

Efficient Simple Temporal Cycle Enumeration on Large Graphs with Lightweight Preprocessing

Qi Liang
Shanghai Jiao Tong University
Shanghai, China
qi_liang@sjtu.edu.cn

Dian Ouyang*
Guangzhou University
Guangzhou, China
dian.ouyang@gzhu.edu.cn

Kang Chen
Li Auto Inc.
Beijing, China
chenkang6@lixiang.com

Fan Zhang
Guangzhou University
Guangzhou, China
zhangf@gzhu.edu.cn

Xuemin Lin
Shanghai Jiao Tong University
Shanghai, China
xuemin.lin@gmail.com

Abstract

Temporal cycles are fundamental patterns in graphs, with important applications in finance, security, and neuroscience. In this work, we study the Simple Temporal Cycle Enumeration (STCE) problem, which aims to enumerate all simple cycles with strictly increasing timestamps within a given time window. However, existing methods, such as 2SCENT, suffer from redundant checks and expensive detection phase, making them inefficient for large-scale or dynamically evolving graphs. To overcome these challenges, we introduce a novel edge-centric framework that treats temporal edges as the core units of exploration. By computing edge offsets in linear time, we eliminate redundant temporal checks, and our constraint-based DFS avoids the expensive detection phase required by prior work. This design ensures polynomial delay and leads to substantial performance gains over existing approaches. Furthermore, we extend our framework to dynamic settings by introducing an efficient incremental update algorithm that selectively identifies affected paths only. Experiments show over an order-of-magnitude speedup on static graphs and up to six orders-of-magnitude improvement for dynamic updates, with most updates completing within 1 ms.

CCS Concepts

• **Theory of computation** → **Graph algorithms analysis**; • **Information systems** → **Query optimization**.

Keywords

Simple Temporal Cycle Enumeration; Temporal Graphs; Dynamic Graph Algorithms

ACM Reference Format:

Qi Liang, Dian Ouyang, Kang Chen, Fan Zhang, and Xuemin Lin. 2026. Efficient Simple Temporal Cycle Enumeration on Large Graphs with Lightweight Preprocessing. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '26)*, August 09–13,

*Dian Ouyang is the corresponding author.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

KDD '26, Jeju Island, Republic of Korea

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2259-2/2026/08

<https://doi.org/10.1145/3770855.3817777>

2026, Jeju Island, Republic of Korea. ACM, New York, NY, USA, 15 pages.
<https://doi.org/10.1145/3770855.3817777>

1 Introduction

Graphs are used to model entities and their relationships across a variety of domains, including finance, cybersecurity, recommendation systems, and more [37, 38, 51, 56, 57]. One common analytical approach for graph-based data is the enumeration of subgraph patterns, such as cycles, cliques, and motifs, which provide insights into structural properties. In this paper, we study the problem of Simple Temporal Cycle Enumeration (STCE), which aims to identify all valid simple temporal cycles under given temporal constraints.

Temporal cycles naturally arise in many real-world domains. In stock trading, cyclic transaction patterns may indicate suspicious activities such as wash trading [20, 22, 39]. In financial transaction networks, fraud and money laundering often manifest as cyclic flows of funds [16, 18, 33, 47]. In biological and neural networks, temporal cycles can represent feedback loops that are essential for understanding system dynamics [10, 25, 30]. In temporal network resilience, temporal cycles have recently been identified as resilience-relevant structural signals in dynamic networks [32]. In closed-loop supply chains, studies show that reverse flows and closed-loop structures strongly affect system dynamics such as the bullwhip effect, highlighting the importance of cyclic dependencies in dynamic supply networks [12, 42]. In these scenarios, it is crucial to respect the temporal order of interactions when identifying cycles. To further ensure temporal coherence and avoid spurious long-range patterns, a time-window constraint is commonly imposed, requiring that the duration between the first and last edges of a cycle does not exceed a given window size w .

The STCE problem has recently attracted significant attention due to its practical importance. The state-of-the-art method 2SCENT [28] employs a two-phase framework to enumerate valid simple temporal cycles. Notably, the second-phase enumeration critically depends on the completion of the first phase, which is mandatory and cannot be bypassed. However, this first phase incurs a high computational cost of $O(m(m+c)W)$, where m is the number of edges, c is the number of valid simple temporal cycles, and W is the maximum number of edges within a time window of size w , making it a major performance bottleneck in practice. Moreover, its vertex-centric exploration leads to excessive redundant temporal

checks: vertices can be revisited many times, and each traversal requires verifying temporal constraints, causing substantial overhead. In addition, existing solutions are not designed to handle graph updates. Since many real-world applications involve continuously evolving data, the inability to support incremental updates makes these algorithms unsuitable for real-time processing.

