with that of $B(b)$ to get the new Bloom filter for b. Using the Bloom filter approach we guarantee that all summaries have equal (restricted) length and cannot grow unboundedly. Notice, however, that this schema has a number of disadvantages as well. We list them in increasing order of severity: (1) There may be false positives when we test for $b \in S(a)$. This will incorrectly lead to the conclusion that there is a cycle rooted at b. These spurious root nodes, however, will be eliminated in the second phase of the algorithm that will be discussed later. False positives do not affect the correctness of the complete 2SCENT algorithm although they will affect the efficiency. (2) we can no longer apply the old path pruning because the Bloom filter does not contain the information when elements were added to it. We handle this problem by *inactive nodes pruning.* In

Algorithm 2 GenerateSeedsBloom

```
Require: Threshold $\omega$, interactions $\mathcal{E}$
    Hash functions $h_1, \ldots, h_k$, Bloom filter size $q$.
Ensure: Candidate root nodes $s$ with start and end time of
    the cycle and a bloom filter representing the candidate
    set. It is guaranteed that for each temporal simple cycle
    there will be such a four-tuple.
1: function GENERATESEEDSBLOOM($\omega$, $\mathcal{E}$)
2:     $fwSeeds \leftarrow \emptyset$
3:     for $(a, b, t) \in \mathcal{E}$, ordered ascending w.r.t. $t$ do
4:         $fwSeeds \leftarrow fwSeeds \cup$ PROCESSEDGE(a,b,t,$\omega$)
5:     Remove all bloom filters
6:     $bwSeeds \leftarrow \emptyset$
7:     for $(a, b, t) \in \mathcal{E}$, ordered descending w.r.t. $t$ do
8:         $bwSeeds \leftarrow bwSeeds \cup$ PROCESSEDGE(b,a,t,$\omega$)
9:     Output all $(a, [t_s, t_e], (B_f \cap B_b))$ s.t. there exists
       $(a, t_e, B_f) \in fwSeeds$ and $(a, t_s, B_b) \in bwSeeds$ with $0 <$
       $t_e - t_s \leq \omega$

10: function PROCESSEDGE($a$,$b$,t,$\omega$)
11:     $seeds \leftarrow \{\}$
12:     if $B(b)$ does not exist or $|Last(b) - t| > \omega$ then
13:         $B(b) \leftarrow [0, \ldots, 0]$                ▷ Empty bloom filter
14:     Set bits $h_1(a), \ldots h_k(a)$ to 1 in $B(b)$
15:     $Last(b) \leftarrow t$         ▷ Update last modified time stamp
16:     if $B(a)$ exists and $|Last(a) - t| > \omega$ then
17:         if $h_1(b), \ldots, h_k(b)$ all 1 in $B(a)$ then
18:             $seeds \leftarrow \{(b, t, B(a))\}$
19:         $B(b) \leftarrow B(b) \cup B(a)$                    ▷ Bitwise or
20:     if time to prune then
21:         for all summaries $B(x)$ do
22:             if $|Last(x) - t| > \omega$ then remove $B(x)$
23:     return $seeds$
```

inactive nodes pruning, we keep for every node a the last time, denoted $Last(a)$, that $B(a)$ was updated. In this way we can prune all nodes that have not been active within the current window. This pruning mechanism is less effective, but at least bounds the number of summaries that simultaneously need to be held in memory. (3) The last, most severe disadvantage is that because of the use of a Bloom filter we are no longer able to capture the starting time of cycles. Indeed, where $S(a)$ contains pairs (b, t_b), $B(a)$ can only be used to test if there is a pair $(b, ?)$ in $S(a)$. This problem can be resolved with an additional pass through the data. This additional pass is based on the observation that every cycle rooted at node v that starts at t_s and ends at t_e becomes the root node of a cycle starting at t_e and ending at t_s if we reverse time and the direction of all interactions. E.g., the temporal cycle $a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} a$ becomes the inverse temporal cycle $a \xrightarrow{3} c \xrightarrow{2} b \xrightarrow{1} a$. In the end we generate candidates by combining the inverse temporal cycle roots with the normal cycle roots.

Combining these elements we get Algorithm 2. The function *processEdge* is similar to the function *GenerateSeeds* in Algorithm 1 with a difference that instead of the exact set summary $S(a)$, a bloom filter $B(a)$ is maintained and updated. Also, instead of pruning individual nodes in the summary set of $S(a)$ based on the time of addition in the set we reset the bloom filter $B(a)$ if it has not been updated in a window of size ω. As *processEdge* is used for both a forward scan and a backward scan while checking for last update we take an absolute difference of current time and update time in steps 12, 16, and 22. In the end, to find all root nodes with start time, end time, and the bloom filter consisting of the candidate nodes, the interactions are scanned both forward and backwards. In steps 2-4 the forward scan is performed by processing every interaction (a, b, t) to find the end time, root nodes, and candidate sets of all cycles, which are stored in *fwSeeds*. Then in steps 6-8 a backward scan is performed by processing edges in reverse to find the start time, root node, and candidate set for each cycle, which are stored in *bwSeeds*. Finally, in step 9 we merge *fwSeeds* and *bwSeeds* to generate the final seed candidates.

Example 2. Consider again the example of Figure 1a. After the initial forward scan, we will have candidate roots with end time and a Bloom filter for the candidates. For this simple example, *fwSeeds* will contain at least the following candidates: $\{(a, 8, B_4), (a, 10, B_5), (a, 12, B_6), (d, 11, B_7)\}$. After the subsequent backward scan the set of backward seeds will be $\{(a, 1, B_1), (a, 7, B_2), (d, 8, B_3)\}$. The next table lists the compatible pairs and the resulting candidate set:

nr	*fwSeeds*	*bwSeeds*	*Candidate*
1	$(a, 8, B_4)$	$(a, 1, B_1)$	$(a, [1, 8], B_1 \cap B_4)$
2	$(a, 8, B_4)$	$(a, 7, B_2)$	$(a, [7, 8], B_2 \cap B_4)$
3	$(a, 10, B_5)$	$(a, 1, B_1)$	$(a, [1, 10], B_1 \cap B_5)$
4	$(a, 10, B_5)$	$(a, 7, B_2)$	$(a, [7, 10], B_2 \cap B_5)$
5	$(a, 12, B_6)$	$(a, 7, B_2)$	$(a, [7, 12], B_2 \cap B_6)$
6	$(d, 11, B_7)$	$(d, 8, B_3)$	$(d, [8, 11], B_3 \cap B_7)$

In the second step of our algorithm the candidates will generate the following cycles of Figure 1b: Candidate 1 generates (1), candidate 2 is a false positive due to the merging operation and will not generate any cycle (issue (3) mentioned above). Candidate 3 generates (ii) and (iii), candidate 4, (iv), candidate 5, (v), and finally candidate 6, (vi).

