2020 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)

A Hybrid Approach to Temporal Pattern Matching

Konstantinos Semertzidis
IBM Research Europe
Dublin, Ireland
konstantinos.semertzidis1@ibm.com

Evaggelia Pitoura
Dept. of Computer Science & Engineering
University of Ioannina, Greece
pitoura@cse.uoi.gr

***Abstract*—Temporal graphs represent relationships and interactions among entities over time, such as those occurring among users in social, transaction, and telecommunication networks. The analysis of their temporal structure help us understand, and predict the behavior of their entities. A typical analysis task in graph networks is the finding of all appearances of an input graph pattern query. Such appearances are called matches. In this paper, we are interested in finding all matches of an interaction pattern query within temporal graphs. To this end, we propose a hybrid approach that achieves effective filtering of potential matches based both on structure and time. Our approach exploits a graph representation where edges are ordered by time. We present experiments with real datasets that illustrate the efficiency of our approach.**

I. Introduction

In this paper, we focus on graphs whose edges model interactions between entities over time. We refer to such networks as *temporal graphs*. In such graphs, each edge is timestamped with the time when the corresponding interaction took place. For example, networks of communication via text messages, or phone calls, can be represented as a sequence of timetamped edges, one for each instantaneous contact between two people. Other examples include biological, social and financial transaction networks. We are interested in finding patterns of interaction within such graphs. Specifically, we assume that we are given as input a graph pattern query P whose edges are ordered and this order specifies the desired order of appearance of the corresponding interactions. We want to find all matches of P in a temporal graph G, that is, the subgraphs of G that match P structurally, and whose edges respect the specified time order. We also ask that all interactions in the matching subgraph appear within a given time interval δ. An example interaction pattern P and temporal graph G are shown in Fig. 1.

Locating patterns of interactions in temporal graphs finds many applications. Take for example communication and transportation networks, where links between nodes and hubs are only established temporarily. Routing policies of computer networks may decide differently at various time points on how to create links between network nodes for data transmission. Transportation networks with delivery vehicles link hubs differently according to current demand. A temporal pattern analysis of the interaction graph can help in mining information from the graph's history which will help us understand, predict and optimize the behavior of those networks.

IEEE/ACM ASONAM 2020, December 7-10, 2020
978-1-7281-1056-1/20/$31.00

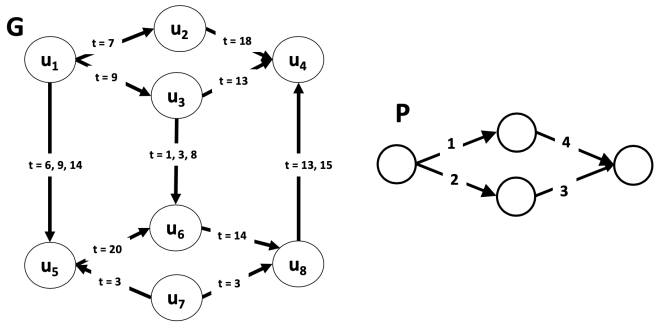


Fig. 1: Temporal graph G and graph pattern P. For clarity, multiple edges between nodes in G are shown as a single edge with multiple timestamps.

Another example, is the collaboration networks, in which it might be interesting to detect in which order a group of researchers formed a network, which researchers only have short-term collaborations and which have sustainable collaborations. Finding interaction patterns are also useful in the case of temporal graphs modeling a network of patients where we are trying to detect a disease contagion by identifying patterns of interactions between patients in a short period of time.

The straightforward approach to locating interaction patterns is to use a state-of-the-art graph pattern algorithm and find the matches in the static graph induced from temporal graph, if we ignore timestamps and multiple edges. Then, for each match we need to find all time-ordered matches. However, even an efficient implementation of this approach incurs large computational costs, since many redundant matches will be generated that do not follow the required temporal order. To this end, we propose an efficient algorithm which uses an edge-based representation of the graph where edges are ordered based on time. This representation allows fast pruning of the candidate matches that do not meet the temporal constraints. We then extent this representation to achieve combined structural and temporal pruning. Our experiments on four real datasets show the efficiency of our approach.