To address the inefficiencies of vertex-centric methods, we introduce a fundamental shift to an edge-centric paradigm. Instead of treating vertices as the primary units of exploration, our approach organizes edges as the core elements. This shift enables precomputation and reuse of inter-edge relationships (represented as edge offsets), meaning that temporal constraints are checked only once per edge pair rather than repeatedly for every vertex traversal. As a result, redundant computations are drastically reduced, and traversal becomes more efficient and scalable. On top of this edge-centric structure, our algorithm enumerates all valid simple temporal cycles with polynomial delay ($O((c+n)(n+m))$, where n is the number of vertices), ensuring correctness without sacrificing performance, especially in large graphs or those with long time windows.

To support dynamic graph scenarios, we further introduce an Incremental Breakpoint-based DFS algorithm for efficient affected cycle enumeration. By initiating search only from the updated edge and incorporating a breakpoint mechanism, our method identifies newly inserted or removed cycles without revisiting previously discovered ones. In summary, our main contributions are as follows:

- (1) We introduce a lightweight preprocessing edge-centric framework for STCE that computes the edge relationships in $O(m)$ time. This preprocessing avoids repeated temporal compatibility checks between edges and forms the foundation for efficient cycle enumeration.
- (2) We design a constrained DFS algorithm that leverages the edge offsets to learn from failed explorations and avoid revisiting fruitless paths, guaranteeing polynomial delay for cycle enumeration.
- (3) To support dynamic graphs, we develop an efficient update algorithm that selectively explores only the affected regions after edge updates, greatly reducing computation.
- (4) Experiments demonstrate that our method achieves over an order-of-magnitude speedup on static graphs and up to six orders-of-magnitude improvement on dynamic graphs, with the vast majority of updates completed within 1 ms.

2 Background

2.1 Preliminaries

We consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where \mathcal{V} denotes a set of vertices and \mathcal{E} denotes a set of directed edges. Each edge $e \in \mathcal{E}$ is defined as a triplet (u, v, t) , where $u, v \in \mathcal{V}$ and t is a natural number representing the time when the connection between u and v takes place. We assume multiple edges can have the same time, but different edges between two vertices must have distinct times. Given a vertex $u \in \mathcal{V}$, the out-neighbor set of u is defined as $N_{out}(u) = \{v \mid (u, v, t) \in \mathcal{E}\}$. The out-degree of u is denoted as $N^+(u) = |N_{out}(u)|$. We denote $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ as the number of vertices and edges in the graph \mathcal{G} , respectively.

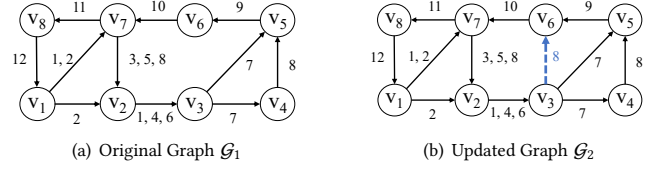


Figure 1: \mathcal{G}_1 is the original graph, and \mathcal{G}_2 is the updated graph after the blue edge insertion.

DEFINITION 1 (TEMPORAL PATH). A temporal path p between two vertices $u, v \in \mathcal{V}$ is a sequence of edges: $((u, v_1, t_1), (v_1, v_2, t_2), \dots, (v_{k-1}, v, t_k))$ such that all edges are in \mathcal{E} and $t_1 < t_2 < \dots < t_k$. We denote the duration of a temporal path p as $dur(p) := t_k - t_1$. We say p is valid for a given time window w if $dur(p) \leq w$.

DEFINITION 2 (TEMPORAL CYCLE). A temporal cycle c rooted at vertex u is a temporal path that starts and ends at u . The cycle c is said to be simple if every intermediate vertex appears at most once.

Problem Statement. Given $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a time window w , the simple temporal cycle enumeration (STCE) problem aims to find all valid simple temporal cycles C with respect to the time window w .

EXAMPLE 1. Figure 1(a) illustrates an example temporal graph \mathcal{G}_1 . The walk $V_7 \xrightarrow{3} V_2 \xrightarrow{1} V_3$ is not a valid temporal path. Given a time window $w = 10$, the path $p = V_1 \xrightarrow{2} V_7 \xrightarrow{11} V_8 \xrightarrow{12} V_1$ forms a valid simple temporal cycle. However, when $w = 5$, p becomes invalid since its duration exceeds the time window. Under $w = 10$, there are 11 valid simple temporal cycles in \mathcal{G}_1 , including, for example, the cycle $V_7 \xrightarrow{3} V_2 \xrightarrow{4} V_3 \xrightarrow{7} V_6 \xrightarrow{9} V_7$.

2.2 Existing Solutions

In this subsection, we introduce two main existing solutions.

(1) A baseline solution [27]. Kumar and Calders addressed the STCE problem by enumerating all simple temporal paths within a given time window w to identify valid cycles [27]. Their algorithm processes temporal edges chronologically, maintaining a list of valid paths. When a new edge (u, v, t) is encountered, each path ending at u that excludes v is extended to v ; if the new path returns to the starting vertex, it is reported as a cycle. Paths violating the time constraint (i.e., $t - t_1 \geq w$) are discarded. Although the method scans edges in one direction only once, the number of paths grows exponentially, making it computationally infeasible for large graphs despite its conceptual simplicity.