THEOREM 2. *Let q be the size of the bloom filters, W be the maximal number of interactions in a window of size ω. The complexity of processing one interaction with* PROCESSEDGE *is $\mathcal{O}(q)$. The time complexity of* GENERATESEEDSBLOOM *is $\mathcal{O}(q(m + c'))$ where c' denotes the number of cycle candidates that are generated by the merge of forward and backward candidates. The memory complexity is $\mathcal{O}(q \min(W, n))$.*

4.3 Combining Root Node Candidate Tuples

An essential last step before we can proceed to the exact cycle finding, is combining seeds for efficiency, and avoiding overlapping seeds. Suppose for instance that there exist 3 cycles rooted at a, with start and end times respectively $[100, 110]$, $[106, 110]$, and $[105, 120]$. GENERATESEEDS will produce three seeds $(s, [100, 110], C_1)$, $(s, [106, 110], C_2)$, and $(s, [105, 120], C_3)$. The second cycle, however, is included in all three seeds and will be generated three times by the cDFS algorithm we will introduce in the next section. Furthermore, we can merge some of the highly overlapping candidates. Consider again the example of Figure 1. For all the cycles rooted at a Figure 1b(i)-(v), the corresponding seeds are $(a, [1, 7], \{b, d\})$, $(a, [1, 10], \{b, d, e, f\})$, $(a, [7, 10], \{b, d, e, f\})$, and $(a, [7, 12], \{b, d, e, f\})$. The first three seeds could be combined into a single seed given as

$(a, [1, 10], \{b, d, e, f\})$. A cDFS run on this single seed will generate all the cycles rooted at a; i.e., cycles 1b(i)-(iv), by considering interactions only in interval $[1, 10]$ between the candidate nodes $\{b, d, e, f\}$. Furthermore, additionally we will also record the starting time of the next seed with the same root and add this information in the seed nodes to obtain the *extended* candidates: $(a, [1, 10], 7, \{b, d, e, f\})$ and $(a, [7, 12], 12, \{b, d, e, f\})$ (The value 12 in the second seed is a dummy value as there is no next seed). cDFS will use these extended candidates $(s, [t_s, t_e], t_n, C)$ to generate exactly those cycles rooted at s, consisting only of vertices in C, starting in the interval $[t_s, t_n[$, and ending the latest at time t_e. By adding the restriction on t_n we avoid duplicate cycle generation. The algorithm to combine seeds rooted at a single node s is given in Algorithm 3. It starts with sorting all candidates on start time ascending and end time descending. Subsequently it gets the first non-merged candidate and merges it with all following compatible candidates. This procedure is repeated until all candidates have been processed. In this way we are often able to compress the list of candidates considerably.

Algorithm 3 Combining Root Node Candidate

```
Require: List of cycle seeds $\mathcal{C}$ for a root node $s$. Each seed
    is of the form $(s, [t_s, t_e], C)$, window length $\omega$
Ensure: Combined candidates
 1: function CombineSeeds($\mathcal{C}$,$\omega$)
 2:     Sort $\mathcal{C}$ on $t_s$ ascending, then $t_e$ descending.
 3:     while $\mathcal{C}$ not empty do
 4:         Let $(s, [t_s, t_e], C)$ be first in $\mathcal{C}$
 5:         Let Compatible be the maximal prefix of $\mathcal{C}$ such
            that for all $(s, [t'_s, t'_e], C') \in$ Compatible it holds that
            $t'_e < t_s + \omega$
 6:         $\mathcal{C} \leftarrow \mathcal{C} \setminus$ Compatible
 7:         if $\mathcal{C}$ is empty then $t_n \leftarrow t_s + \omega$
 8:         else
 9:             Let $(s, [t'_s, t'_e], C')$ be first in $\mathcal{C}$
10:             $t_n \leftarrow t'_s$
11:         $t_{max} \leftarrow \max\{t'_e \mid (s, [t'_s, t'_e], C') \in$ Compatible$\}$
12:         $C_{all} \leftarrow \bigcup\{C' \mid (s, [t'_s, t'_e], C') \in$ Compatible$\}$
13:         Output $(s, [t_s, t_{max}], t_n, C_{all})$
```

Theorem 3. *Algorithm 3 ensures that for every temporal cycle rooted at s, which start at t_s and end at t_e, there is exactly one extended seed $(s, [t'_s, t'_e], t_n, C)$ that contains the cycle; that is: all nodes of the cycle are in C, $t_s \in [t'_s, t_n[$, and $t_e \in [t'_s, t'_e]$.*

5. CONSTRAINED DFS

After finding candidates, we find the exact cycles. For each extended candidate $(s, [t_s, t_e], t_n, C)$ we run our constrained Depth-First Search to find all cycles represented by this candidate. Algorithm 7 gives the complete procedure. We will now step by step describe how this procedure works.

We apply a depth-first procedure to find all temporal paths in a dynamic graph. If the path reaches a node which is the same as the start node, we output it as a cycle. We start with a given node s and a start time t_s. All edges that branch out of s at this time stamp are now recursively explored. A pure depth-first exploration, however, has the disadvantage that some unsuccessful paths will be explored

Algorithm 4 Unblock

```
Require: Node $v$ that gets a new closing time $t_v$.
    Global: interactions $\mathcal{E}$, closing times $ct(v)$ and unblock
    list $U(v)$ for all nodes $v \in V$.
Ensure: Recursive unblocking of the nodes.
 1: function Unblock(Node $v$, time stamp $t_v$)
 2:     if $t_v > ct(v)$ then
 3:         $ct(v) \leftarrow t_v$
 4:         for $(w, t_w) \in U(v)$ do
 5:             if $t_w < t_v$ then
 6:                 $U(v) \leftarrow U(v) \setminus \{(w, t_w)\}$
 7:                 $T[w, v] = \{t \mid (w, v, t) \in \mathcal{E}\}$
 8:                 $T \leftarrow \{t \in T[w, v] \mid t_v \leq t\}$
 9:                 if $T \neq \emptyset$ then
10:                     $U(v) \leftarrow U(v) \cup \{(w, \min(T))\}$
11:                 $t_{max} \leftarrow \max\{t \in T[w, v] \mid t < t_v\}$
12:                 Unblock($w$,$t_{max}$)
```

Algorithm 5 Add to unblock list

```
Require: Unblock list $U(v)$ of node $v$, pair $(w, t)$ to be
    added
Ensure: New unblock list $U(v)$ with $(w, t)$ added.
 1: function Extend($U(v)$,$(w, t)$)
 2:     if there is an entry $(w, t') \in U(v)$ then
 3:         if $t' > t$ then $U(v) \leftarrow U(v) \setminus \{(w, t')\} \cup \{(w, t)\}$
 4:     else $U(v) \leftarrow U(v) \cup \{(w, t)\}$
```

over and over again. Consider for instance the example in Figure 2 *without the dotted lines*. As there exist 2 paths from a to c, an exhaustive depth-first exploration of all paths will visit node c two times, and each time the subgraph formed by h, j, and k will be explored again. In order to avoid such fruitless repeated explorations, we will keep track of the success status of different nodes in earlier depth-first explorations of the dynamic network. This information is stored in the form of a so-called "closing time" of a node. Intuitively, node v having closing time $ct(v)$ indicates that there do not exist paths back to a from node v that start at time $ct(v)$ or later. Hence, if during the depth-first exploration, we arrive at a node on or after its closing time, then we can abort our search. So, while exploring node h, arriving there at 11, we will notice that there are no paths from h back to a and hence its closing time will become 11 and h will never be expanded again. Similarly, after the first time we visit node c, we will notice that the last path from c back to a starts at timestamp 7, so its closing time will become 7. Due to this update in closing time of c any depth-first exploration of c will be aborted from timestamp 7 onwards.