Related Work: There has been recent interest in processing and mining temporal graphs, including among others discovering communities [14], computing measures such as density [1], [12], PageRank [3], [9], and shortest path distances [4], [10], [13]. There have also been previous work on locating temporal motifs and subgraphs [2], [5], [6], [7], [8], [11], [15]. The work

in [11] introduces the problem of finding the most durable matches of an input graph pattern query, that is, the matches that exist for the longest period of time in an evolving graph. An algorithm for finding temporal subgraphs is provided in [5], with the restriction that edges in the motif must represent consecutive events for the node. Temporal cycles are studied in [6] and a heuristic approach for computing an estimation of the numbers of temporal motifs is studied in [2]. The authors of [8] consider a query graph with non time-stamped edges and define as temporal match a subgraph such that all its incoming edges occur before its outgoing edges.

Most closely related to ours is the work in [7], and [15]. The authors of [15] index basic graph structures in the static graph so as to find a candidate set of isomorphic subgraphs quickly at query time, and then verify for each candidate match whether the time conditions hold. The authors of [7] use as similar to ours time ordered representation of the temporal graph but they locate the temporal graphs by a static subgraph matching algorithm and then use a separate algorithm to find all temporal subgraph matches. In summary, these works enumerate matches of a pattern graph using a two phase approach: they first perform structural matching and then they filter the results that do not meet the time conditions. Our work proposes a combined structural-temporal matching approach.

II. Problem definition

A *temporal graph* $G(V, E)$ is a directed graph where V is a set of nodes, E is a set of *temporal edges* (u, v, t) where u and v are nodes in V, and t is a timestamp in $\mathbb{R}$. There may be multiple temporal edges between a pair of nodes capturing interactions appearing at different time instants. An example is shown in Fig. 1.

We use $\Pi(u, v)$ to denote the set of edges from u to v, $\pi(u, v)$ the number of such edges, and $time(e)$ the timestamp of temporal edge e. For example in Fig. 1, the nodes u_1, u_5 are connected through three edges, e.g, $\Pi(u_1, u_5) = \{e_1, e_2, e_3\}$ with $time(e_i) = 6, time(e_2) = 9$, and $time(e_3) = 14$.

The same timestamp may appear more than once modeling simultaneous interactions. Thus, timestamps impose a partial order $\prec_{\mathrm{T}}$ on temporal edges. For a pair of temporal edges, e_i and e_j in E, $e_i \prec_{\mathrm{T}} e_j$, if $time(e_i) < time(e_j)$ and $e_i =_{\mathrm{T}} e_j$, if $time(e_i) = time(e_j)$.

We define the start time, $start\_time(G)$, and end time, $end\_time(G)$, of a temporal graph G as the smallest and largest timestamp appearing in any of the temporal edges in E. We also define the duration, $dur(G)$ of a temporal graph as $end\_time(G)$ - $start\_time(G) + 1$.

We are interested in finding patterns of interactions in a temporal graph that appear within a time period of δ time units. A δ-interaction graph pattern P is a temporal graph with $dur(P) \leq \delta$.

Definition 1: [Interaction Graph Pattern Matching] Given a temporal graph $G(V, E)$, a time window δ and a δ-interaction graph pattern $P = G(V_P, E_P)$, a subgraph $M = G(V_M, E_M)$ of G is a match of P, such that the following conditions hold:

Algorithm 1 InteractionSearch(L_G, L_P, δ)

Require: List of temporal graph edges L_G, list of graph pattern edges L_P, time window δ
Ensure: Interaction matches m

```
1: S ← ∅                                          ▷ stack with candidate matches
2: M ← ∅                                          ▷ interaction matches
3: sT ← δ                                         ▷ remaining time
4: MoreMatches ← True; i ← 1
5: while (MoreMatches) do
6:     e^G ← MatchingEdge(e_i^P, sT)
7:     if e^G ≠ null then
8:         Push(e^G, S)
9:         if i = |E_P| then                      ▷ a compete match is found
10:            M ← M ∪ Content(S)
11:            Pop(S)
12:        else
13:            sT ← sT − time(e^G)
14:            i ← i + 1
15:    else                                       ▷ no match found
16:        if i = 1 then
17:            MoreMatches ← False
18:        else
19:            Pop(S)
20:            i ← i − 1
```

1) (subgraph isomorphism) There exists a bijection f: $V_P \rightarrow V_M$ such that, there is a one-to-one mapping between the edges in E_P and E_M, such that. for each edge (u, v, t) in E_P, there is exactly one edge ($f(u)$, $f(v)$, t') in E_M,
2) (temporal order preservation) For each pair of edges e_1 and e_2 in E_P mapped to edges e'_1 and e'_2 in E_M, it holds $e_1 \prec_{\mathrm{T}} e_2 \Rightarrow e'_1 \prec_{\mathrm{T}} e'_2$ and $e_1 =_{\mathrm{T}} e_2 \Rightarrow e'_1 =_{\mathrm{T}} e'_2$.
3) (within δ) $dur(M) \leq \delta$.

