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 where one person posts a message and after a while reacts to a thread of reactions from peers on the post. In financial networks, on the other hand, the presence of a temporal cycle could be indicative for certain types of fraud. 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*, (): xxxx-yyyy, 2018.
DOI: https://doi.org/TBD

*Just a note.

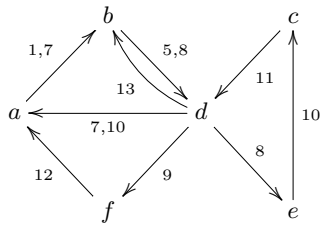
Proceedings of the VLDB Endowment, Vol. , No.

DOI: https://doi.org/TBD

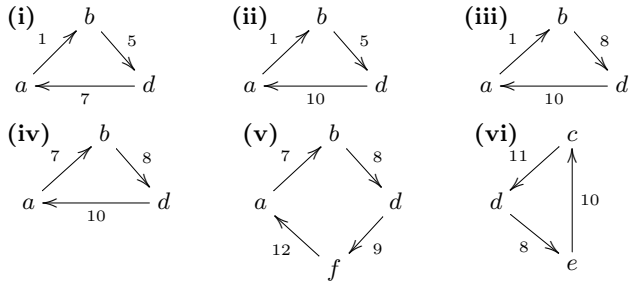
1. INTRODUCTION

Analyzing the temporal dynamics of a network is becoming very popular. In 2011, Pan et al. [16] 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 motivates 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. [17] 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 this work to temporal cycles of any length. Cycles appear naturally in many problem settings. For instance, in logistics the interactions may represent resources being moved between facilities, and a cycle could indicate an optimization opportunity by reducing excessive relocation of resources; in stock trading, cyclic patterns could indicate attempts to artificially create high trading volumes; 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.

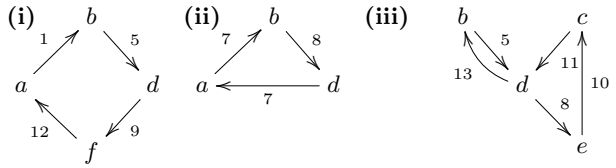
Figure 1b illustrates our notion of a temporal cycle in the temporal graph given in Figure 1a. To avoid spurious cycles stretched out over time we bound the window in which a cycle has to occur to $\omega = 10$. Figure 1c contains some examples of cycles in the static graph which are not considered as they either (i) extend over a too long time window, (i) the interactions do not respect temporal order, or (iii) the cycle is not *simple* in the sense that there are repeated vertices. For enumerating all simple temporal cycles, we first looked into the vast literature on enumerating cycles in static graphs from the early 70s [26, 9, 27]. The algorithms proposed in these works, however, are not directly applicable to temporal networks. The same holds for recent methods [4, 23, 25]; these approaches focus on a different model in which the dynamics are captured by considering a sequence of snapshots of the network. Therefore, in this paper, we propose a new efficient algorithm (2SCENT) for enumerating all simple temporal cycles of bounded times-



(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

pan. 2SCENT proceeds in two phases. 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, finding root nodes of cycles can be done 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 with a factor exponential in the size of the paths.

We ran extensive experiments with our new algorithm. The experiments show consistent performance improvements by the extensions and an improvement of two orders of magnitude over the algorithm of *Kumar and Calders* [11]. We used 6 real world data sets in the experiments. We also present a qualitative analysis concerning the distribution of frequency and size of simple 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 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 where only friends interact. Cycle detection is able to quantify these differences.

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 [19, 15, 29, 18, 32, 26, 9, 27]. 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 to enumerate all simple cycles.

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 so-called *path bundles*.

Patterns in temporal graphs. Temporal graphs, also know as interaction networks [13, 22] 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, 21], shortest path [16, 24, 30], or centrality measures [1, 20] 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, 17] 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. [17] 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. [17] 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. The key idea behind the algorithm is to maintain an indexed list of all *valid temporal paths*. A temporal path is considered valid at time t if the first interaction in the temporal path is within $t - \omega$ duration where ω is the time window. When an interaction (u, v, t) is processed, all temporal paths with last node u are extended to create a new path if v is not already present in the path. This algorithm, however, does not scale well for large graphs. In the empirical evaluation we present in the experimental section, 2SCENT outperforms the algorithm of [11] by a factor of 300 in terms of time needed to enumerate all cycles.

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. We will use $n = |V|$ to denote the number of nodes in the temporal graph, and $m = |\mathcal{E}|$ to denote the total 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 use the more compact notation $u \xrightarrow{t_1} n_1 \xrightarrow{t_2} n_2 \ldots \xrightarrow{t_k} v$ to represent a temporal path from u to 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 $dur(p) \leq \omega$. The *start time* $t_s(p) := t_1$ and *end time* $t_e(p) := t_k$ of path p are given by the time stamps of the first and last interactions in the path.

For example, in the temporal graph shown in Figure 1a, the path $b \xrightarrow{5} d \xrightarrow{8} e \xrightarrow{10} c \xrightarrow{11} d$ 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 $11 - 5 = 6$. On the other hand, $b \xrightarrow{5} d \xrightarrow{8} e \xrightarrow{10} c$ 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 $dur(C) \leq \omega$.

For the temporal graph given in Figure 1a, the solution of the SCE problem for $\omega = 10$ is given by the cycles in Figure 1b plus the cycles $b \xrightarrow{5} d \xrightarrow{13} b$ 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