(2) 2SCENT [28]. To overcome the limitations of [27], the authors proposed a two-phase algorithm, 2SCENT, for efficiently identifying valid simple temporal cycles. In the first phase (i.e., the detection phase), the algorithm finds all root vertices along with their corresponding start time t_s , end time t_e , and a superset of cycle-related vertices V . It maintains a reverse-reachability set $S(u)$ for each vertex $u \in \mathcal{V}$, where each element (x, t_x) denotes a path from x to u at time t_x . Temporal edges are processed chronologically. Upon encountering edge (a, b, t) , the algorithm adds (a, t) to $S(b)$ and propagates all pairs in $S(a)$ to $S(b)$, removing any (x, t_x) where $t_x < t - w$ to satisfy the time window constraint. If a pair

$(b, t_b) \in S(b)$ is found, it indicates a valid cycle starting and ending at b , with the vertex set $V = \{x \mid (x, t_x) \in S(a), t_b < t_x < t\}$.

The second phase applies a constrained depth-first search (cDFS) to eliminate redundant explorations. For each quadruple of root vertex r , start time t_s , end time t_e , and set of candidate nodes V , the search is performed on a reduced graph $\mathcal{G}' = (V, \mathcal{E}')$, where $\mathcal{E}' = \{(u, v, t) \in \mathcal{E} \mid t \in [t_s, t_e]\}$, starting from each root vertex r . To prune fruitless paths, the algorithm assigns each vertex a closing time: if a vertex is reached after its closing time, it is skipped. Upon discovering a valid cycle, the closing times of its vertices are updated to guide future pruning. This strategy reduces the cycle enumeration complexity to $O((c+1)(m+n))$, where c is the number of valid simple temporal cycles. Although 2SCENT is more efficient than baseline [27], its performance deteriorates on large graphs, especially with increasing window size w . In some cases, it cannot complete the detection phase within a reasonable time. Moreover, neither 2SCENT nor the baseline approach supports dynamic updates due to their inherent computational overhead.

3 Edge-centric Framework

3.1 Motivation

We observe that 2SCENT suffers from two key drawbacks:

(1) Excessive Redundant Checks. 2SCENT treats vertices as the central elements, essentially assuming that the relationships among edges are determined solely by their associated vertices. Consequently, the exploration process is vertex-centric: starting from a vertex, the algorithm searches its neighbors, checks whether the temporal constraint is satisfied, and either continues the exploration or prunes the search accordingly. However, it is important to note that the satisfaction of the temporal constraint must be verified at each step, and these checks cannot be avoided in 2SCENT. Since a vertex can be revisited an exponential number of times during the search, the cumulative overhead of performing these checks becomes substantial, severely impacting the efficiency.

(2) Limitations of the Detection Phase. The detection phase of 2SCENT suffers from fundamental inefficiencies. Specifically, it computes all root vertices and the associated auxiliary information, resulting in a time complexity of $O(m(m+c)W)$. Moreover, the cDFS in 2SCENT critically depends on the completeness of the detection phase; otherwise, the closing time mechanism may incorrectly prune valid exploration paths, leading to the loss of correct results. For example, when starting from a vertex u , cDFS may reach a vertex v and prematurely update its closing time following a failed exploration. This forces subsequent traversals involving v to be pruned. However, such pruning can be erroneous if the failure at v is path-specific, for instance, when the associated timing constraints (i.e., violation of the time window) only apply to the current traversal. In scenarios where a different starting edge is selected, v might still contribute to a valid cycle. Consequently, 2SCENT suffers from poor efficiency and scalability, particularly when applied to large graphs with extended time windows.

To address these two limitations, we break the vertex-centric bottleneck by introducing a novel edge-centric framework. By organizing edges as the core exploration units, inter-edge relationships are computed once and efficiently reused, substantially reducing redundant checks during traversal. Based on this structure, we further

Vertices	E_{out}	Offset	Next Edge
V_1	$(V_1, V_7, 1)$	$[0, 3)$	$(V_7, V_2, 3)$
	$(V_1, V_2, 2)$	$[1, 3)$	$(V_7, V_2, 5)$
	$(V_1, V_7, 2)$	$[0, 3)$	$(V_7, V_2, 8)$
V_2	$(V_2, V_3, 1)$	$[0, 2)$	$(V_2, V_3, 1)$
	$(V_2, V_3, 4)$	$[0, 2)$	$(V_2, V_3, 4)$
	$(V_2, V_3, 6)$	$[0, 2)$	$(V_2, V_3, 6)$
...

Figure 2: The offsets of edges in \mathcal{G}_1

design an algorithm that enumerates all valid simple temporal cycles with polynomial delay, significantly improving both efficiency and scalability. In the following, we first present the edge-based data structure and then detail the full algorithm.