For example, in Fig. 1, the subgraph G' induced by $\{u_1, u_2, u_3, u_4\}$ is a match of P with $dur(G') = 12$.

III. Algorithms

We call *static graph* the directed graph $G_S(V, E_S)$ induced from temporal graph $G(V, E)$, if we ignore timestamps and multiple edges, i.e., (u, v) is an edge in E_S, if and only if, there is a temporal edge (u, v, t) in E.

A straightforward approach is to apply a graph pattern matching algorithm on the static graph G_S and then for each match find all time-ordered matches. However, it is easy to see that for a match M_S in G_S, the number of candidate time-ordered matches can be up to $\prod_{i=1}^{|E_{M_S}|} \pi(e_i^{M_S})$, for each $e_i \in M_S$. In the following, we introduce an alternative approach.

A. Interaction Graph Pattern Matching

Our graph pattern matching algorithm identifies the interaction pattern matches by traversing a representation of the graph G where temporal edges are ordered based on $\prec_{\mathrm{T}}$.

Let L_G be this representation, that is, the list of graph edges of G ordered by $\prec_{\mathrm{T}}$. We also order by $\prec_{\mathrm{T}}$ the edges in the graph pattern, let $L_P = e_1^P, \ldots e_{|E_P|}^P$ be the resulting list. We search for the matches of each graph pattern edge following

the order specified in the input pattern. The algorithm matches the edges in the graph pattern edge-by-edge in a depth-first manner until all edges are mapped and a full matching subgraph is found. The basic steps are outlined in Algorithm 1. We use a stack S to maintain the graph edges that have been mapped to graph pattern edges so far. Index i denotes the edge in the graph pattern to be mapped next. MATCHINGEDGE performs two types of pruning:

(1) temporal pruning: matched edges must (a) follow the time-order and (b) be within δ.
(2) structural pruning: the matched graph must be isomorphic to the pattern subgraph.

When a graph edge is mapped to a pattern edge we verify if it is the last edge to complete the pattern (Lines 9–11). If this is the case, we store the matching graph (Line 10) in matches set M, and we remove the last matched edge in order to locate a new edge that could also lead to a new match. Otherwise, we increase the index by one to look for matches of the next pattern edge, and compute the remaining δ, that is, the time in which the next edge to mapped should be active (Lines 13–14). If there is not any graph edge that can be mapped to the current pattern edge, the algorithm backtracks by removing the last matched edge and reducing the index in order start looking for a new edge that could result to a full match (Lines 18–20). Note that if no graph edge can be mapped to the first pattern edge , we stop the algorithm (Lines 16–17) since no other matches can be found. Next, we describe a simple variation of MATCHINGEDGE termed SIMPLEME and then a more efficient one termed INDEXME.

Simple Temporal-Structural filtering. SIMPLEME shown in Algorithm 2 scans the graph edges in L_G linearly. The key idea is that when we have matched the i-th pattern edge with a graph edge at position j, to match the next $i+1$-th pattern edge, we need to look only at positions in L_G larger than j. As we build the match, we maintain the mapping F of pattern edges to graph edges. We use $F(u)$ to denote the graph node that pattern node u was mapped to. If u is not yet mapped, $F(u)$ is $null$.

Let i_G denote the last position in L_G where a graph edge has been mapped to a graph pattern edge. In SIMPLEME, we just go through the edges of the graph starting just after the position of the previously matched edge (position i^G + 1). For each edge, we check whether the edge satisfies the temporal and graph isomorphism conditions. The maximum number of edges to be checked is equal to the number of edges e_i for which $time(e_i) \leq rT + time(e_i^G)$. For example, if $rT = 5$ and the time of the last mapped edge is 4 then only the edges e_i with $time(e_i) \leq 9$ should be checked. The computational complexity is expected to be $O(\Delta^{|E_P|-1}|E|)$, where Δ is the number of edges that are within the time window δ and need to be visited for each pattern edge in E_P.

Indexed temporal-structural filtering. To reduce the algorithms' complexity, INDEXME uses information about the ingoing and outgoing edges of each node to avoid the linear scanning of the graph. Two structures are used: (a) an additional mapping structure F_{edge}, and (b) an extension of the graph structure L_G with neighborhood information.