Algorithm 1 GenerateSeeds

Require: Threshold ω, 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(ω, E)
2:     for (a, b, t) ∈ E, ordered ascending w.r.t. t do
3:         if S(b) does not exist then
4:             S(b) ← {}
5:         S(b) ← S(b) ∪ {(a, t)}
6:         if S(a) exists then
7:             S(a) ← S(a) \ {(x, t_x) ∈ S(a) | t_x ≤ t − ω}
8:             S(b) ← S(b) ∪ S(a)
9:             for (b, t_b) ∈ S(b) do
10:                C ← {c | (c, t_c) ∈ S(a), t_c > t_b} ∪ {b}
11:                Output (b, [t_b, t], C)
12:                S(b) ← S(b) \ {(b, t_b)}
13:        if time to prune then
14:            for all summaries S(x) do
15:                S(x) ← S(x) \ {(y, t_y) ∈ S(x) | t_y ≤ t − ω}
```

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 $\{x \mid \exists (x, t_x) \in S(a) \mid t_b < t_x < t\}$.

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 $\mathcal{O}(\min(n, W)W)$ assuming the pruning is done every $\mathcal{O}(\omega)$ steps.*

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 $b \in S(a)$.

Algorithm 2 GenerateSeedsBloom

Require: Threshold ω, 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(ω, $\mathcal{E}$)
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,ω)
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,ω)
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,ω)
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** $h_1(b)$, ..., $h_k(b)$ 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*

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 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. For instance 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 11, 15, and 21. 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 6-9 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 *(i)s $\mathcal{O}(q(m + c'))$ 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 $(a, [1, 10], \{b, d, e, f\})$ and a cDFS run on this 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 ω
Ensure: Combined candidates

function CombineSeeds($\mathcal{C}$,ω)
  Sort $\mathcal{C}$ on t_s ascending, then t_e descending.
  while $\mathcal{C}$ not empty **do**
    Let $(s, [t_s, t_e], C)$ be first in $\mathcal{C}$
    Let *Compatible* be the maximal prefix of $\mathcal{C}$ such that for all $(s, [t'_s, t'_e], C') \in$ *Compatible* it holds that $t'_e < t_s + \omega$
    $\mathcal{C} \leftarrow \mathcal{C} \setminus$ *Compatible*
    if $\mathcal{C}$ is empty **then** $t_n \leftarrow t_s + \omega$
    else
      Let $(s, [t'_s, t'_e], C')$ be first in $\mathcal{C}$
      $t_n \leftarrow t'_s$
    $t_{max} \leftarrow \max\{t'_e \mid (s, [t'_s, t'_e], C') \in$ *Compatible*$\}$
    $C_{all} \leftarrow \bigcup\{C' \mid (s, [t'_s, t'_e], C') \in$ *Compatible*$\}$
    Output $(s, [t_s, t_{max}], t_n, C_{all})$

Theorem 3. *Algorithm 3 ensures that for every temporal cycle rooted at s and starting and ending at times t_s and t_e respectively, 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 DEPTH-FIRST SEARCH

After finding candidates, we want to find the exact cycles for all candidates. For each extended candidate $(s, [t_s, t_e], t_n, C)$ we will 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 over and over again. Consider for instance the example in Figure 2. 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 7, so its closing time will become 7 and any depth-first exploration of c will be aborted from timestamp 7 on.

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.

function Unblock(Node v, time stamp t_v)
  if $t_v > ct(v)$ **then**
    $ct(v) \leftarrow t_v$
    for $(w, t_w) \in U(v)$ **do**
      if $t_w < t_v$ **then**
        $U(v) \leftarrow U(v) \setminus \{(w, t_w)\}$
        $T[w, v] = \{t \mid (w, v, t) \in \mathcal{E}\}$
        $T \leftarrow \{t \in T[w, v] \mid t_v \leq t\}$
        if $T \neq \emptyset$ **then**
          $U(v) \leftarrow U(v) \cup \{(w, \min(T))\}$
        $t_{max} \leftarrow \max\{t \in T[w, v] \mid t < t_v\}$
        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.

function Extend($U(v)$,(w, t))
  if there is an entry $(w, t') \in U(v)$ **then**
    if $t' > t$ **then** $U(v) \leftarrow U(v) \setminus \{(w, t')\} \cup \{(w, t)\}$
  else
    $U(v) \leftarrow U(v) \cup \{(w, t)\}$

Algorithm 6 Algorithm AllPaths

Require: Prefix path $s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_k} v_k$ 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 -t1-> v1 ... -tk-> vk)
2:     v_cur ← v_k, t_cur ← t_k
3:     ct(v_cur) ← t_cur, lastp ← 0
4:     Out ← {(v_cur, x, t) ∈ E | t_cur < t}
5:     N ← {x ∈ V | (v_cur, x, t) ∈ Out}
6:     if s ∈ N then
7:         for (v_cur, s, t) ∈ Out do
8:             if t > lastp then
9:                 lastp ← t
10:            Output pr · ⟨(v_cur, s, t)⟩
11:    for x ∈ N \ {s} do
12:        T_x ← {t | (v_cur, x, t) ∈ Out}
13:        while T_x ≠ ∅ do
14:            t_m ← min(T_x)
15:            pass ← False
16:            if ct(x) ≤ t_m then pass ← False
17:            else pass ← AllPaths(pr · ⟨(v_cur, x, t_m)⟩)
18:            if not pass then
19:                T_x ← ∅
20:                Extend(U(x),(v_cur, t_m))
21:            else
22:                T_x ← T_x \ {t_m}
23:                if t_m > lastp then
24:                    lastp ← t_m
25:    if lastp > 0 then Unblock(v_cur,lastp)
26:    return (lastp ≠ 0)
```

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 and this nodes cannot be used to extend the path without violating the simplicity condition;
- $b \xrightarrow{5} c$: $ct(c)$ becomes 5;
- 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. Similarly, during the recursive calls, the closing times of the other nodes have been updated as well.

In order to find additional paths, we backtrack and find the next solution. Suppose now that we already explored the

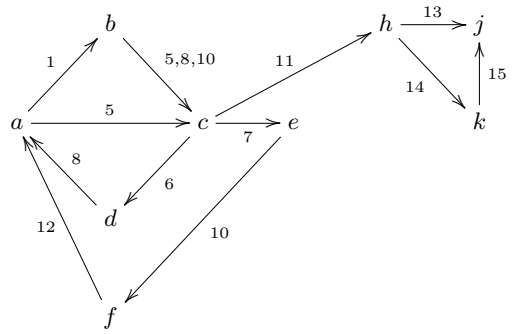


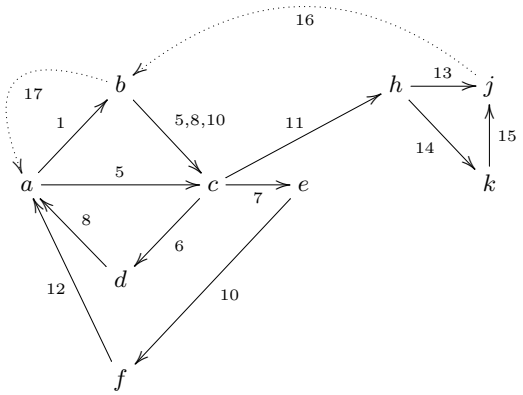
Figure 2: Example temporal network with simple cycles

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. The new interactions are marked by 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 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

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 starting in interval $[t_s, t_n[$ and ending before t_e, using only vertices of C.

```
1: function CYCLE(s)
2:     E ← {(u,v,t) ∈ E | u,v ∈ C, t ∈ [t_s, t_e]}      ▷ Reduce the interaction graph
3:     V ← C
4:     for x ∈ C do
5:         ct(x) ← ∞, U(x) ← ∅
6:     for (s,x,t) ∈ E | t < t_n do
7:         AllPaths(s -t-> x)
```

start with $a \xrightarrow{1} c$. 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 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.*

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 $\mathcal{O}((c + 1)(m + n))$.*

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 3. 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.

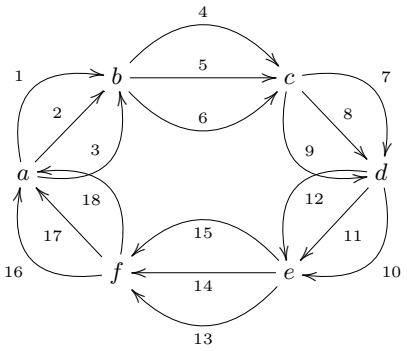


Figure 3: Example temporal network with simple cycles having multiple repeated edges

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:

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

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 $\mathcal{P}(B) = \mathcal{P}(B')$*

For the above example, all cycles could be represented by a single path bundle: $a \xrightarrow{1,2,3} b \xrightarrow{4,5,6} c \xrightarrow{7,8,9} d \xrightarrow{10,11,12} e \xrightarrow{13,14,15} d \xrightarrow{16,17,18} a$.

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

Algorithm 8 Extending a path bundle with an edge bundle

Require: Minimal path bundle $B = v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$, 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($v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$,$v_{k+1} \xrightarrow{T_{k+1}} v_{k+2}$)
2: $T'_{k+1} \leftarrow \{t \in T_{k+1} \mid t > \min(T_k)\}$
3: **if** $T'_{k+1} = \emptyset$ **then**
4: **return** $(v_1 \xrightarrow{\emptyset} \dots v_i \xrightarrow{\emptyset} \dots \xrightarrow{\emptyset} v_{k+2})$
5: **for** $i = k$ down to 1 **do**
6: $T'_i = \{t \in T_i \mid t < \max(T'_{i+1})\}$
7: **return** $(v_1 \xrightarrow{T'_1} \dots v_i \xrightarrow{T'_i} \dots \xrightarrow{T'_{k+1}} v_{k+2})$

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: $a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{4,7} d$.