3.2 Edge-based Structure

Organizing edges as fundamental exploration units can substantially reduce redundant checks for temporal constraints such as timestamp ordering and window size. However, explicitly storing all valid successors for each edge incurs prohibitive $O(m^2)$ space. To address this, we propose an edge-centric data structure that groups edges by their source vertex and stores them contiguously in temporal order. Instead of materializing all successors, we record for each edge only the start and end offsets of its valid successors in the edge list, reducing space complexity to $O(m)$.

DEFINITION 3 (EDGE OFFSETS). Let $e = (u, v, t)$ be an edge in a temporal graph and w be a given time window. Denote by $E_{out}(v)$ the list of outgoing edges from vertex v , sorted in ascending order of timestamp. We define the start offset of e , denoted by $set_s(e)$, as the smallest index in $E_{out}(v)$ such that the timestamp exceeds t , and the end offset of e , denoted by $set_e(e)$, as the smallest index in $E_{out}(v)$ such that the timestamp exceeds $t + w$. If no such indices exist, we set the offsets to $N^+(v)$. Then, the valid successor edges of e are exactly the edges in the half-open range $E_{out}(v)[set_s(e) : set_e(e))$.

A naive implementation would scan all subsequent edges to locate the valid successors of each edge, leading to a prohibitive $O(m^2)$ time complexity. Our key observation is that this repeated search is largely redundant: because edges are temporally ordered and grouped by source vertex, the successor set of nearby edges changes only gradually. We therefore compute successor offsets using a sliding-window procedure. For an edge e_i , once the feasible successor interval among the outgoing edges of its destination vertex has been located, the corresponding window boundaries can be advanced monotonically and reused when processing other edges that point to the same vertex. In this way, the algorithm avoids restarting the search from scratch for every edge; instead, it maintains and updates local temporal windows incrementally. Since each pointer only moves forward and each valid successor belongs to the outgoing edge list of some destination vertex, the total amount of pointer movement over all edges is linear in the number of edges. Consequently, all successor offsets can be computed in $O(m)$ time, with the formal proof deferred to the Appendix.

Algorithm 1 outlines the procedure for computing offsets. We first organize all temporal edges into two arrays, E_{out} and E_{in} (Line 1). $E_{out}[v]$ denotes the contiguous block of outgoing edges from

Algorithm 1: Compute Edge Offsets

Input: Temporal graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a time window w
Output: The start and end offsets of each edge

```

1  $E_{out}, E_{in} \leftarrow$  edges grouped by source and target;
2 for  $v \in \mathcal{V}$  do
3   for  $e \in E_{in}[v]$ , ordered ascending w.r.t.  $t$  do
4     if  $e$  is the first edge then
5        $set_s[e] \leftarrow \min\{x \mid 0 \leq x < N^+(v), E_{out}[v][x].t > e.t\}$ 
6        $set_e[e] \leftarrow \min\{x \mid 0 \leq x < N^+(v), E_{out}[v][x].t > e.t + w\}$ 
7        $set_e[e] \leftarrow N^+(v)$ ;
8     else
9        $e' \leftarrow$  the previous edge of  $E_{in}[v]$ ;
10       $set_s[e] \leftarrow \min\{x \mid set_s[e'] \leq x < N^+(v), E_{out}[v][x].t > e.t\}$ 
11       $set_e[e] \leftarrow \min\{x \mid set_e[e'] \leq x < N^+(v), E_{out}[v][x].t > e.t + w\}$ 
12       $set_e[e] \leftarrow N^+(v)$ ;
13 return  $set_s$  and  $set_e$ ;

```

v , while $E_{in}[v]$ contains all incoming edges to v . For each vertex $v \in \mathcal{V}$, we process its incoming edges in temporal order. Given an incoming edge $e \in E_{in}[v]$, we identify its valid successor range in $E_{out}[v]$ by locating (i) the first outgoing edge whose timestamp is larger than $e.t$, and (ii) the first outgoing edge whose timestamp exceeds $e.t + w$. These two positions define the start and end offsets of e ; if the outgoing edge does not exist, the corresponding offset is set to $N^+(v)$ (Lines 4–6). To achieve linear-time complexity, when processing successive incoming edges of the same vertex, we reuse the previously computed offsets as the initial scan positions and advance them monotonically as needed (Lines 7–10).

EXAMPLE 2. Figure 2 illustrates the computed offsets for each edge in graph G . For example, the edge $(V_1, V_2, 2)$ has offsets $(1, 3)$ (assuming indices start from 0), indicating that valid successors from V_2 are $(V_2, V_3, 4)$ and $(V_2, V_3, 6)$. Thus, the edge $(V_2, V_3, 1)$ can be safely skipped during the remaining exploration.

3.3 Filter-based DFS

Building on the precomputed edge offsets, we develop an offset-guided enumeration algorithm, Filter-based DFS (F-DFS). The central idea is to shift temporal reasoning out of the recursive search: instead of repeatedly checking whether a candidate edge satisfies the time-window constraint, F-DFS directly jumps to the precomputed range of valid successors. Thus, each expansion step only considers edges that are already guaranteed to be temporally feasible, enabling efficient enumeration of valid simple temporal paths and, consequently, cycles. This design is particularly effective when the time window w is small. In this regime, feasible paths are naturally short and the main bottleneck is identifying temporally valid next edges. By replacing repeated constraint checks with offset-based access, F-DFS substantially reduces this overhead and achieves high efficiency for small window sizes.