Specifically, in addition to the mapping table F, we maintain a mapping table F_{edge}. Assume that pattern node u^P during the course of the algorithm is mapped to graph node u^G. We maintain the matching graph edge u^G in $F(u^P)$, and also in $F_{edge}(u^P)$ the most recently matched graph edge for which u^G was either a source node or a target node. Note that this edge is also the most recent edge. All graph edges that will match the remaining pattern edges must appear later in time

Algorithm 2 SimpleME((u^P, v^P), rT)

Require: Graph pattern edge (u^P, v^P) to be mapped, remaining time rT, mapping F of pattern nodes to graph nodes, position in the graph of the previously matched edge i^G
Ensure: Graph edge (u^G, v^G) matching pattern edge (u^P, v^P)

1: **for** ($i = i^G + 1$ to max) **do**
2: $(u^G, v^G) = L_G[i]$
3: **if** ($time(u^G, v^G) \leq rT$) **then**
4: **if** ($F(u^P) = F(u^P) = null$) **or** ($F(u^P) = null$ **and** $F(v^P) = v^G$) **or** ($F(v^P) = null$ **and** $F(u^P) = u^G$) **or** ($F(u^P) = u^G$ **and** $F(v^P) = v^G$) **then**
5: $i^G = i$ ▷ matching edge found
6: Update map F
7: **return** $((u^G, v^G))$
8: **else**
9: $i \leftarrow i$ + 1
10: **return** (null) ▷ no matching edge found

Algorithm 3 IndexME((u^P, v^P), rT)

Require: Pattern edge (u^P, v^P) to be mapped, remaining time rT, mapping F of pattern nodes to graph nodes, position in the graph of the previously matched edge i^G, list L_{in} and L_{out}
Ensure: Graph edge (u^G, v^G) matching pattern edge (u^P, v^P)

1: **if** $F(u^P) = null$ **and** $F(v^P) = null$ **then**
2: **for** ($i = i^G + 1$ to max) **do**
3: $(u^G, v^G) = L_G[i^G]$
4: **if** ($time(u^G, v^G) \leq rT$) **then**
5: $i^G = i$ ▷ matching edge found
6: Update $F(u^P)$, $F_{edge}(u^P)$, $F(v^P)$, $F_{edge}(v^P)$
7: **return** $((u^G, v^G))$
8: **else** $i \leftarrow i$ + 1
9: **return** (null) ▷ no matching edge found
10: **else if** $F(u^P) \neq null$ **and** $F(v^P) \neq null$ **then**
11: $j \leftarrow max\{F_{edge}(u^P), F_{edge}(v^P)\}$
12: **for** ($i = j$ to max) **do**
13: **if** $j = F_{edge}(u^P)$ **then**
14: search the out-neighbors of $N_s(L_G[j]$ to find $F(v^P)$
15: **else** search the in-neighbors of $N_t(L_G[j]$ to find $F(u^P)$
16: Let m be the edge found
17: **if** ($time(m) \leq rT$) **then**
18: $i^G = m$ ▷ matching edge found
19: Update $F_{edge}(u^P)$, $F_{edge}(v^P)$
20: **return** $((m))$
21: **else** $i \leftarrow i$ + 1
22: **return** (null) ▷ no matching edge found
23: **else if** $F(u^P) \neq null$ **then**
24: search as above using next out $N_s(L_G[j])$
25: **else** i search as above using next in $N_t(L_G[j])$

than this edge.

We also extend L_G. For each edge $e^G = (u^G, v^G)$ in L_G, we maintain in $N_s(u^G, e)$ the next in time edge with source node u^G, in $N_t(u^G, e)$ the next in time edge with target node u^G and similarly $N_s(v^G, e)$ and $N_t(v^G, e)$ for node v^G. We can retrieve all in and out neighbors of a node ordered by time by simply following these lists. For example, to find the out-neighbors of node u, we start by the first edge, say e, where u appears, retrieve $N_s(u, e)$, then $N_s(u, N_s(u, e))$ and so on. We will use the notation $next_{in}(u)$ ($next_{out}(u)$) to denote getting the next in-neighbor (resp. out-neighbor) of u. Note, that each edge is represented by its position in the L_G graph.