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, Algorithm 10 comes to the rescue. In Algorithm 10, we compute the number of paths in a bundle $v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$ by iteratively considering all the prefixes of the bundle in increasing length. For each prefix $P_i = v_1 \xrightarrow{T_1} \dots \xrightarrow{T_i} v_{i+1}$, 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. We illustrate the algorithm with an example.

Consider the path bundle $a \xrightarrow{1,3,7} b \xrightarrow{4,8,12} c \xrightarrow{7,13} d$.

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**
10: $lastp \leftarrow t$
11: **Output** $Expand(B, v_{cur} \xrightarrow{T} s)$
12: **for** $x \in N \setminus \{s\}$ **do**
13: $T_x \leftarrow \{t \mid (v_{cur}, x, t) \in Out\}$
14: $T'_x \leftarrow \{t \in T_x \mid t < ct(x)\}$
15: **if** $T'_x \neq \emptyset$ **then**
16: $last_x \leftarrow AllBundles(s, Expand(B, v_{cur} \xrightarrow{T'_x} x))$
17: **if** $last_x > lastp$ **then**
18: $lastp \leftarrow last_x$
19: $t_m \leftarrow \min\{t \in T_x \mid t > last_x\}$
20: EXTEND($U(x)$,(v_{cur}, t_m))
21: **if** $lastp > 0$ **then**
22: UNBLOCK(v_{cur},$lastp$)
23: **return** $lastp$

- The heap for the length 0 prefix P_0 contains just one pair $(0, 1)$, indicating that there is one path of length 0 that ends at timestamp 0.

- For $P_1 = a \xrightarrow{1,3,7} b$, we go over the timestamps from small to large and for each of the timestamps t we look how many paths in P_0 it can extend. This number is computed by popping off elements from the heap until the head of the heap contains a timestamp larger than t. The last pair we popped off contains the number n we need. In order to compute the total number of paths ending at t or an earlier timestamp, we add to n the number of paths we already gathered. So, for P_1, the content of the variables and the pair pushed on the heap H_1 evolve as follows:

t	H_0	n	$prev$	pushed in H_1
1	$\langle(0,1)\rangle$	1	1	(1,1)
3	$\langle\rangle$	1	2	(3,2)
7	$\langle\rangle$	1	3	(7,3)

In the end H_1 will be: $\langle(1,1),(3,2),(7,3)\rangle$. Recall that $(7,3)$ on H_1 means that there are 3 paths in $\mathcal{P}(P_1)$ that end at timestamp 7 or earlier. Indeed, these three paths are: $a \xrightarrow{1} b$, $a \xrightarrow{3} b$, and $a \xrightarrow{7} b$.

- Now we proceed to $P_2 = a \overset{1,3,7}{\longrightarrow} b \overset{4,8,12}{\longrightarrow} c$. We will compute H_2 by combining T_2 with H_1. Again we iterate from small to large over T_2. For $t = 4$ we need to compute how many of the paths in $\mathcal{P}(P_1)$ it can complete. For this purpose, as long as $t' < t$, we pop off the pairs (t', n) from H_1. The last pair (t', n') we pop off contains the number of paths with which we can combine. This is $(3, 2)$, hence n becomes 2. We push $(4, 2)$ on the heap H_2. So, for P_2, the content of the variables and the pair pushed on the heap H_2 evolve as follows:

t	H_1	n	$prev$	pushed in H_2
4	$\langle(1,1),(3,2),(7,3)\rangle$	2	2	(4,2)
8	$\langle(7,3)\rangle$	3	5	(8,5)
12	$\langle\rangle$	3	8	(12,8)

In the end H_2 will be: $\langle(4, 2), (8, 5), (12, 8)\rangle$.

- We continue the same procedure for P_3 and iteratively get the following evolution:

t	H_2	n	$prev$	pushed in H_3
7	$\langle(4,2),(8,5),(12,8)\rangle$	2	2	(7,2)
13	$\langle(8,5),(12,8)\rangle$	8	10	(13,10)

Hence, H_3 ends up to be : $\langle(7, 2), (13, 10)\rangle$. The final answer is in the tail of H_3 and is 10.

Algorithm 10 Counting the number of paths in a bundle

Require: Path bundle $B = v_1 \overset{T_1}{\to} \dots v_i \overset{T_i}{\to} \dots \overset{T_k}{\to} v_{k+1}$
Ensure: The cardinality of $\mathcal{P}(B)$
Let H_0 be an empty heap
Push $(0, 1)$ on H_0
for i=1...k **do**
  Let H_i be an empty heap
  $n \leftarrow 0$
  $prev \leftarrow 0$
  for $t \in T_i$ sorted ascending **do**
    if H_{i-1} is not empty **then**
      $(t', n') \leftarrow head(H_{i-1})$
      while $t' < t$ **do**
        Pop (t', n') from H_{i-1}
        $n \leftarrow n'$
        $(t', n') \leftarrow head(H_{i-1})$
    Push $(t, prev + n)$ on H_i
    $prev \leftarrow prev + n$
Let (t, n) be the tail of H_k
return n

The reason that we went for the complication of having a heap is because it allows us to compute H_i in time proportional to $|H_{i-1}| + |T_i|$. Since $|H_i| = |T_i|$, we get as total time complexity $\mathcal{O}(\sum_{i=1}^{k} |T_i|)$. This is much more efficient than iterating over all paths which in worst case takes time $\prod_{i=1}^{n} |T_i|$. This complexity occurs when for all $i = 1 \dots k-1$, $\max(T_i) < \min(T_{i+1})$.

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 CPU and 16 GB of RAM, running the Linux operating system. The code and instructions to run the experiments are available online (`https://github.com/rohit13k/CycleDetection`).

7.1 Dataset

All datasets except `SMS` [31], `Facebook` [28] and `USElection` [12] were obtained from the SNAP repository [14]. 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 higher window of 1 day and 1 week were considered.

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

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 approach 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.

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

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

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

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

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).

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

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**

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

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 which will contain a cycle. For some datasets such as Higgs, Stackoverflow, and Wiki-talk, for higher window length, the Naive algorithm crashes due to the high number of temporal paths it is maintaining in memory. The results are presented at Table 5.

Effect of Window Length: 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 4. We make two observations; first, as expected, the processing

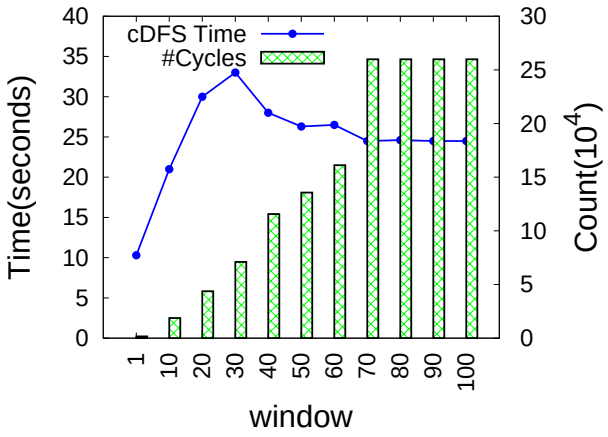


Figure 4: Effect of window length on processing time and cycle count for SMS data set

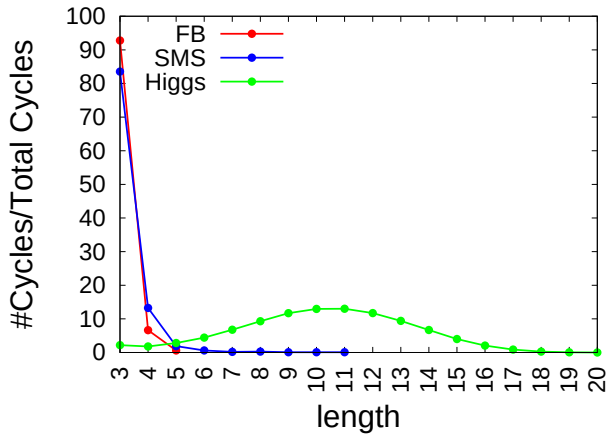


Figure 5: Distribution of simple cycle count and length for $\omega = 10$ hours.