Let's illustrate the principle with our example graph. For the subsequent steps we will show how the closing times of the nodes evolve and how this saves us costly repetitions of useless explorations. For now the reader does not need to worry about how the closing times are affected by backtracking to find additional solutions as this will be treated in detail right after the example.

- $a \xrightarrow{1} b$: $ct(b)$ becomes 1; this node cannot be used to extend the path without violating the simplicity condition;
- $b \xrightarrow{5} c$: $ct(c)$ becomes 5;

Algorithm 6 Algorithm AllPaths

```
Require: Prefix path $s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_k$ that starts in
    target node $s$.
Ensure: All simple temporal paths in $G(V,\mathcal{E})$ from $v_1$ to
    $s$, starting with the given prefix are output. The return
    value is false if no such path exists, otherwise it is true.
 1: function AllPaths($pr = s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k$)
 2:     $v_{cur} \leftarrow v_k$, $t_{cur} \leftarrow t_k$
 3:     $ct(v_{cur}) \leftarrow t_{cur}$, $lastp \leftarrow 0$
 4:     $Out \leftarrow \{(v_{cur}, x, t) \in \mathcal{E} \mid t_{cur} < t\}$
 5:     $N \leftarrow \{x \in V \mid (v_{cur}, x, t) \in Out\}$
 6:     if $s \in N$ then
 7:         for $(v_{cur}, s, t) \in Out$ do
 8:             if $t > lastp$ then $lastp \leftarrow t$
 9:             Output $pr \cdot \langle (v_{cur}, s, t) \rangle$
10:     for $x \in N \setminus \{s\}$ do
11:         $T_x \leftarrow \{t \mid (v_{cur}, x, t) \in Out\}$
12:         while $T_x \neq \emptyset$ do
13:             $t_m \leftarrow \min(T_x)$
14:             $pass \leftarrow False$
15:             if $ct(x) \leq t_m$ then $pass \leftarrow False$
16:             else $pass \leftarrow$ AllPaths($pr \cdot \langle (v_{cur}, x, t_m) \rangle$)
17:             if not $pass$ then
18:                 $T_x \leftarrow \emptyset$
19:                 Extend($U(x)$,$(v_{cur}, t_m)$)
20:             else
21:                 $T_x \leftarrow T_x \setminus \{t_m\}$
22:                 if $t_m > lastp$ then $lastp \leftarrow t_m$
23:     if $lastp > 0$ then Unblock($v_{cur}$,$lastp$)
24:     return ($lastp \neq 0$)
```

- We explore recursively all paths that start with $c \xrightarrow{11} h$. No paths are found, hence during this recursion $ct(h)$, $ct(j)$, and $ct(k)$ become respectively 11, 13, and 14;
- Via recursive calls we find a path from c that start with $c \xrightarrow{7} e$ and $c \xrightarrow{6} d$. We hence derive that the latest path leaving c starts at time 7. Hence, when backtracking, $ct(c)$ becomes 7, and during the recursive calls also the closing times of the other nodes are updated.

In order to find additional paths, we backtrack and find the next solution. Suppose now that we already explored the subspace of all cycles that start with $a \xrightarrow{1} b$. At this point in time the closing times are as follows:

a	b	c	d	e	f	h	j	k
–	5	7	8	10	12	11	13	14

- $a \xrightarrow{5} c$ can be explored next, because $5 < ct(c) = 7$.
- From c we cannot go to node h because $11 \not< ct(h)$.
- From there on we continue to find our last 2 paths.

So far so good, but until now we have been ignoring a major problem with the closing times when backtracking to find the next solution: while backtracking, the path becomes shorter again, and nodes become available again which on its turn may affect the correctness of the closing times. We illustrate this problem by slightly extending the example in Figure 2 by adding the dotted lines. When exploring all paths starting with the edge $a \xrightarrow{1} b$, the node b temporarily gets $ct(b) = 1$ to force that our cycles are simple. As a

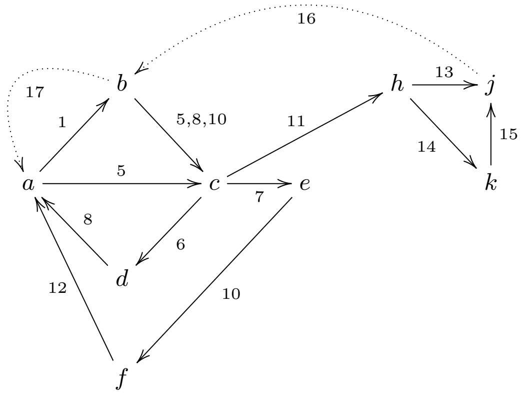


Figure 2: Example temporal network with simple cycles

Algorithm 7 Dynamic Depth-First Simple Cycle Search

```
Require: Source node $s \in V$
    Global: Interaction network $G(V,\mathcal{E})$; closing time $ct(v)$
    and unblock list $U(v)$ for all nodes $v \in V$; Timestamp
    $t_s$,$t_e$ and $t_n$; Set of candidates $C \subseteq V$
Ensure: All simple temporal cycles in $\mathcal{E}$ rooted at $s$ start-
    ing in interval $[t_s, t_n[$ and ending before $t_e$, using only
    vertices of $C$.
1: function Cycle($s$)
2:     $\mathcal{E} \leftarrow \{(u,v,t) \in \mathcal{E} \mid u,v \in C, t \in [t_s,t_e]\}$ ▷ Reduce $G$
3:     $V \leftarrow C$
4:     for $x \in C$ do
5:         $ct(x) \leftarrow \infty$, $U(x) \leftarrow \emptyset$
6:     for $(s,x,t) \in \mathcal{E} | t < t_n$ do
7:         AllPaths($s \xrightarrow{t} x$)
```