Let us now describe the INDEXME algorithm (shown in Algorithm 3) in detail. The algorithm first checks whether any of the pattern edge endpoints has already been mapped. If none of them has already been matched, then (Lines 2–9), it searches for any match satisfying the time constraints (Line 4). Otherwise, if both endpoints have been matched (Lines 10–22), we get one of the endpoints, let this be node u and depending on whether it is a source or a target node, either the $next_{out}(u)$ or $next_{in}(u)$ is used to find the edges containing the other endpoint (Lines 13–15). Next, if the mapped edge is active within the remaining time window the algorithm updates the matched edge index structure and returns the edge (Lines 17–20). In case just one of the two endpoints is matched, say u, we follow the same steps for missing endpoint v by looking into the in-neighbor (resp. out-neighbor) lists of u (Lines 23–25).

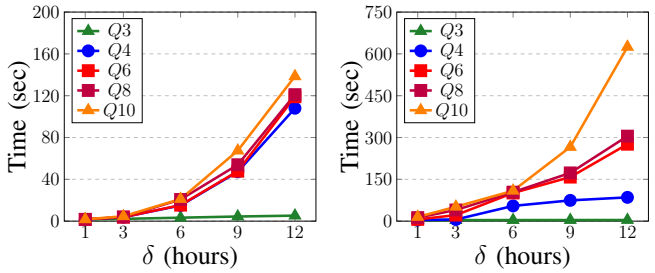
IV. EXPERIMENTAL EVALUATION

We use the following real datasets [7]: (1) **Email-EU** contains emails between members of a European research institution; an edge indicates an email send from a person u to a person v at time t. (2) **FBWall** records the wall posts between users on Facebook located in the New Orleans region; each edge indicates the post of a user u to a user v at a time t. (3) **Bitcoin** refers to the decentralized digital currency and payment system. It consists of all payments made up to October 19, 2014; an edge records the transfer of a bitcoin from address u to address v at a time t. (4) **Superuser** records the interactions on the stack exchange web site Super User; edges indicate the users' answers/comments/replies to comments. The characteristics of the datasets are summarized in Table I.

TABLE I: Dataset characteristics

Dataset	# Nodes	# Static Edges	Edges	# Time (days)
Email-EU	986	24,929	332,334	803
FBWall	46,952	274,086	876,993	1,506
Bitcoin	1,704	4,845	5,121,024	1,089
Superuser	194,085	924,886	1,443,339	2,645

Results: We first compare our algorithm, termed IPM, with the competitive approach followed in [7], [8] which first generates the candidate subgraphs that match the topology of the pattern and then filters the results that do not follow the specified temporal order. The generation of candidate subgraphs is done edge by edge and only the matches active within the δ time window are retained. As our default pattern queries, we use two type of queries: (a) path queries, and (b) random graph pattern queries. Path pattern queries represented as a sequence of edges with consequent timestamps. Random graph pattern queries are generated as follows. For a random query of size n, we select a node randomly from the graph and starting from this node, we perform a DFS traversal until the required number n of nodes is visited. We use as our pattern, the graph created by the union of visited nodes and traveled edges. We set the ordering of edges using the topological ordering of the graph. We report the average performance of 100 random queries for each size n.



(a) Bitcoin (b) Superuser

Fig. 2: Query time for random queries for various query sizes and time windows.

Table II reports the results a) for various δ (for $\delta = 1$ up to 12 hours) and path length = 6, and b) for various path lengths (from 2 up to 10) and $\delta = 1$ hour. As shown, the $Competitor$'s response time increases with δ, while IPM is considerably faster in all datasets. This is because the $Competitor$ tends to generate many redundant matches which do not follow the required temporal order. This is more prominent when we increase the query duration δ. Also, there are cases where a found match m contains edges with multiple timestamps and thus the algorithm must generate a large number of temporal matches. This is the case especially with the Bitcoin and the Email-EU datasets, where there are multiple timestamped edges per static edge. Regarding the path length, the $Competitor$'s response time is increasing with the path length while, again, IPM is considerably faster and not affected much by the query size. IPM takes advantage of the indexes and verifies for each edge with a few lookups whether there is any other valid edge within the remaining time window. Similar results hold for random queries.

Fig. 2 reports the performance of our algorithm for random queries of different graph sizes Qn (from $n = 2$ up to $n = 10$ nodes) and different time windows for our two large datasets. All small queries are processed very fast because the algorithm requires only a few lookups to identify adjacent nodes within the given time window. Moreover, even for largest queries and longest timer windows where larger lookups are required, our algorithm only takes ∼2 and ∼10 minutes to process the query in Bitcoin and Superuser respectively. The performance

TABLE II: Execution time (sec) of IPM and $Competitor$ for varying time window δ and path lengths.