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.

7.3 Qualitative Evaluation

Cycle Frequency Distribution: In figure 5, 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 with interactions among unknown followers 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. Related to this is the evaluation of the usefulness of the cycles in applications such as fraud detection. 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 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. Only in this way we can properly assess the significance of the cycles found.

Acknowledgment

This work was supported by the Fonds de la Recherche Scientifique-FNRS under Grant(s) no T.0183.14 PDR.

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] D. M. Dunlavy, T. G. Kolda, and E. Acar. Temporal link prediction using matrix and tensor factorizations. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2):10, 2011.

[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, 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.

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

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

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

[17] 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.

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

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

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

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

[22] 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.

[23] K. Semertzidis and E. Pitoura. Historical traversals in native graph databases. In *Advances in Databases and Information Systems*, pages 167–181. Springer, 2017.

[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] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe. A framework for community identification in dynamic social networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 717–726. ACM, 2007.

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

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

[28] 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.

[29] 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.

[30] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[31] 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.

[32] 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.

APPENDIX

A. PROOF OF CORRECTNESS FOR SOURCE DETECTION PHASE

THEOREM 6. *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.*

PROOF. By induction on the prefixes of sequence of interactions we can show that at time t, $S(x)$ contains at least all pairs (y, t_s) such that (i) $t - t_s < \omega$ and (ii) there exists a temporal path from y to x that starts with an interaction at time t_s. Furthermore, (iii) if $(y, t_s) \in S(x)$ then (ii) holds (but not necessarily (i)). If there is a valid temporal cycle $a \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} a$, then there is such a temporal path $a \xrightarrow{t_1} v_1 \xrightarrow{t_2} \ldots \xrightarrow{t_{n-1}} v_{n-1}$ (a, t_1) at time t_n and hence $(a, t_1) \in S(v_{n-1})$ when interaction $v_{n-1} \xrightarrow{t_n} a$ is processed. Therefore, the cycle is detected and reported, and because of (iii), the set C will contain at least all $\{a, v_1, \ldots, v_n\}$. Because of (ii) only cycles are reported and it is easy to see that due to line 7, all reported cycles are valid. □

THEOREM 7. *Algorithm 3 ensures that for every temporal cycle rooted at s and starting and ending at times t_s and 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 $t_e \in [t'_s, t'_e]$.*

PROOF. GENERATESEEDS generates a seed for each cycle. In the definition of *Compatible* it is guaranteed that all elements that are removed from $\mathcal{C}$ are contained in the extended cycle that is output. Furthermore, it is easy to see that the intervals $[t_s, t_n[$ for all generated extended seeds are disjoint, as t_n is the starting point t'_s of the first seed that is not contained in *Combined*. □

B. PROOF OF CORRECTNESS FOR CONSTRAINED DEPTH-FIRST SEARCH

CYCLE(s) calls ALLPATHS($s \xrightarrow{t} x$) multiple times, once for each interaction $s \xrightarrow{t} x$. We will show that CYCLE(s) generates all cycles with root s. During the execution of ALLPATHS(pr), ALLPATHS($pr \xrightarrow{t_m} x$) is recursively called. Notice that in ALLPATHS(pr) there is no order specified in which the neighbors of v_{cur} are considered in the for-loop that starts at line 7. This arbitrary order of the for-loop is, however, not a problem for the correctness of the algorithm.

It is easy to see that regardless of the order in which the for-loop runs through the interactions, the call tree of CYCLE(s) that contains the different calls ALLPATHS(pr) satisfies the following conditions:

LEMMA 3. *Consider the call tree formed by the call* CYCLE(s) *and containing all (direct and recursive) calls* ALLPATHS(pr). *Let the root node that corresponds to the call to* CYCLE(s) *be labeled with the empty path, and let the other nodes of the call tree be labeled with the argument passed for parameter pr in the call* ALLPATHS(pr). *This tree satisfies the following properties:*

1. *Every non-root node has a valid simple temporal path rooted at s as label;*
2. *For every non-root node it holds that its label is the label of its parent extended with one interaction;*
3. *There are no two nodes with the same label.*

PROOF. ALLPATHS(.) explores paths in depth-first order. Every path $s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_{k-1}} v_{k-1}$ is generated only in ALLPATHS($s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_k} v_k$), and no edge is considered twice for extending the prefix as N and Out are sets. All paths are temporal because in a call ALLPATHS($s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_k} v_k$) only interactions $v_k \xrightarrow{t} x$ are considered with $t > t_k$. All paths are simple, because $ct(v_k)$ is set to t_k before all recursive calls, and hence the if-condition on line 16 will fail for all nodes x that are already in the path pr. □

This implies that we can refer to a specific call ALLPATHS(pr) in a run of CYCLE(s) with the unique value of the argument for parameter pr.

B.1 Soundness

LEMMA 4. *If a cycle $s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_k} v_k \xrightarrow{t} s$ is output, then it is output in* ALLPATHS($s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_k} v_k$) *and $v_k \xrightarrow{t} s \in \mathcal{E}$.*

As a direct result of Lemma 3 and Lemma 4, we get the following corollary.

COROLLARY 1. *Every cycle generated by AllPaths is a simple temporal cycle, and no cycle is generated more than once.*

B.2 Completeness

For notational convenience, we start by introducing some new notations.

Definition 5. An interaction $x \xrightarrow{t} y$ is called *blocked* if $ct(y) \leq t$. An interaction that is not blocked is called *free*.

Let $p = s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_k} v_k$ be a temporal path; $V(p)$ denotes the set of vertices on the path p. Let $v_k \xrightarrow{t} x \in \mathcal{E}$ with $t > t_k$. $p \xrightarrow{t} x$ denotes the temporal path $s \xrightarrow{t_1} v_1 \ldots \xrightarrow{t_k} v_k \xrightarrow{t} x$.

We will use $U(y) \preceq (x, t)$ to denote that there exists a pair $(x, t') \in U(y)$ with $t' \leq t$.

A key property in the proof of completeness of our algorithm will be that an interaction becomes unblocked if and only if a path starting from that interaction to the root s becomes available using only available nodes; i.e., nodes that are not in pr. This happens only when we return from a call ALLPATHS(pr) and has to be implemented in UNBLOCK(v, t_v) on line 27. *consistency* will be the notion expressing that a call to UNBLOCK(v, t_v) is well-behaved. The paths that become available are those using no nodes from pr, except the root node s, and the node v_{cur} that will become available again when we return from ALLPATHS(pr).

Definition 6. **Consistency of Unblock**
We say that the call UNBLOCK($v_{cur}, last$) in ALLPATHS(pr) is *consistent* if the following holds: an interaction $x \xrightarrow{t} y$ that is blocked just before the call changes status from blocked to free if and only if there exists a temporal path $p_{x \to s}$ from x to s that starts with $x \xrightarrow{t} y$ and with $V(pr) \cap V(p_{x \to s}) \subseteq \{s, v_{cur}\}$.