result, when recursively exploring all paths with prefix $a \xrightarrow{1} b \xrightarrow{5} c$, we will conclude there is no path from h, k, and j back to a and set their closing times to 11, 13, and 14 respectively. As a result, later on, when exploring all paths with prefix $a \xrightarrow{1} b \xrightarrow{8} c$ and $a \xrightarrow{1} b \xrightarrow{10} c$, we will correctly abort exploration of the branch below h. However, when the search continues, at a certain point we will have explored all paths starting with $a \xrightarrow{1} b$, and we are back at node a. The closing time of b is set to 17 because of the cycle $a \xrightarrow{1} b \xrightarrow{17} a$. We continue exploring all paths that start with $a \xrightarrow{5} c$. It is at this very moment that things start becoming ugly. Indeed, at this point in time, we do have to explore the branch below h, because now there is a cycle that involves h, namely $a \xrightarrow{1} c \xrightarrow{11} h \xrightarrow{13} j \xrightarrow{16} b \xrightarrow{17} a$! So, what went wrong? The first time we visited node h, node b was blocked as it appeared on the path from a to h. Therefore, we correctly concluded that h should be blocked, too. This situation remained until the point that b became unblocked because of backtracking. At that point, in fact, the closing time of h should have been reconsidered. The mechanism to realize the correct update of the closing times is as follows: whenever we limit the closing time of a node, at the same time we also evaluate under which conditions the closing time of the node can increase again. In the case of node j, we see that there is an outgoing edge with time stamp 16

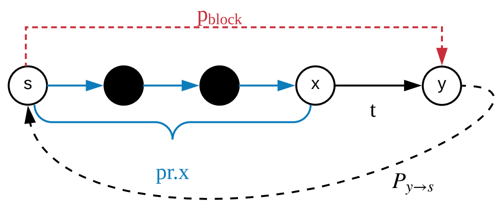


Figure 3: Illustration of the concepts introduced in the proof of completeness of AllPaths.

to node b with closing time 1. Hence, from the moment on that the closing time of b increases to above 16, the closing time of j should increase to 16. For this purpose, we add for every node an "unblock list" $U(v)$ that contains a list of nodes and thresholds (w, t). From the moment on that the closing time of v exceeds again the threshold t, for each pair (w, t) in $U(v)$, the closing time of node w will have to be adapted as well. In our example this amounts to adding $(j, 16)$ to $U(b)$. Whenever we increase the closing time of any node v in the graph, we will go over its unblock list and unblock the other nodes as needed. Notice that unblocking a node may result in a cascade of unblock operations; indeed, in our example, unblocking b causes j to become unblocked, which on its turn causes h and k to become unblocked. The pseudo code of the algorithm is given in Algorithms 4, 6, and 7.

THEOREM 4. ***Correctness.*** *CYCLE(s) returns all simple cycles rooted at s starting in interval $[t_s, t_n[$ and ending before t_e.*

Proof Sketch First of all, it is important to realize that the cDFS algorithm is a truncated depth-first search: all paths are explored from a node s that a normal depth-first search would also explore, except for paths that (a) do not respect the temporal order, (b) contain duplicate nodes other than s, or (c) paths that are blocked because of a closing time that is too low. The fact that it is a depth-first search together with (a) and (b) guarantees the correctness of each simple temporal cycle that is output. For completeness, from (a) and (b) it is easy to see that these cases do not restrict the completeness of the algorithm. Hence, what is left to show is that whenever there exists a simple temporal cycle $a_1 \xrightarrow{t_1} a_2 \ldots \xrightarrow{t_n} a_n \xrightarrow{t_n} a_1$, none of the edges $a_{i-1} \xrightarrow{t_i} a_i$ is blocked because of the closing time of a_i at the moment that $a_1 \xrightarrow{t_1} a_2 \ldots \xrightarrow{t_{i-1}} a_{i-1}$ is explored in the depth-first search. Suppose, for the sake of contradiction, that nevertheless at some point in the algorithm, the interaction $x \xrightarrow{t} y$ is blocked while there exists an extension of the current prefix to s that uses this edge. Let $p_{y\to s}$ be this path such that $pr \cdot x \xrightarrow{t} y \cdot p_{y\to s}$ is the simple temporal cycle that is not found. Furthermore, we can assume without loss of generality that this is the first time this happens. $x \xrightarrow{t} y$ must have gotten blocked in another path p_{block} that reached y (only the closing times of the last nodes on a path can decrease). This situation is depicted in Figure 3. It is clear that $p_{block} \cdot p_{y\to s}$ is a temporal cycle. This temporal cycle is either simple, in which case y won't get blocked, or the path is not simple, which means that p_{block} contains at least one of the nodes of $p_{y\to s}$. In the full proof [13] it is shown, however, that in such case there is a chain of unblock operations which will be invoked when the depth-first exploration back tracks from the path p_{block}, and it can be shown that when the last common node between p_{block} and $p_{y\to s}$ gets unblocked, $x \xrightarrow{t} y$ must be free again (that is: $ct(y) > t$). This contradicts our earlier assumption that the interaction $x \xrightarrow{t} y$ is blocked and hence proves the theorem. An important invariant that facilitates the full proof is the following consistency between closing times and unblock lists that is guaranteed at crucial times: whenever there is a path from y to s that starts at time t and does not intersect the current prefix, $ct(y) > t$. In all other cases, $ct(y) \leq t$, and $(x, t') \in U(y)$ with $t' \leq t$. □

THEOREM 5. ***Complexity.*** *Let $m = |\mathcal{E}|$ and $n = |V|$. We can implement CYCLE(s) in such a way that in between two cycles being output, CYCLE(s) takes at most $\mathcal{O}(m + n)$ steps. Hence, if there are c cycles in the network, the total time complexity to find all of them is $\mathcal{O}((c + 1)(m + n))$.*

Proof Sketch The proof of this theorem is based on the observation that the only way to unblock an edge $x \xrightarrow{t} y$; that is, lower the closing time of y to lower than t, is by a call to UNBLOCK, which only happens when a cycle is output. Whenever a cycle $a_1 \xrightarrow{t_1} a_2 \ldots \xrightarrow{t_n} a_n \xrightarrow{t_n} a_1$ is found, UNBLOCK will be executed for a_n (for the prefix $a_1 \xrightarrow{t_1} a_2 \ldots \xrightarrow{t_n} a_n$), then for a_{n-1} (for the prefix $a_1 \xrightarrow{t_1} a_2 \ldots \xrightarrow{t_{n-1}} a_{n-1}$), etc, until it is called for a_1 (for prefix a_1). Each of the calls to UNBLOCK, however, unblock different interactions. Indeed, it can be shown that, if we are in a call to ALLPATHS, and the prefix is $a_1 \xrightarrow{t_1} a_2 \ldots \xrightarrow{t_{i-1}} a_i$, a call to UNBLOCK, only unblocks interactions $x \xrightarrow{t} y$ such that there is a temporal path to the source node a_1 that uses this interaction and a_i, and at the same time there is no path to the source node a_1 that does not intersect te prefix anywhere. This seemingly cumbersome condition is actually quite intuitive: UNBLOCK only unblocks interactions that at the moment cannot be used in a path to the source node a_1, but once a_i becomes available again, they can be used again. This condition immediately leads to the conclusion that every interaction can become unblocked at most once in between two cycles are being output. Therefore, since the depth-first search blocks the interactions it visits, every interaction can be traversed at most twice in between two cycles are output. Hence, either a cycle is output after $\mathcal{O}(m)$ steps, or all interactions will be blocked and the algorithm stops. The term $\mathcal{O}(n)$ in the theorem is for the initialization of the closing times of all nodes at the start of the algorithm. We furthermore refer to the full proof in the extended version of the paper [13] for an in-depth discussion on the data structures that can be used in order to guarantee that we can run the algorithm without the need to inspect blocked edges. □