	Email-EU		FBWall		Bitcoin		Superuser	
δ (hours)	IPM	Comp	IPM	Comp	IPM	Comp	IPM	Comp
$\delta = 1$	1.45	21	3.72	42	1.48	15.7	4.07	45
$\delta = 3$	1.56	49	3.85	65	4.24	135	4.92	66
$\delta = 6$	1.86	115	3.92	105	15.7	$>$ 0.5h	4.94	101
$\delta = 9$	2.85	171	4.17	132	48.8	$>$ 0.5h	5.3	127
$\delta = 12$	3.41	190	4.65	152	113	$>$ 0.5h	5.5	160
Path length	IPM	Comp	IPM	Comp	IPM	Comp	IPM	Comp
$l = 2$	0.05	0.3	3.4	10	1	6.5	3.35	15
$l = 4$	0.9	1.05	3.5	28	1.1	10	3.8	18
$l = 6$	1.45	21	3.72	42	1.48	15.7	4.07	45
$l = 8$	1.9	103	4	80	1.63	28	4.93	110
$l = 10$	2.2	115	4.28	85	1.72	36	5.53	121

difference in Superuser dataset is expected since it consists of a much larger number of edges and time instants than Bitcoin. In general, the increase of the time window affects the algorithm performance since for larger time windows, the algorithm has to visit more graph edges to check for a match with the edges in the pattern graph.

V. Conclusions

In this paper, we studied the problem of locating matches of patterns of interactions in temporal graphs that appear within a specified time period. We presented an efficient algorithm based on a representation of graph where edges are ordered based on their interaction time. Our experimental evaluation with real datasets demonstrated the efficiency of our algorithm in finding time-ordered matches. In the future, we plan to study the partial ordering version of the problem that is, repetition of timestamps in the query pattern, which may be interpreted as (a) requiring that the matching graph edges have equal timestamps, or, (b) indifference, no specific order between them. We also intend to investigate the streaming version of the problem where given a stream of graph updates we locate the interaction patterns occurring within a sliding time window.

Acknowledgements

Research work supported by the Hellenic Foundation for Research and Innovation (H.F.R.I.) under the "1st Call for H.F.R.I. Research Projects to Support Faculty Members & Researchers and Procure High-Value Research Equipment" (Project Number: HFRI-FM17-1873).

References

[1] Edoardo Galimberti, Alain Barrat, Francesco Bonchi, Ciro Cattuto, and Francesco Gullo. Mining (maximal) span-cores from temporal networks. In *CIKM*, pages 107–116, 2018.

[2] Saket Gurukar, Sayan Ranu, and Balaraman Ravindran. COMMIT: A scalable approach to mining communication motifs from dynamic networks. In *ACM SIGMOD*, pages 475–489, 2015.

[3] Weishu Hu, Haitao Zou, and Zhiguo Gong. Temporal pagerank on social networks. In *WISE*, pages 262–276, 2015.

[4] Wenyu Huo and Vassilis J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM*, pages 38:1–38:4, 2014.

[5] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[6] Rohit Kumar and Toon Calders. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *PVLDB*, 11(11):1441–1453, 2018.

[7] Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in temporal networks. In *WSDM*, pages 601–610, 2017.

[8] Ursula Redmond and Pádraig Cunningham. Temporal subgraph isomorphism. In *ASONAM*, pages 1451–1452, 2013.

[9] Polina Rozenshtein and Aristides Gionis. Temporal pagerank. In *ECML PKDD*, pages 674–689, 2016.

[10] Konstantinos Semertzidis and Evaggelia Pitoura. Historical traversals in native graph databases. In *ADBIS*, pages 167–181, 2017.

[11] Konstantinos Semertzidis and Evaggelia Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Trans. Knowl. Data Eng.*, 31(1):181–194, 2019.

[12] Konstantinos Semertzidis, Evaggelia Pitoura, Evimaria Terzi, and Panayiotis Tsaparas. Finding lasting dense subgraphs. *Data Min. Knowl. Discov.*, 33(5):1417–1445, 2019.

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

[14] Ding Zhou, Isaac G. Councill, Hongyuan Zha, and C. Lee Giles. Discovering temporal communities from social network documents. In *ICDM*, pages 745–750, 2007.

[15] Andreas Züfle, Matthias Renz, Tobias Emrich, and Maximilian Franzke. Pattern search in temporal social networks. In *EDBT*, pages 289–300, 2018.