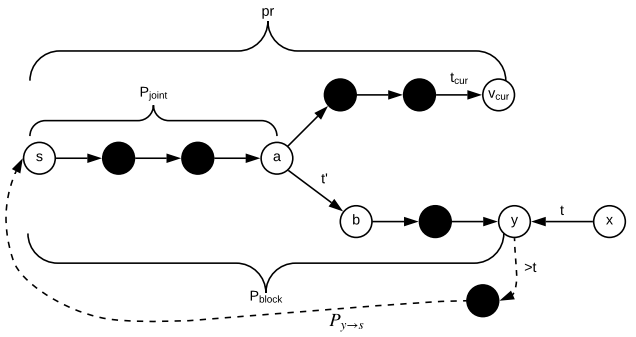


Figure 6: Illustration for proof of Lemma 5.

B.2.1 Consistency Implies Completeness

LEMMA 5. *Let $G(V, \mathcal{E})$ be an interaction graph and consider a run of cs. As long as all finished calls to* UNBLOCK *are consistent, the following holds: if at the start of* ALLPATHS(pr), *interaction $x \xrightarrow{t} y$ is blocked, then there doesn't exist a temporal path from x to s that starts with $x \xrightarrow{t} y$ and intersects pr only at s.*

PROOF. We prove by contradiction. Check the illustration given in Figure 6 for more clarity. Suppose there is a ALLPATHS(pr) in which at the start $x \xrightarrow{t} y$ is blocked while at the same time there exists a path $p_{x\to s}$ from x to s that starts with $x \xrightarrow{t} y$ and intersects pr only in s, and all UNBLOCK operations up to that point were consistent. Then, there must have been a previous call ALLPATHS(p_{block}) in which $x \xrightarrow{t} y$ got blocked, which means that $ct(y)$ decreased from above t to lower than or equal to t. $ct(y)$ only gets lower inside a call in which $v_{cur} = y$, hence p_{block} is a path that ends in y. Since $y \in V(p_{x\to s})$ and $V(p_{x\to s}) \cap V(pr) = \{s\}$, y cannot be in pr, and hence p_{block} is not a prefix of pr. Since we are at the start of ALLPATHS(pr), neither is pr a prefix of p_{block}. Let now p_{joint} be the longest common prefix of pr and p_{block}, and consider the first interaction $a \xrightarrow{t'} b$ in p_{block} after p_{joint}. Since ALLPATHS(p_{block}) was executed before ALLPATHS(pr), and $p_{joint} \cdot a \xrightarrow{t'} b$ is a prefix of p_{block}, ALLPATHS($p_{joint} \cdot a \xrightarrow{t'} b$) finishes after ALLPATHS(p_{block}) does, and before ALLPATHS(pr) starts. So, UNBLOCK in ALLPATHS($p_{joint} \cdot a \xrightarrow{t'} b$) is consistent according to our assumptions at the start of the proof. This implies that after this UNBLOCK, $x \xrightarrow{t} y$ must be free because of the path $p_{x\to s}$ that intersects $p_{joint} \cdot a \xrightarrow{t'} b$ in at most b and s. We have reached a contradiction, because we considered an arbitrary call ALLPATHS(p_{block}) that blocks $x \xrightarrow{t} y$ and precedes ALLPATHS(pr), and have proven that it $x \xrightarrow{t} y$ will be freed in between ALLPATHS(p_block) and ALLPATHS(pr). As such, $x \xrightarrow{t} y$ cannot be blocked at the start of ALLPATHS(pr), a contradiction with our assumptions at the start of the proof. □

LEMMA 6. *Let $G(V, \mathcal{E})$ be an interaction graph. If in an run of* CYCLE(s) *all calls* UNBLOCK *are consistent, then* CYCLE(s) *generates all simple cycles rooted at s.*

PROOF. We will prove the lemma by contradiction. Suppose cycle $s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k \xrightarrow{t} s$ is not output. This means that ALLPATHS($s \xrightarrow{t_1} v_1 \dots \xrightarrow{t_k} v_k$) is never executed. Let p_j denote $s \xrightarrow{t_1} v1 \dots \xrightarrow{t_j} v_j$, and let i be the smallest index such that ALLPATHS(p_i) is not executed. This means that before we reach line 19 in ALLPATHS(p_{i-1}) for $x = v_i$ and $t_m = t_i$, the interaction $v_{i-1} \xrightarrow{t_i} v_i$ is blocked. Because of Lemma 5, $v_{i-1} \xrightarrow{t_{i-1}} v_i$ was not blocked at the start of ALLPATHS(p_{i-1}). On the other hand, neither can any recursive call inside ALLPATHS(p_{i-1}) return with $v_{i-1} \xrightarrow{t_{i-1}} v_i$ blocked, because such a recursive call ends with a consistent UNBLOCK which must end with a free $v_{i-1} \xrightarrow{t_{i-1}} v_i$ because of the temporal path $v_i \xrightarrow{t_i} v_{i+1} \dots \xrightarrow{t_k} v_k \xrightarrow{t} s$. Therefore, $v_{i-1} \xrightarrow{t_{i-1}} v_i$ will be considered at some point and ALLPATHS(p_i) is executed. This is in contradiction with our initial assumption and hence proves the lemma. □

B.2.2 Helper Lemmas: Relation Unblock Sets and Blocked Interactions

LEMMA 7. *Let $G(V, \mathcal{E})$ be an interaction graph and consider a run of* CYCLE(s). *Just before and after the call* UNBLOCK *in any* ALLPATHS(pr) *that is executed, the following holds: if $U(y) \preceq (x, t)$, then $x \xrightarrow{t} y$ is blocked.*

PROOF. Consider the call to Unblock in ALLPATHS(pr). (v_{cur}, t_m) is added to $U(x)$ in line 22 of the algorithm only if ALLPATHS($pr \xrightarrow{t_m} x$) fails. Hence, only if at the end of ALLPATHS($pr \xrightarrow{t_m} x$), $lastp = 0$. As a result, $ct(x)$ will be t_m right after the call which coincides with the moment that (v_{cur}, t_m) is added to $U(x)$. The only other place where pairs are added to unblock lists is in line 10 of unblock, and here it is easy to verify that if (w, t_{min}) is added to $U(v)$, then $ct(v) \leq t_{min}$. Hence we have already proven that whenever a pair (x, t) is added to an unblock list $U(y)$, $x \xrightarrow{t} y$ is blocked. We still need to show that whenever an edge $x \xrightarrow{t} y$ becomes unblocked, any (x, t') with $t' \leq t$ gets removed from $U(y)$. This is straightforward, as the only place where interactions become free is in line 3 of UNBLOCK, where $ct(v)$ is raised to t_v. Now any pair $(w, t_w) \in U(v)$ with $t_w < t_v$ needs to be removed from $U(v)$ to maintain the lemma, what happens right after in steps 4-6 of UNBLOCK. So, only during the execution of Unblock, the lemma may be temporarily broken, but just before and just after it holds. □

LEMMA 8. *Let $G(V, \mathcal{E})$ be an interaction graph and consider a run of* CYCLE(s). *Just before any call to* UNBLOCK, *the following holds: if $x \xrightarrow{t} y$ is blocked, then either $y \in V(pr)$ or for each interaction $y \xrightarrow{t'} z$ with $t' > t$ it holds that $(y, t') \preceq U(z)$ and $y \xrightarrow{t'} z$ is blocked as well.*

PROOF. Consider ALLPATHS(pr) for which $x \xrightarrow{t} y$ is blocked at the start of UNBLOCK, and $y \notin V(pr)$. Furthermore, assume that for every preceding call to Unblock the lemma held.Let ALLPATHS(p_y) be the last preceding call that lowered $cl(y)$ to t or below. Hence, p_y ends with an interaction $a \xrightarrow{t_y} y$ with $t_y \leq t$, and during ALLPATHS(p_y), $lastp$ never exceeds t. This implies that for all edges $y \xrightarrow{t'} z$ with $t' > t$ there exists an interaction $y \xrightarrow{t''} z$ with $t'' \leq t'$ such that ALLPATHS($p_y \xrightarrow{t''} z$) returns unsuccessfully. As a result $ct(z)$ becomes $t'' \leq t'$, and thus $y \xrightarrow{t'} z$ blocked. At the same time,

because AllPaths($p_y \xrightarrow{t''} z$) returns unsuccessfully, (y, t'') is added to $U(z)$ and hence $U(z) \preceq (y, t')$. As *lastp* in *call*(p_y) remains less than or equal to t, the call to Unblock at the end of AllPaths($p_y \xrightarrow{t''} z$) does not unblock any of the edges $y \xrightarrow{t'} z$ with $t' > t$.