6. PATH BUNDLES

The algorithm presented in the last section still has one big disadvantage: especially in the presence of repeated edges the same paths and cycles can be explored over and over again. Consider for instance the example in Figure 4. In this example there are $3^6 = 729$ cycles and each of them will be generated separately. There will be one call starting with a, 3 for $a \to b$, 9 for $a \to b \to c$, etc. A lot of this work could be avoided though by combining the computations for multiple edges and paths. It is exactly for this purpose that we introduce the following notion of a path bundle.

Definition 4. A *path bundle* B in an interaction network $G(V, \mathcal{E})$ between nodes v_1 and v_{k+1} consists of a sequence of vertices $v_1, \ldots, v_{k+1}$, and sets of timestamps $T_1, \ldots, T_k$ such that for all $i = 1 \ldots k$, $t \in T_i$ it holds that $(v_i, v_{i+1}, t) \in \mathcal{E}$. We will denote the path bundle B by $v_1 \xrightarrow{T_1} v_2 \xrightarrow{T_2} \ldots \xrightarrow{T_k} v_{k+1}$.

The *set of temporal paths represented by* B, denoted $\mathcal{P}(B)$ is defined as:

$$\mathcal{P}(B) := \{v_1 \xrightarrow{t_1} v_2 \ldots \xrightarrow{t_k} v_{k+1} \mid \forall i : t_i \in T_i \;\&\; t_1 < \ldots < t_k\}$$

A path bundle is called *minimal* if for all $i = 1 \ldots k$, $t \in T_i$ it holds that

$$\mathcal{P}(v_1 \xrightarrow{T_1} \ldots v_i \xrightarrow{T_i \setminus \{t\}} \ldots \xrightarrow{T_k} v_{k+1}) \subsetneq$$

$$\mathcal{P}(v_1 \xrightarrow{T_1} \ldots v_i \xrightarrow{T_i} \ldots \xrightarrow{T_k} v_{k+1})$$

LEMMA 1. *Let B be a path bundle. There exists a unique minimal path bundle B' such that $\mathcal{P}(B) = \mathcal{P}(B')$*

6.1 Expanding a Bundle

In order to extend our algorithm to work with path bundles instead of individual paths, we need to extend all operations performed on paths in the algorithm to bundles. The first operation we consider is extending the path with an extra edge. This operation is easy enough, as we can just add the edge with all its timestamps to the bundle. We do want, however, to keep the bundles minimal for efficiency reasons. Algorithm 8 does exactly that; it extends a bundle with an edge while maintaining the minimality of the bundle.

Let's illustrate with an example. Suppose we have a path bundle $a \xrightarrow{1,5,7} b \xrightarrow{3,8} c$ which we want to extend with the edges $c \xrightarrow{2,4,7} d$. Since there is no edge from b to c earlier than timestamp 3, we can prune away 2 from the paths between c and d. Furthermore, the last edge between c and d has timestamp 7, so all edges between b and c later than 7 should be removed. Only the edge with timestamp 3 remains between c and d which causes the timestamps 5 and 7 between a and b to be removed. Hence, the result of the extension is: $a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{4,7} d$.

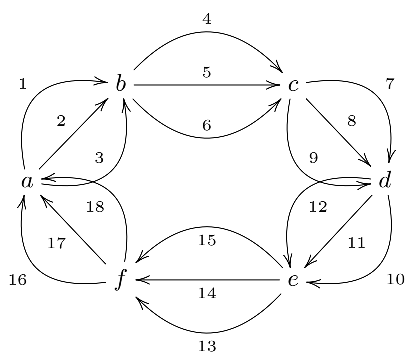


Figure 4: Example temporal network with simple cycles having multiple repeated edges

LEMMA 2. *Given a minimal bundle B between u and v and a bundle $v \xrightarrow{T} w$, Algorithm 8 returns a minimal bundle B' such that $\mathcal{P}(B')$ consists of all temporal paths from u to w that can be constructed by extending a path from $\mathcal{P}(B)$ with an edge from $v \xrightarrow{T} w$.*

6.2 Extending the Algorithm to Bundles

By directly manipulating path bundles instead of individual paths we can significantly reduce the number of recursions needed as well as output the cycles much more compactly. In algorithm 9 we provide extensions of the algorithm presented in 6 to consider the path bundle notion. There is not much change in algorithm 7 except at step 7 where instead of looking for path from x to the root node s using algorithm 6, a path bundle is searched using algorithm 9. The output of the algorithm 9 is not all the simple temporal cycles as we required, but a more compact representation of cycles using the path bundles.

6.3 Counting the Number of Paths in a Bundle

For some applications we need the exact number of paths represented by a bundle. This number, however, is not entirely straightforward to obtain efficiently. Indeed, we may easily come up with a recursive procedure that generates all valid combinations of the timestamps, but that would somewhat defy the purpose of the bundles, which is exactly to avoid such costly individual treatment of the paths.

Luckily it is not hard to develop a dynamic algorithm to count the number of paths in a path bundle. We can iteratively compute the number of paths in a bundle $v_1 \xrightarrow{T_1} \ldots \xrightarrow{T_k} v_{k+1}$ by considering all the prefixes of the bundle in increasing length. For each prefix $P_i = v_1 \xrightarrow{T_1} \ldots \xrightarrow{T_i} v_{i+1}$, the number of paths are stored on a heap H_i. For each end time t of a path in P_i, the number of paths n ending at that time or earlier is stored as a pair (t, n) on the heap. The heap H_{i+1} can easily be computed based on T_i and H_i. Due to space constrained full details of this algorithm are provided in the technical report [13].