Algorithm 2 presents the algorithm in detail. Lines 1–3 initialize the path vector p and the visited flag array. A depth-first search (DFS) is performed from each edge (Lines 4–5). Upon reaching an edge e_c , it is appended to p and its destination is marked as visited (Lines 7–8). F-DFS then uses offsets to identify the valid successor range of e_c , avoiding redundant temporal checks (Line 9). For each

Algorithm 2: Filter-based DFS

Input: Edges Array E_{out} , a time window w , the edge offsets set_s, set_e
Output: All valid simple temporal cycles

```

1  $p \leftarrow \emptyset$ ;
2 for  $v \in \mathcal{V}$  do
3    $visited[v] \leftarrow false$ ;
4 for  $e \in E_{out}$  do
5    $F\text{-DFS}(e, e)$ ;
6 Procedure  $F\text{-DFS}(e_s, e_c)$ 
7    $p \leftarrow p \cup \{e_c\}$ ;
8    $visited[e_c.to] \leftarrow true$ ;
9   for  $set_s[e_c] \leq i < set_e[e_c]$  do
10     $e' \leftarrow E_{out}[e_c.to][i]$ ;
11    if  $e'.t - e_s.t > w$  then
12      Break;
13    if  $e'.to = e_s.from$  then
14      return  $p \cup \{e'\}$ ;
15    if  $visited[e'.to] = false$  then
16       $F\text{-DFS}(e_s, e')$ ;
17    $p \leftarrow p - \{e_c\}$ ;
18    $visited[e_c.to] \leftarrow false$ ;

```

candidate successor, it first verifies whether the timestamp remains within the time window w (Lines 11–12); since successors are time-ordered, a violation allows early termination. If the successor leads back to the source, a valid temporal cycle is reported (Lines 13–14). If the destination has already been visited, it is skipped to ensure path simplicity (Lines 15–16). Upon backtracking, the edge is removed and the visited flag resets (Lines 17–18).

EXAMPLE 3. Assuming a time window $w = 10$, Figure 3 illustrates a partial DFS search tree rooted at V_1 on \mathcal{G}_1 . Under traditional DFS-based approaches, when the search reaches vertex V_2 , all its outgoing edges must be examined to verify temporal feasibility, including $(V_2, V_3, 1)$, $(V_2, V_3, 4)$, and $(V_2, V_3, 6)$. Although it is straightforward to determine that $(V_2, V_3, 1)$ violates the temporal constraint, this check is repeatedly performed every time the same exploration state is reached, leading to substantial cumulative overhead. In contrast, our F-DFS method leverages precomputed edge offsets to directly identify the valid successor range. As a result, temporally invalid edges such as $(V_2, V_3, 1)$ are skipped entirely during exploration, as indicated by the red dotted edges in Figure 3.

3.4 Polynomial Delay Method

Although F-DFS avoids repeated temporal checks through offset-guided expansion, it may still spend substantial time exploring branches that can never form valid cycles. Such fruitless exploration mainly arises for two reasons. First, the current vertex may have no temporally valid way to return to the source after timestamp t , so any continuation from this state cannot close a cycle. Second, even if a temporal return path exists, all such paths may revisit vertices already on the current search path, thereby violating the simplicity constraint. A natural way to avoid the first type of failure is to add a detection phase that precomputes which vertices can temporally return to the source, and then restrict DFS to these candidates. However, this global precomputation is often expensive and may dominate the total running time.

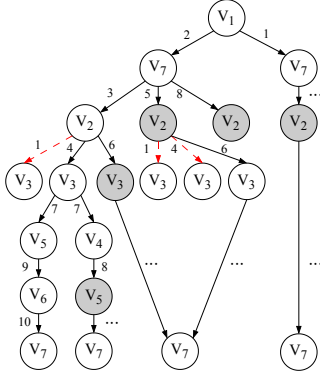


Figure 3: Partial DFS search tree on graph \mathcal{G}_1 with time window $w = 10$. Red dotted edges denote temporally invalid edges that are skipped using precomputed edge offsets, while gray vertices indicate blocked nodes that prune redundant traversal paths, illustrating the efficiency of our strategy.

Our goal is to obtain the pruning power of detection without paying its upfront cost. To this end, we propose Block-based DFS (B-DFS), a feedback-guided enumeration algorithm that removes the detection phase entirely. It allows DFS to explore uncertain branches, but once a branch is discovered to be unable to produce a valid simple temporal cycle, this negative information is recorded and reused to block future searches from entering the same dead end. This feedback mechanism turns failed explorations into reusable pruning knowledge. As the search proceeds, B-DFS progressively refines its understanding of which states are unproductive, thereby avoiding redundant traversal while still eliminating the costly detection phase. Before presenting the algorithm, we introduce a key observation and a supporting lemma.