Suppose now that before the start of Unblock in AllPaths(pr) one of the edges $y \xrightarrow{t'} z$ with $t' > t$ becomes unblocked or $U(z) \preceq (y, t')$ becomes false. It is easy to see that $y \xrightarrow{t'} z$ becomes unblocked implies $U(z) \preceq (y, t')$ becomes false and vice versa, as both occur in a call Unblock(z, t'') with $t'' > t'$. This on its turn implies that $(y, t') \in U(z)$ will trigger Unblock(y, t') and $ct(y)$ will be raised to at least $t' > t$, which is in contradiction with the fact that AllPaths(p_y) was the last preceding call that lowered $cl(y)$ to t or below. Hence the lemma still obtains at the moment we reach Unblock in AllPaths(pr). □

B.3 Main Result

Lemma 9. *Let $G(V, \mathcal{E})$ be an arbitrary interaction graph and consider a run of* Cycle(s). *Every execution of* Unblock *during that run is consistent.*

Proof. We prove the lemma by contradiction. Suppose there is at least one call to Unblock that is not consistent. Let the first call in the run that is not consistent be the Unblock operation in AllPaths(p_{fail}). The lemma can fail for two reasons: either an interaction that needs to be unblocked isn't, or one that shouldn't, is. We show that both cases lead to a contradiction.

Case 1: *An interaction $x \xrightarrow{t} y$ that is blocked just before the call changes status from blocked to free but there does not exist a simple temporal path p from x to s that starts with $x \xrightarrow{t} y$ and with $V(pr) \cap V(p) \subseteq \{s, v_{cur}\}$.* As $x \xrightarrow{t} y$ gets unblocked, there must be a sequence $y = y_1, y_2, \ldots, y_n = v_{cur}$ such that right before the call to Unblock in AllPaths(p_{fail}) starts, $(y_i, t_i) \in U(y_{i+1})$ for $i = 1 \ldots n-1$, with $ct(y_i) < t_i$, and $t_1 < t_2 < \ldots < t_{n-1} < lastp$. Furthermore, as $lastp > 0$, there is a temporal path $p_{v_{cur} \to s}$ from v_{cur} to s that does not intersect pr except in v_{cur} itself and s.

We first show that for $i = 1, \ldots, n-1$, $y_i \notin V(p_{fail})$. Suppose for the sake of contradiction there is at least one $y_i \in V(p_{fail})$. We can assume without loss of generality that y_i is the first such node on the path p_{fail}. Let p_i be the prefix of p_{fail} that ends in y_i. According to Lemma 5, $y_i \xrightarrow{t_i} y_{i+1}$ cannot be blocked at the start of AllPaths(p_i) because of the temporal path $y_{i+1} \xrightarrow{t_{i+1}} y_{i+2} \ldots \xrightarrow{t_n} y_n \cdot p_{v_{cur} \to s}$ that does not intersect p_i except in s. Hence, because of Lemma 7, $U(y_{i+1}) \not\preceq (y_i, t_i)$ at the start of AllPaths(p_i). Furthermore, (y_i, t_i) cannot have been added into $U(y_{i+1})$ in AllPaths(p_i) because of the existence of the temporal path $y_i \xrightarrow{t_i} y_{i+1} \ldots \xrightarrow{t_n} y_n \cdot p_{v_{cur} \to s}$. This implies that $(y_i, t_i) \notin U(y_{i+1})$; a contradiction.

Hence, $y_1 \xrightarrow{t_1} y_2 \ldots \xrightarrow{t_n} y_n \cdot p_{v_{cur} \to s}$ is a path from y_1 to s that intersects p_{fail} only in s and v_{cur}. This contradicts our assumption at the start of case 1 that no such path exists.

Case 2: *Interaction $x \xrightarrow{t} y$ remains blocked throughout the call to Unblock even though there exists a simple temporal path $p_{x \to s}$ from x to s that starts with $x \xrightarrow{t} y$ and with $V(pr) \cap V(p) \subseteq \{s, v_{cur}\}$.* Note that this implies that $y \notin V(p_{fail})$.

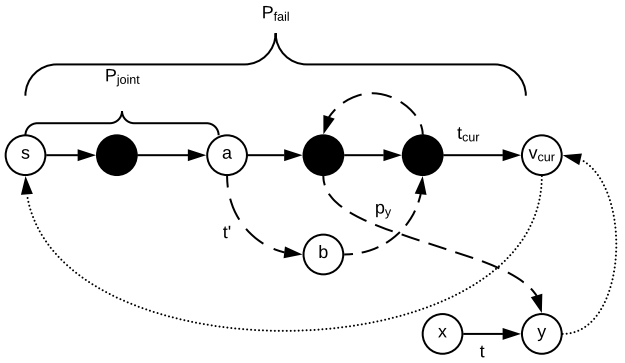


Figure 7: Illustration for proof of Lemma 9 case 2.

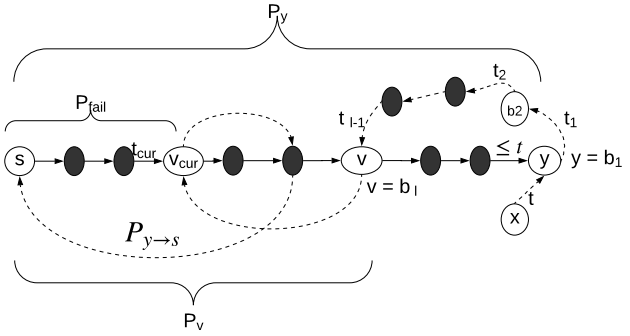


Figure 8: Illustration for proof of Lemma 9 case 2 when $p_{y \to s}$ intersects p_y at another node v other than v_{cur}.

Since $x \xrightarrow{t} y$ is blocked before the execution of Unblock in AllPaths(p_{fail}), at some point earlier in the run, this interaction got blocked. Consider the last time this happened, and let p_y be the prefix at that point in time. We show now that p_{fail} must be a prefix of p_y. Refer to Figure 7 for clarity of this case. Indeed, suppose it isn't, then p_{fail} and p_y have a shared prefix p_{joint} and $p_y \neq p_{joint}$. Let $a \xrightarrow{t'} b$ be the first edge after p_{joint} in p_y. AllPaths($p_y \xrightarrow{t'} b$) hence ended before AllPaths(p_{fail}) started and Unblock() in AllPaths($p_y \xrightarrow{t'} b$) is consistent and therefore unblocks $x \xrightarrow{t} y$ if it is still blocked, because of the existence of the path from b to y (the continuation of p_y) followed by the path from y to s that intersects p_{fail} only at v_{cur}. This path hence intersects $p_{joint} \xrightarrow{t'} b$ at most in b and s.

So, we have established that p_{fail} is a prefix of p_y. Consider the path $p_{y \to s}$ that is obtained by removing $x \xrightarrow{t} y$ from the start of $p_{x \to s}$; that is: $p_{x \to s} = x \xrightarrow{t} p_{y \to s}$. $p_{y \to s}$ intersects p_y at least in v_{cur}, but potentially also in other nodes thereafter as shown in case Figure 8.