7. EXPERIMENTS

We evaluated the performance of our algorithms on 6 different real world temporal networks. The performance results presented in this section are for a C++ implementation of our algorithm. All experiments were run on a simple desktop machine with an Intel Core i5-4590 CPU @3.33GHz

Algorithm 8 Extending a path bundle with an edge bundle

```
Require: Minimal path bundle $B = v_1 \xrightarrow{T_1} \ldots \xrightarrow{T_k} v_{k+1}$, edge
    bundle $E = v_{k+1} \xrightarrow{T_{k+1}} v_{k+2}$
Ensure: Minimal path bundle with all valid paths composed of $B$ and an edge of $E$.
1: function EXPAND($v_1 \xrightarrow{T_1} \ldots \xrightarrow{T_k} v_{k+1}, v_{k+1} \xrightarrow{T_{k+1}} v_{k+2}$)
2:     $T'_{k+1} \leftarrow \{t \in T_{k+1} \mid t > \min(T_k)\}$
3:     if $T'_{k+1} = \emptyset$ then
4:         return $(v_1 \xrightarrow{\emptyset} \ldots v_i \xrightarrow{\emptyset} \ldots \xrightarrow{\emptyset} v_{k+2})$
5:     for $i = k$ down to 1 do
6:         $T'_i = \{t \in T_i \mid t < \max(T'_{i+1})\}$
7:     return $(v_1 \xrightarrow{T'_1} \ldots v_i \xrightarrow{T'_i} \ldots \xrightarrow{T'_{k+1}} v_{k+2})$
```

Algorithm 9 Algorithm AllBundles

```
Require: Prefix bundle $B$ starting in node $s$
    Global: Interaction network $G(V,\mathcal{E})$, closing times $ct(v)$, unblock list $U(v)$ for all nodes $v \in V$, latest timestamp $t_e$ in $\mathcal{E}$.
Ensure: All simple temporal paths in $G(V,\mathcal{E})$ from $x$ to $v_e$, prefixed with $path$.
1: function AllBundles($B = s \xrightarrow{T_1} v_1 \xrightarrow{T_2} \dots \xrightarrow{T_k} v_k$)
2:     $t_{cur} \leftarrow \min T_k$, $v_{cur} \leftarrow v_k$
3:     $ct(v_{cur}) \leftarrow t_{cur}$, $lastp \leftarrow 0$
4:     $Out \leftarrow \{(v_{cur}, x, t) \in \mathcal{E} \mid t_{cur} < t \leq ct(x)\}$
5:     $N \leftarrow \{x \in V \mid (v_{cur}, x, t) \in Out\}$
6:     if $s \in N$ then
7:         $T \leftarrow \{t \mid (v_{cur}, s, t) \in Out\}$
8:         $t \leftarrow \max(T)$
9:         if $t > lastp$ then $lastp \leftarrow t$
10:        Output $Expand(B, v_{cur} \xrightarrow{T} s)$
11:    for $x \in N \setminus \{s\}$ do
12:        $T_x \leftarrow \{t \mid (v_{cur}, x, t) \in Out\}$
13:        $T'_x \leftarrow \{t \in T_x \mid t < ct(x)\}$
14:        if $T'_x \neq \emptyset$ then
15:            $last_x \leftarrow AllBundles(Expand(B, v_{cur} \xrightarrow{T'_x} x))$
16:            if $last_x > lastp$ then $lastp \leftarrow last_x$
17:            $t_m \leftarrow \min\{t \in T_x \mid t > last_x\}$
18:            Extend($U(x)$,($v_{cur}, t_m$))
19:    if $lastp > 0$ then
20:        Unblock($v_{cur}$,$lastp$)
21:    return $lastp$
```

CPU and 16 GB of RAM, running the Linux operating system. The code and instructions to run the experiments are available online [1].

7.1 Dataset

All datasets except `SMS` [30], `Facebook` [27] and `USElection` [12] were obtained from the SNAP repository [15]. The characteristics of the datasets are given in Table 1. While running the experiments we choose smaller windows for the high frequency dataset `SMS`, `Facebook`, `USElection`, and `Higgs` whereas for the low frequency datasets `Stackoverflow` and `Wiki-talk` a longer window size were considered.

The exact meaning of a temporal cycle $a_1 \xrightarrow{t_1} a_2 \dots a_n \xrightarrow{t_n} a_1$ is that there is an interactions from a_1 to a_2 at time t_1, followed by and itereaction at t_2 from a_2 to a_3, and so on, followed by an interaction at time t_n from a_n back to a_1. For `SMS`, for instance, this means that a_1 sms'ed a_2 followed by a_2 sms'ed a_3, etc., until a_n sms'ed back to a_1. Such pattern could be of interest for application-oriented follow-up research such as sociological studies of communication patterns; although speculative at this stage, we think that these patterns could be indicators of feedback mechanisms between peers in which opinions are formed, reinforced and strengthened over time.

7.2 Performance Evaluation

Effect of bloom filter: The efficiency and effectiveness of the bloom filter depends on the Bloom filter size and the number of hash functions used. For our experiments, we used a *projected element count* of 500 and *false positive probability* of 0.0001, which results in a filter of size 9592 using 13 hash functions. Using the bloom-filter-based approach for the SD phase is not always efficient. This is mostly because of two reasons: (1) in the Bloom Filter approach we have to scan the data twice; and (2) creating bloom filters for data sets where the candidate set is very small is an overkill. Hence, as long as the candidate set size is not getting so large that it stresses memory usage and set operations like union and cardinality test, the set-based ap-

[1] Code: `https://github.com/rohit13k/CycleDetection`

Table 1: Characteristics of interaction network along with the time span of the interactions as number of days.

Dataset	$n[.10^3]$	$m[.10^3]$	**Days**
`Facebook`	46.9	877.0	1592
`SMS`	44.1	545	338
`Higgs`	304.7	526.2	7
`Stackoverflow`	2464.6	16266.4	2774
`Wiki-talk`	1140	7833.1	2320
`USElection`	233.8	1000	10 hours

Table 2: Time and Memory Comparison between Exact set based and bloom filter approach to find root candidates.

Dataset	ω	Time(seconds)		Memory(MB)	
		Exact	Bloom	Exact	Bloom
`Facebook`	1 hour	**4**	12	**20**	225
	10 hours	**6**	17	**24**	375
`SMS`	1 hour	**12**	40	**27**	730
	10 hours	**50**	59	**112**	972
`Higgs`	1 hour	**4**	8	**114**	170
	10 hours	45	**10**	3048	**325**
`Stackoverflow`	1 day	**78**	399	**26**	1578
	1 week	**138**	454	**346**	2309
`Wiki-talk`	10 hours	**66**	223	**98**	3541
	1 day	**147**	344	**269**	5675
`USElection`	1 hour	**20**	21	**157**	315
	10 hours	-	**27**	-	**700**

Table 3: Effect of pruning (P) versus no pruning (NP) on Time and Memory usage.

DataSet	ω	Time(sec)		Memory(MB)	
		P	NP	P	NP
`Facebook`	1 hour	**3.9**	4.1	**9**	25
	10 hours	**4.9**	5.1	**11**	28
`SMS`	1 hour	**11.6**	12.1	**16**	51
	10 hours	**45.6**	46.1	**41**	90
`Higgs`	1 hour	4.1	**3.8**	**103**	177
	10 hours	44.3	**41.6**	**3037**	3295
`Stackoverflow`	1 day	**79.7**	97.4	**26**	1441
	1 week	**112.3**	130.8	**343**	2184
`Wiki-talk`	10 hours	**58.5**	62.5	**98**	1231
	1 day	**129**	133.5	**269**	3174

proach is faster than the bloom-filter-based approach. The summary set size becomes very large for interaction networks in which the ratio of the number of interactions over the number of nodes is high. This is the case for Higgs and USElection with ω set to 10 hours. In this case, the Bloom-filter-based approach is the best approach because of the time and memory savings it provides. In our experiments, for USElection, the Exact-set-based approach ran out of memory after 18 minutes, whereas the Bloom-filter-based approach finished within 27 seconds taking only 700 MB of space. More results for time and memory consumption in the SD phase are shown in table Table 2.