OBSERVATION 1. *Given a DFS starting from vertex s , if the current exploration path $p = s \rightarrow \dots \rightarrow v$ constitutes a valid simple cycle rooted at s , then the DFS is guaranteed to report it after fully exploring the subtree rooted at v .*

LEMMA 1. *Given a current DFS exploration path $p = s \rightarrow \dots \rightarrow v$, if no valid simple cycle is found after fully exploring the search tree rooted at v , then any valid simple cycle involving an edge explored after reaching v must traverse one or more vertices that were already visited before v (referred to as conflict vertices). Moreover, such a cycle can exist only if these conflict vertices can form a valid simple cycle.*

When performing DFS from a fixed starting edge, there are two cases where the search tree rooted at a vertex v is fully explored without finding a valid simple cycle: (1) no conflict vertices are encountered, and the search eventually terminates at a vertex t ; (2) one or more conflict vertices are encountered, and the search terminates at a conflict vertex v_c . In the first case, all edges explored between v and t can be permanently blocked, as they cannot contribute to any valid cycle. In the second case, according to Lemma 1, if the conflict vertex v_c is later found to be part of a valid simple cycle, then the path from v to v_c may also be valid. These edges should therefore be temporarily blocked and only unblocked once a cycle involving v_c is discovered. This leads to a natural question: how

Algorithm 3: Unblock

Input: Block set B , unblock list U , vertex v , time t

```

1 Procedure Unblock( $B, U, v, t$ )
2   if  $t > B[v]$  then
3      $B[v] \leftarrow t$ ;
4     for  $(u, t') \in U[v]$  do
5       if  $t' < t$  then
6         Unblock( $B, U, u, t'$ );
7          $U[v] \leftarrow U[v] - \{(u, t')\}$ ;

```

can we distinguish explored from unexplored edges and design a mechanism that blocks the former while preserving the latter for future exploration? To address this, we adopt a time-threshold-based vertex blocking strategy, motivated by the following observation.

OBSERVATION 2. *Given a DFS exploration starting from vertex s , let the current path be $p = s \rightarrow \dots \rightarrow v$. After fully exploring the search tree rooted at v , all valid outgoing edges from v whose timestamps are greater than the arrival time at v have been explored.*

Based on Observation 2, we maintain a time threshold t for each vertex v , allowing only incoming edges with timestamps earlier than t to be explored. When a previously blocked edge becomes part of a valid cycle, the threshold of the corresponding vertex is raised to a higher value $t' > t$, enabling re-exploration of previously pruned edges. To support dependency-aware unblocking, each vertex v maintains an unblock list $U(v)$, where each entry (u, t) indicates that once v 's threshold exceeds t , vertex u should be reconsidered, as it may now reach v via a valid temporal path. We adopt a conservative pruning strategy: all edges are initially considered non-contributing, and this assumption is updated only when a valid cycle is discovered. During DFS, each visited vertex sets its time threshold to its arrival time upon first visitation, and thresholds are selectively updated when cycles are found.

Lines 7–30 of Algorithm 4 describe the DFS procedure from a fixed starting edge. The current edge e_c is first appended to the path p (Line 8), and the arrival time at $e_c.to$ is set as its time limit, thereby blocking traversal via later edges (Line 9). A variable *lastp* records the maximum timestamp among all valid cycles found, and is set to 0 if none exists. For each candidate successor edge, the algorithm proceeds as follows. If its timestamp exceeds the window w , the search branch is pruned immediately (Lines 13–14). If the successor returns to the starting vertex, a valid cycle is reported and *lastp* is updated (Lines 15–17). If the successor leads to a blocked vertex, it is skipped and the flag *pass* is set to false (Lines 19–20); otherwise, DFS continues recursively (Lines 21–22). The simplicity constraint is implicitly enforced by vertex time limits, preventing revisits. If no valid cycle is found through e_c , a dependency $(e_c.to, e'.time)$ is added to $U[e'.to]$ (Lines 23–24), enabling future unblocking. Otherwise, *lastp* is updated to the maximum timestamp among all discovered cycles (Lines 25–26), and e_c together with its dependent edges are unblocked via UNBLOCK (Lines 27–28), which is triggered only when the new time exceeds the current limit. Finally, the algorithm backtracks by removing e_c from p and returns whether the current branch yields a valid cycle (Lines 29–30).

Now, the approach performs DFS exploration from each starting edge, which yields a total time complexity of $O((c + m)(m + n))$.