Let v be the first such node on $p_{y \to s}$ and let p_v be the prefix of p_y that ends in v, and $p_{y \to v}$ be the prefix of $p_{y \to s}$ that ends in v. Let $p_{y \to v} = b_1 \xrightarrow{t_1} b_2 \ldots \xrightarrow{t_n} b_n$; $b_1 = y$ and $b_n = v$. Because $x \xrightarrow{t} y$ got blocked in AllPaths(p_y) and remains blocked all the time until AllPaths(p_{fail}), after Unblock in AllPaths(p_v), $x \xrightarrow{t} y$ is still blocked. By repeated application of Lemma 8 we can show that $U(b_2) \preceq (b_1, t_1) =$

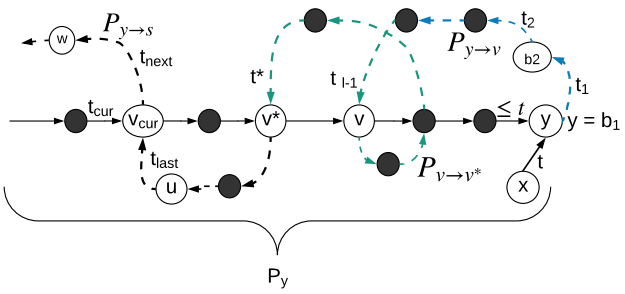


Figure 9: Illustration for proof of Lemma 9 case 2 when $p_{y\to s}$ intersects p_y at more nodes other than v_{cur} such as v and v^*.

(y,t_1), $b_1 \xrightarrow{t_2} b_2$ is blocked, $U(b_3) \preceq (b_2,t_2)$, $b_2 \xrightarrow{t_3} b_3$ is blocked, etc., until $U(v) = U(b_n) \preceq (b_{n-1},t_{n-1})$, $b_{n-1} \xrightarrow{t_n} b_n$ is blocked. We call this sequence $p_{y\to v}$ an *unblock chain* from (v,t_n) till (y,t_1). This implies that any call UNBLOCK(v,t_n) will cause eventually a call to UNBLOCK(y,t_1).

If there is a second intersection v^* on p_y and $p_{y\to s}$ in between v and v_{cur} as shown in Figure 9, then we can follow the same construction and derive an unblock chain from (v^*,t^*) till (v,t_n). This unblock chain can be composed with the first one to form an unblock chain from (v^*,t^*) till (y,t_1). In this way we can continue until we have an unblock chain from (v_{cur},t_{last}) till (y,t_1). t_{last} is the timestamp such that $u \xrightarrow{t_{last}} v_{cur}$ is the interaction arriving in v_{cur} of $p_{y\to s}$. The next interaction on $p_{y\to s}$ leaves v_{cur} and we denote it $v_{cur} \xrightarrow{t_{next}} w$. t_{cur} must be smaller than t_{next}, because $p_{y\to s}$ starts after time t, p_y blocks $x \xrightarrow{t} y$ and hence ends before or at t, and p_{block} is a prefix of p_y. Hence, for sure $t_{cur} < t$, and $t_{next} > t$. Hence, at the start of UNBLOCK in ALLPATHS(p_{block}), $lastp \geq t_{next}$ as we have a path back from v_{cur} to s (the remainder of $p_{y\to s}$). Therefore, the call UNBLOCK$(v_{cur},lastp)$ is made with $lastp > p_{next}$, which will trigger our unblock chain, causing in the end $x \xrightarrow{t} y$ to be unblocked by the call UNBLOCK(y,t_1). This means $x \xrightarrow{t} y$ becomes unblocked, a contradiction and hence proved. □

THEOREM 8. CYCLE(s) *returns all simple cycles.*

PROOF. This follows directly from the combination of Lemma's 6 and 9. □

C. PROOF OF CORRECTNESS OF 2SCENT FOR BUNDLES

LEMMA 10. *Let B be a path bundle. There exists a unique minimal path bundle B' such that $\mathcal{P}(B) = \mathcal{P}(B')$*

PROOF. Suppose $B = v_1 \xrightarrow{T_1} v_2 \xrightarrow{T_2} \dots \xrightarrow{T_k} v_{k+1}$. Let for $i = 1\dots k$, $T_i' := \{t_i \mid \exists v_1 \xrightarrow{t_1} v_2 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_{k+1} \in \mathcal{P}(B)\}$. $B' = v_1 \xrightarrow{T_1'} v_2 \xrightarrow{T_2'} \dots \xrightarrow{T_k'} v_{k+1}$ is now the requested unique minimal path bundle. Indeed, $\mathcal{P}(B') \subseteq \mathcal{P}(B)$ is trivial, since $T_i' \subseteq T_i$ for all $i = 1\dots k$. On the other hand, let $p = v_1 \xrightarrow{t_1} v_2 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_{k+1} \in \mathcal{P}(B)$. Then, by definition of T_i', $v_i \in T_1'$, for all $i = 1\dots k$. Hence, $p \in \mathcal{P}(B')$. As p was chosen arbitrarily, this established the other direction of the inclusion, namely that $\mathcal{P}(B) \subseteq \mathcal{P}(B')$. B' is clearly minimal as every $t_i \in T_i'$ is there because of a path having $v_i \xrightarrow{t_i} v_{i+1}$; removing t_i from T_i' would result in remove at least that path from $\mathcal{P}(B')$.

Suppose now that there exists another minimal path bundle $B'' = v_1 \xrightarrow{T_1''} v_2 \xrightarrow{T_2''} \dots \xrightarrow{T_k''} v_{k+1}$ such that $\mathcal{P}(B'') = \mathcal{P}(B)$. Since B'' is minimal and different from B', there must be at least one $i = 1\dots k$ and one $t \in T_i'$ such that $t \notin T_i''$. However, via a similar argument as for the minimality of B', this would imply that there is a path $p \in \mathcal{P}(B')$ which is not in $\mathcal{P}(B'')$. Therefore $\mathcal{P}(B'') \neq \mathcal{P}(B') = \mathcal{P}(B)$, which is in contradiction with our assumptions. This proves that a minimal B' always exists and is unique. □

LEMMA 11. *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 $v \xrightarrow{T} w$.*

PROOF. Suppose $B = v_1 \xrightarrow{T_1} \dots \xrightarrow{T_k} v_{k+1}$ with $v_1 = u$ and $v_{k+1} = v$. We need to construct a path bundle that contains exactly the paths $\mathcal{P} = \{u \xrightarrow{t_1} \dots \xrightarrow{t_k} v \xrightarrow{t} w \mid u \xrightarrow{t_1} \dots \xrightarrow{t_k} v \in \mathcal{P}B \text{ and } t > t_k \text{ and } t \in T\}$. Since B is minimal, from the proof of 10, we learn that for $t_k in T_k$ if and only if there exists a path $u \dots \xrightarrow{t_k} v \in \mathcal{P}(B)$. Hence, there exists a path in $\mathcal{P}(B)$ with the last interaction at time $t_{min} = \min(T_k)$. This path can be extended by any $v \xrightarrow{t} w$ with $t > t_{min}$ and $t \in T$. Hence, for each $t \in T_{k+1}' := \{t \in T \mid t > t_{min}\}$ there is a path in $\mathcal{P}$ with the last interaction at timestamp t. Also the opposite direction holds; if there is a path in $\mathcal{P}$, then it ends at a timestamp t in T_{k+1}' as it extends a path in $\mathcal{P}B$ and hence the last interaction needs to come after the t_{min}. The minimality of the other T_i' can now be shown by induction. Suppose that T_{i+1}' has the property that $t \in T_{i+1}'$ if and only if there exists a path in $\mathcal{P}$ for which the $i+1$st timestamp is t. Then it is easy to show via a similar argument as above for T_{k+1}' that T_i' are exactly those timestamps for which there exists a path in $\mathcal{P}$ with that ith timestamp. From the proof of 10 it follows now that the resulting bundle as constructed in Algorithm 8 is minimal. □

D. COMPLEXITY OF CONSTRAINED DEPTH-FIRST SEARCH

The proof of complexity revolves around the observation that on the one hand, in order to unblock an interaction, a cycle needs to be output, and on the other hand every cycle that is output unblocks an interaction at most once. To prove the validity of this observation we first establish a more strict condition than consistency that the Unblock operations obey.