For deciding when to use Bloom filters, we remark that it is clear that they become useful only when there are many temporal paths to be maintained. This situation is more likely to occur when the ratio of edges to nodes is high in the windows. This property could be used to estimate whether or not to use the optimization. Another, easier way to estimate, however, is by running the algorithm on a few windows of the dataset. Alternatively, it is straightforward to develop a hybrid version of the algorithm that switches automatically to the use of Bloom filters whenever the memory usage reaches a certain limit; e.g., 90% of available memory. It is, however, unlikely that such hybrid version would be very useful given the ease with which we could run the estimation on a part of the dataset.

Effect of Pruning: We also tested the effect of inactive node pruning in the SD Phase. We ran pruning after processing every batch of 100,000 interactions. As expected, pruning has a huge impact on the memory requirements of the SD Phase. For instance, the memory requirements reduced by a factor of 55 in case of Stackoverflow for a 1 day window. This is because there are too many source nodes and most of them become inactive very quickly. As such, removing their summaries from the memory resulted in a huge gain in memory usage and runtime. In the case of Higgs, however, the number of source nodes is very low and they remain active throughout the whole duration of the dataset resulting in much less memory savings and a modest increase in runtime. In all other cases, however, there are significant memory and time savings due to regular pruning. The results are shown in Table 3.

Effect of Bundling: As expected, using the path bundle approach is never slower than using the simple path approach. On the other hand, in cases where there are multiple repeated edges such as Higgs for a window of 10 hours, we get a speedup of up to 12 times thanks to the path Bundles. The results are shown in Table 4.

Runtime for Complete Cycle Enumeration. Finally, we also compare the total runtime of finding all cycles using 2SCENT with exact set and path bundles to the algorithm presented by Kumar and Calders [11] (Naive algorithm). As 2SCENT is a two-phase algorithm we compare the combined time taken by both phases with the runtime of the Naive algorithm. We observe that for small networks with less frequent interactions, such as Facebook, or for medium-sized networks with a small window length ω, such as SMS with a window of 1 hour, or for large networks with very infrequent interactions, such as Mathoverflow with a 1 day window, the Naive algorithm outperforms 2SCENT and its variants. This is because in these cases there are only few temporal paths to be enumerated which easily fit in memory. Hence a brute force approach as proposed in [11] is feasible. But when we run on larger interaction networks or with larger window lengths, 2SCENT outperforms the Naive algorithm with respect to runtime by a factor of up to 300. The massive gain in performance is due to the fact that the Naive algorithm maintains and updates *all* temporal paths whereas 2SCENT needs to enumerate only paths from nodes that are the starting point of cycles, and does so in a memory-efficient way; one by one, and using a constrained depth-first search. As a result 2SCENT is able to deal with datasets such as Higgs, Stackoverflow, and Wiki-talk, even for high window lengths, whereas the Naive algorithm crashes due to the high number of temporal paths it is maintaining in memory. The results are presented at Table 5.

Table 4: Time comparison (in seconds) to find cycles using Bundle path and without Bundle path.

Dataset	ω	Without Bundle	With Bundle
Facebook	1 hour	4.7	**3.9**
	10 hours	9.4	**7.3**
SMS	1 hour	24.5	**10.3**
	10 hours	104.6	**21.34**
Higgs	1 hour	2.65	**2.26**
	10 hours	1526.5	**136.6**
Stackoverflow	1 day	**62.7**	63.3
	1 week	147.7	**118.4**
Wiki-talk	10 hours	693.9	**320.2**
	1 day	2356	**828**

Table 5: Time Comparison between Naive and 2SCENT to find all cycles.

DataSet	ω	Naive	2SCENT
Facebook	1 hour	**6.5 sec**	12.2 sec
	10 hours	**9.3 sec**	18.2 sec
SMS	1 hour	**21.1 sec**	34.8 sec
	10 hours	15.7 hours	**2.1 min**
Higgs	1 hour	10.6 min	**10.7sec**
	10 hours	Crashed	3.6 min
Stackoverflow	1 day	**3.2 min**	3.7 min
	1 week	Crashed	**6.6 min**
Wiki-talk	10 hours	Crashed	**7.5 min**
	1 day	Crashed	**19 min**

Effect of Window Length(ω): Finding the best ω for a dataset is not straightforward, as it depends on the kind of interaction. For example, in the case of a social network where people respond quickly, an ω of 1 to 10 hours might do, but for slow networks such as stack overflow, where people generally respond after few days only, longer cycles of length 1 day or 1 week could still be interesting. In biological neural networks, ω could be in the range of milliseconds, as feedback loops take place almost instantaneously.

We also study the effect of increasing the window length on processing time and cycle count. We present the results for the SMS dataset in Figure 5. We make two observations;

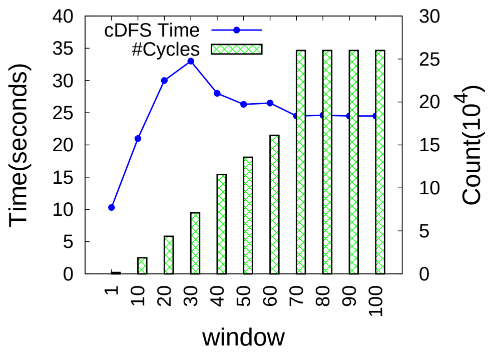


Figure 5: Effect of window length on processing time and cycle count for `SMS` data set

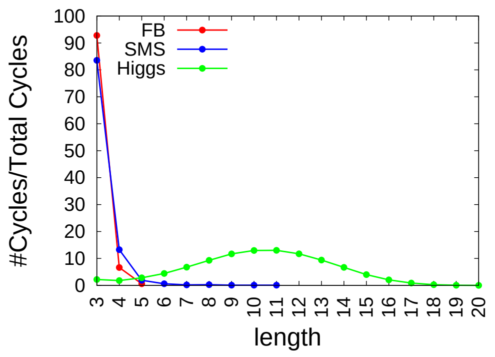


Figure 6: Distribution of simple cycle count and length for $\omega = 10$ hours.

first, as expected, the processing time and count of simple cycles increases with an increase in window length, but after a certain window length both become constant. This is because when the window is large enough, the temporal characteristic of the network do not change any more. In case of the `SMS` data set, this happens at a window length of 70 hours. Second, we see that the processing time increases at first and then decreases slightly again before becoming constant. This decrease in processing time is the result of the higher compression of candidate nodes for larger windows, resulting in fewer root candidates, but each with a higher number of cycles, found in one cDFS scan.