Algorithm 4: Block-based DFS

Input: Temporal edges Array E_{out} , a time window w , the start and end offsets set_s, set_e
Output: All valid simple temporal cycles

```

1  $p \leftarrow \emptyset$ ;
2 for  $v \in \mathcal{V}$  do
3    $\forall x \in V, B[x] \leftarrow \infty$ ;
4    $\forall x \in V, U[x] \leftarrow \emptyset$ ;
5   for  $e \in E_{out}[v]$ , ordered descending w.r.t.  $t$  do
6      $B\text{-DFS}(e, e, B, U)$ ;
7 Procedure  $B\text{-DFS}(e_s, e_c, B, U)$ 
8    $p \leftarrow p \cup \{e_c\}$ ;
9    $B[e_c.to] \leftarrow e_c.time$ ;
10   $lastp \leftarrow 0$ ;
11  for  $set_s[e_c.to] \leq i < set_e[e_c.to]$  do
12     $e' \leftarrow E_{out}[e_c.to][i]$ ;
13    if  $e'.t - e_s.t > w$  then
14       $\text{Break}$ ;
15    if  $e'.to = e_s.from$  then
16       $lastp \leftarrow e'.t$ ;
17       $\text{return } p \cup \{e'\}$ ;
18     $pass \leftarrow false$ ;
19    if  $B[e'.to] \leq e'.t$  then
20       $pass \leftarrow false$ ;
21    else
22       $pass \leftarrow B\text{-DFS}(e_s, e', B, U)$ ;
23    if  $pass = false$  then
24       $U[e'.to] \leftarrow \{e_c.to, e'.t\}$ ;
25    else
26       $lastp \leftarrow \max(e'.t, lastp)$ ;
27  if  $lastp > 0$  then
28     $\text{Unblock}(e_c.to, lastp)$ ;
29   $p \leftarrow p - \{e_c\}$ ;
30  return  $lastp > 0$ ;

```

However, some explorations remain redundant. For instance, if a vertex v is unreachable from the source s , then changing the starting edge from s cannot produce any valid path from v , and all such searches are guaranteed to be fruitless. To enable this pruning, we present the following lemma.

LEMMA 2. *Given a DFS exploration from the source vertex s , if a exploration path fails to return to s when starting from an outgoing edge of s with a later timestamp, then the same path cannot succeed when starting from an outgoing edge with an earlier timestamp.*

Based on Lemma 2, we initiate DFS from the edge with the largest timestamp when multiple edges originate from the same vertex. Blocking information from this initial search can be reused in subsequent traversals. Specifically, if an edge fails to reach the starting vertex in the first search, it remains blocked in later ones. This reuse avoids repeated exploration and reduces the overall time complexity to $O((c + n)(m + n))$. Algorithm 4 outlines the full procedure. Line 1 initializes the path vector p , and for each vertex, the block set B and unblock list U are initialized (Lines 3–4). Edges are explored in descending timestamp order to allow earlier searches to benefit from blocking results of later ones (Line 5).

EXAMPLE 4. *Figure 3 illustrates the search tree on graph \mathcal{G}_1 with time window $w = 10$, starting from V_1 . The algorithm explores the path $V_1 \xrightarrow{2} V_7 \xrightarrow{3} V_2 \xrightarrow{4} V_3 \xrightarrow{7} V_5 \xrightarrow{9} V_6 \xrightarrow{10} V_7$, but backtracks due*

Algorithm 5: Incremental Breakpoint-based DFS

Input: Temporal edges Array E_{out} , a time window w , the new start and end offsets set_s, set_e , the new additional edge e
Output: All new valid simple temporal cycles

```

1 for  $v \in \mathcal{V}$  do
2    $visited[v] \leftarrow false$ ;
3  $IB\text{-DFS}(e, e, B, U, false)$ ;
4 Procedure  $IB\text{-DFS}(e_s, e_c, B, U, breakpoint)$ 
5   if  $breakpoint = false$  then
6     Lines 8-10 in Algorithm 4;
7     for  $e' \in E_{out}[e_c.to]$  and  $e'.t \neq e_c.t$  do
8       Lines 13-20 in Algorithm 4;
9     else
10       $visited[e_c.to] \leftarrow true$ ;
11      if  $e'.t < e_c.t \wedge e_c.t - e'.t \leq w$  then
12         $pass \leftarrow IB\text{-DFS}(e_s, e', B, U, true)$ ;
13      else if  $e'.t > e_c.t$  then
14         $pass \leftarrow IB\text{-DFS}(e_s, e', B, U, false)$ ;
15       $visited[e_c.to] \leftarrow false$ ;
16      Lines 23-29 in Algorithm 4;
17  else
18    Lines 8-12 in Algorithm 4;
19    if  $e'.t \geq e_s.t$  then
20       $\text{Break}$ ;
21    Lines 15-18 in Algorithm 4;
22    if  $B[e'.to] \leq e'.t$  or  $visited[e'.to] = true$  then
23       $pass \leftarrow false$ ;
24    else
25       $visited[e_c.to] \leftarrow true$ ;
26       $pass \leftarrow IB\text{-DFS}(e_s, e', B, U, true)$ ;
27       $visited[e_c.to] \leftarrow false$ ;
28      Lines 23-29 in Algorithm 4;

```

to revisiting V_7 . It then explores an alternate branch via $(V_3, V_4, 7)$ and $(V_4, V_5, 8)$, but terminates early as V_5 is blocked. Compared to Filter-based DFS, which redundantly revisits such paths, Block-based DFS prunes them early by blocking gray vertices. Eventually, a valid cycle $V_1 \xrightarrow{2} V_7 \xrightarrow{11} V_8 \xrightarrow{12} V_1$ is found. The algorithm updates time limits of involved vertices (e.g., $V_8 \leftarrow 12$, $V_7 \leftarrow 11$) and recursively unblocks those dependent on them (e.g., V_6).