LEMMA 12. *For each call to* UNBLOCK *in* ALLPATHS$(pr = s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} v_n)$ *the following holds: if an interaction $x \xrightarrow{t} y$ is blocked before the execution of* UNBLOCK *and free afterwards, then there exists a temporal path $p_{x\to s}$ from x to s such that $V(p_{x\to s}) \cap V(pr) = \{s, v_n\}$ and there does not exist any path $p'_{x\to s}$ such that $V(p'_{x\to s}) \cap V(pr) = \{s\}$.*

PROOF. The existence of the temporal path $p_{x\to s}$ is already established by Lemma 9. Here we will show that if

there exists a path $p'_{x\to s}$ such that $V(p'_{x\to s}) \cap V(pr) = \{s\}$ then $x \xrightarrow{t} y$ cannot be blocked at the start of UNBLOCK in ALLPATHS(pr). According to Lemma 5, $x \xrightarrow{t} y$ cannot be blocked at the start of ALLPATHS(pr). Suppose $x \xrightarrow{t} y$ becomes blocked during ALLPATHS(pr). This must then occur during the execution of ALLPATHS($pr \cdot p_{v_n\to y}$), where $p_{v_n\to y}$ is a path from v_n to y that starts after t_n. Let $v_n \xrightarrow{t_{n+1}} v_{n+1}$ be the first interaction on that path $p_{v_n\to y}$. As UNBLOCK in ALLPATHS($pr \xrightarrow{t_{n+1}} v_{n+1}$) is consistent and $p'_{x\to s}$ is a path from x to s that starts with $x \xrightarrow{t} y$ and intersects $pr \xrightarrow{t_{n+1}} v_{n+1}$ in at most s and possibly v_{n+1}, after UNBLOCK in ALLPATHS($pr \xrightarrow{t_{n+1}} v_{n+1}$), $x \xrightarrow{t} y$ will be free. Hence, after any recursive call from ALLPATHS(pr) returns, $x \xrightarrow{t} y$ is free and hence it will be free at the start of UNBLOCK in ALLPATHS(pr), which establishes our lemma. □

LEMMA 13. *In between two cycles being output every interaction can get unblocked at most once.*

PROOF. The only way an interaction can get unblocked is by a call to Unblock, which is only called when a cycle was found and output. Suppose a cycle $s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} v_n \xrightarrow{t_s} s$ is output. This cycle can only be output by ALLPATHS($s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} v_n$) and will trigger UNBLOCK operations in ALLPATHS($s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_i} v_i$), for all $i = 1 \dots n$. Let $p_i = s \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_i} v_i$. Suppose that there is an edge $x \xrightarrow{t} y$ which gets unblocked twice as a result of this found cycle. Because of Lemma 12 this is however impossible, as any temporal path intersecting p_i only in s and v_i is a path intersecting pr_j only in s for all $j < i$. So, if an interaction gets unblocked in ALLPATHS(pr_i), it cannot get unblocked in ALLPATHS(pr_j) for $j < i$. □

With the last lemma we have almost reached the complexity result we are aiming at; blocked edges do not need to be explored, or at least not after unblock lists have been properly updated. Hence the last hurdle to be taken is showing that once a blocked interaction $x \xrightarrow{t} y$ was considered, we do not ever have to consider it again

THEOREM 9. *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.*

PROOF. We consider the time in between two cycles being output. By Lemma 13 any edge gets unblocked at most once. We say that an edge $x \xrightarrow{t} y$ is *considered* in a step of the algorithm whenever its existence or non-existence matters for the execution of that step. An edge $x \xrightarrow{t} y$ is only considered in calls ALLPATHS(pr) where pr ends in x. Suppose now that $x \xrightarrow{t} y$ was considered and found blocked at that time, did not become free in between, and is considered again. Then, any edge $x \xrightarrow{t'} y$ with $t' \geq t$ does not need to be considered anymore until $x \xrightarrow{t} y$ becomes unblocked again. Indeed, the first time $x \xrightarrow{t} y$ is considered, and no cycle found, $ct(y)$ gets lowered to at most t and (x, t) added to $U(y)$ unless already $U(y) \preceq (x, y)$. Should $x \xrightarrow{t'} y$ with $t' \geq t$ be considered, and there wasn't an unblock operation of $x \xrightarrow{t} y$ in between, $ct(y)$ is still at most t and the interaction will not have any effect (no recursive calls because of it, *lastp* does not get influenced). We can achieve the complexity bound by using a data structure that allows to consider only interactions $v_{cur} \xrightarrow{t} y$ such that never before an interaction $v_{cur} \xrightarrow{t'} y$ was considered and blocked without being freed in the meantime, and $t > t_{cur}$. For this, we will keep for each pair of nodes (x, y) such that there exists an interaction $x \xrightarrow{t} y$ an ordered list of timestamps $t_1 < t_2 < \dots t_n$ of all interactions that took place between x and y, and a pointer to the last timestamp that wasn't considered yet. As, in a call ALLPATHS(pr) we can ignore all interactions (v_{cur}, y, t) with $t \leq t_{cur}$ and all interactions (v_{cur}, y, t) where t comes after the pointer position, we can identify all interactions to be considered in linear time in the number of interactions. If a cycle is output, the complexity is trivially satisfied and we reset the pointer to last timestamp. Otherwise, the pointer will decrease to the last timestamp lower than t_{cur}. Hence, in a subsequent call none of the edges considered will be reconsidered. In this way, as long as there is no cycle output, any interaction get unblocked at most once, and hence any interaction will be considered at most twice. This gives us a total time complexity in between two cycles being output of $\mathcal{O}(|V|+|\mathcal{E}|)$. The term $|V|$ comes from generating all unblock lists and closing time variables at the start of the algorithm. □

E. COMPLEXITY OF SOURCE DETECTION PHASE

THEOREM 10. *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. The time complexity for handling one interaction is bounded by $\mathcal{O}((m+c)W)$, and the memory complexity is $\mathcal{O}(\min(n, W)W)$ assuming the pruning is done every $\mathcal{O}(\omega)$ steps.*

PROOF. For the proof it suffices to notice that because pruning is done every $\mathcal{O}(\omega)$ steps, there are at most $\mathcal{O}(W)$ interactions that need to be taken into account to determine the size of the summaries (pairs resulting from older interactions are removed in the pruning step). Therefore there are at most $\mathcal{O}(\min(W, n))$ summaries maintained holding each at most $\mathcal{O}(W)$ entries. Merging two summaries takes linear time. For each cycle we need to output the set C which has size at most W. □

THEOREM 11. *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 $\mathcal{O}(q \min(W, n))$.*

PROOF. The argument of the proof is similar as for Theorem 6, but with the maximal size W of the summaries $S(a)$ replaced by the fixed size q of the Bloom filters $B(a)$. For merging the forward and backward seeds it suffices to notice that the forward and backward candidates are generated in order, and hence we can merge in linear time. Additionally, For each candidate cycle we have to intersect two Bloom filters. □