Therefore, in cases where the application does not provide a natural way to set ω, one could take the approach of starting with a small window at first and iteratively increase it until a sufficient number of interesting cycles is found. In our experiments we take ω as 1 hour and 10 hours for fast interaction network such as Twitter and Facebook and large ω for slower interaction networks.

7.3 Qualitative Evaluation

Cycle Frequency Distribution: In figure 6, we present the frequency distribution of the number simple cycles by cycle length for the `Facebook`, `SMS` and `Higgs` data sets for a window of 10 hours. The maximum cycle length is 5 and 11 respectively for the `Facebook` and `SMS` data set, and the number of triangles is very high as compared to the number of longer cycles. In the `Higgs` data set, however, the maximal cycle length is 20 and the cycle count distribution is very different. We think this could be because the `SMS` and `Facebook` data sets capture interactions between friends whereas `Higgs` is an open interaction platform where followers are interested in similar topic of discussion.

8. CONCLUSION

We addressed the problem of enumerating simple temporal cycles that do not exceed a given time window length ω in an interaction network. One of the applications we proposed and explored in the paper is using the number and length distribution of temporal cycles to characterize (part of) the dynamic behaviour of the temporal network. This is similar in spirit to using metrics such as clustering coefficient or diameter to characterize static networks. In order to visualize this distribution, it is necessary to enumerate, or at least count the number of cycles of all lengths. We presented an efficient algorithm, 2SCENT, which consists of two phases. In the first phase all sources of cycles are detected, which are then further expanded into the full cycles in the second phase. The base version of 2SCENT was extended in two important ways: first, we introduced the use of Bloom filters to reduce the memory consumption of the source detection phase by replacing the reverse reachability set by a reverse reachability filter. The second extension, using path bundles, handles the common case of repeated interactions leading to an explosion in the number of cycles. In experiments, we found that 2SCENT with its extensions runs up to 300 times faster than the only existing competitor. The experiments show that the algorithm could scale to millions of nodes and interactions using only commodity hardware. While the focus of this paper was more on algorithms and general aspects of temporal cycle enumeration, we also presented a qualitative analysis of cycles in temporal networks and analyzed the temporal nature of different real-world networks using the cycle count frequency distribution. For closed versus open friendship networks we could observe different cycle distributions, indicating different dynamic behaviours in these networks.

We consider two important avenues for future work. First, more research is required to definitely answer the question whether or not the temporal cycle distribution is a good way to represent dynamic behaviour in networks. Moreover, for the datasets used in this paper, we did not have access to the actual content of the interactions such as the tweets on the Twitter network. A qualitative study of the cycles found and their meaning and significance from an application perspective are of great interest. Secondly, it is also important to take into account the frequency of interaction between nodes when assessing the significance of the cycles found. Indeed, for nodes that are closely collaborating and interacting frequently, it is likely that accidental cycles may emerge. Therefore, methods need to be developed to measure the probability of temporal cycles emerging by chance.

Acknowledgment

This work was supported by the Fonds de la Recherche Scientifique-FNRS under Grant(s) no T.0183.14 PDR. The student is also part of IT4BI DC program.

9. REFERENCES

[1] E. Bergamini, H. Meyerhenke, and C. L. Staudt. Approximating betweenness centrality in large evolving networks. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 133–146. SIAM, 2014.

[2] E. Birmelé, R. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1884–1896. Society for Industrial and Applied Mathematics, 2013.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[4] C.-Y. Dong, D. Shin, S. Joo, Y. Nam, and K.-H. Cho. Identification of feedback loops in neural networks based on multi-step granger causality. *Bioinformatics*, 28(16):2146–2153, 2012.

[5] P.-L. Giscard, P. Rochet, and R. C. Wilson. Evaluating balance on social networks from their simple cycles. *Journal of Complex Networks*, page cnx005, 2017.

[6] F. Hoffmann and D. Krasle. Fraud detection using network analysis, 2015. EP Patent App. EP20,140,003,010.

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

[8] W. Hu, H. Zou, and Z. Gong. Temporal pagerank on social networks. In *International Conference on Web Information Systems Engineering*, pages 262–276. Springer, 2015.

[9] D. B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[10] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[11] R. Kumar and T. Calders. Finding simple temporal cycles in an interaction network. In *Proceedings of the Workshop on Large-Scale Time Dependent Graphs (TD-LSG 2017) co-located with the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD 2017), Skopje, Macedonia, September 18, 2017.*, pages 3–6, 2017.

[12] R. Kumar and T. Calders. Information propagation in interaction networks. In *EDBT*, pages 270–281, 2017.

[13] R. Kumar and T. Calders. 2SCENT: An Efficient Algorithm for Enumerating All Simple Temporal Cycles(Full version). `http://rohit13k.github.io/doc/2SCENT.pdf`, 2018. [Online; accessed 11-June-2018].

[14] R. Kumar, T. Calders, A. Gionis, and N. Tatti. Maintaining sliding-window neighborhood profiles in interaction networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 719–735. Springer, 2015.

[15] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[16] P. Mateti and N. Deo. On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, 5(1):90–99, 1976.

[17] R. K. Pan and J. Saramäki. Path lengths, correlations, and centrality in temporal networks. *Physical Review E*, 84(1):016105, 2011.

[18] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610. ACM, 2017.

[19] J. Ponstein. Self-avoiding paths and the adjacency matrix of a graph. *SIAM Journal on Applied Mathematics*, 14(3):600–609, 1966.

[20] V. B. Rao and V. Murti. Enumeration of all circuits of a graph. *Proceedings of the IEEE*, 57(4):700–701, 1969.

[21] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *Data Mining and Knowledge Discovery*, 30(2):438–475, 2016.

[22] P. Rozenshtein and A. Gionis. Temporal pagerank. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 674–689. Springer, 2016.

[23] P. Rozenshtein, N. Tatti, and A. Gionis. Discovering dynamic communities in interaction networks. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 678–693. Springer, 2014.

[24] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Temporal distance metrics for social network analysis. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 31–36. ACM, 2009.

[25] R. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[26] J. C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Communications of the ACM*, 13(12):722–726, 1970.

[27] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM workshop on Online social networks*, pages 37–42. ACM, 2009.

[28] J. T. Welch Jr. A mechanical analysis of the cyclic structure of undirected linear graphs. *Journal of the ACM (JACM)*, 13(2):205–210, 1966.

[29] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *PVLDB*, 7(9):721–732, 2014.

[30] Y. Wu, C. Zhou, J. Xiao, J. Kurths, and H. J. Schellnhuber. Evidence for a bimodal distribution in human communication. *Proceedings of the national academy of sciences*, 107(44):18803–18808, 2010.

[31] S. Yau. Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems. *IEEE Transactions on Circuit Theory*, 14(1):79–81, 1967.