THEOREM 1. *The Block-based DFS method only returns all valid simple temporal cycles within a given time window w .*

THEOREM 2. *The Block-based DFS method has a time complexity of $O((c + n)(m + n))$ and a space complexity of $O(m + n)$, where c is the number of all valid simple temporal cycles.*

4 Dynamic Update

Real-world temporal graphs evolve continuously through edge insertions and deletions, making it inefficient to rerun Block-based DFS after every update. Most cycles are unaffected by a single edge update, and the only cycles that can be newly created by an insertion must contain the inserted edge. This suggests a localized update strategy: rather than re-enumerating the entire graph, we repair only the affected offset entries and launch a DFS anchored at the new edge to discover the newly formed cycles. The main challenge is that the inserted edge may not be the earliest edge in a temporal cycle; thus, the search must first identify the temporal breakpoint of the cycle and then enforce temporal consistency from that point

Table 1: Characteristics of Datasets ($K = 10^3, M = 10^6, B = 10^9$)

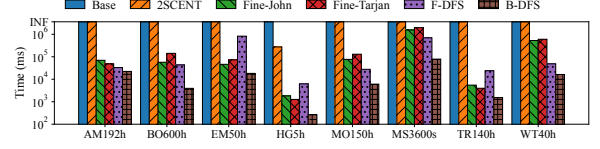
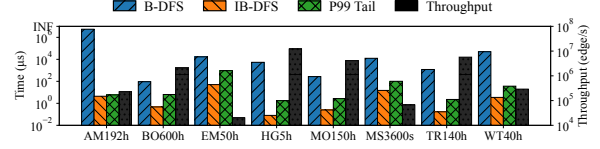
Name	Dataset	$ \mathcal{V} $	$ \mathcal{E} $	Time Span (Days)
BO	bitcoinotc	5.8K	35K	1903
EM	email-Eu	987	332K	803
MO	mathoverflow	24K	506K	2350
TR	transactions	112K	538K	1803
HG	higgs	304K	563K	7
WT	wiki-talk	1.1M	7.8M	2320
MS	messages	364K	26M	1880
AM	AML-data	9.9M	45M	30
FS	Friendster	65M	1.8B	3600

onward. In this section, we present this incremental strategy for edge insertions; edge deletions are handled similarly and discussed at the end of the section.

When a new edge is added, the first step is to insert it into the appropriate position within the outgoing edge list E_{out} and update the corresponding start and end offsets. The update process involves two cases: (1) New edge $e = (u, v, t)$: If e is the first edge to arrive at vertex v , we must recompute the starting and ending offsets for all edges starting from v by scanning the entire $E_{out}[v]$ list. Otherwise, we can similarly update the offsets with Algorithm 1. (2) Edges that arrive at u : We examine all edges that arrive at u to update their cached offsets, based on the insertion position of the new edge in $E_{out}[u]$. Let i denote the index of the newly inserted edge in $E_{out}[u]$, and let (set_s, set_e) be the start and end offsets of an existing edge e' . The updates proceed as follows: If $set_s < i < set_e$, increment set_e by 1. If $i = set_s$, compare the timestamp t' of e' with the new edge's timestamp t : If $t' < t$, then increment set_e by 1; Otherwise, increment set_s by 1. If $i = set_e$ and $t' + w < t$, then increment set_e by 1. In all other cases, no update is necessary.

A key observation is that any newly formed cycle must include the inserted edge. Thus, we initiate a DFS from the new edge to identify all such cycles, with the main challenge being the maintenance of temporal consistency. The new edge may serve as either the earliest (i.e., minimal timestamp) edge in the cycle or appear later. To handle both cases, the algorithm allows any traversal (excluding edges with same timestamp) to find the earliest edge of cycles, if no previously visited edge has a smaller timestamp. Once a smaller timestamp is encountered, a *breakpoint* is triggered, marking the earliest edge in the cycle. From that point on, the algorithm enforces the strict temporal ordering, allowing only increasing timestamps. By distinguishing these two phases and managing breakpoints dynamically, the algorithm efficiently handle with edge insertions.

Algorithm 5 outlines the details of the algorithm. Initially, when the *breakpoint* is unset, the algorithm explores all successors except those with the same timestamp (Line 7). If a successor has an earlier timestamp, the *breakpoint* is triggered (Line 12), enforcing stricter traversal where successors must have strictly increasing timestamps and satisfy $e_c.t - e'.t \leq w$ (Line 11). If the timestamp is later, exploration continues without restriction (Line 14). Once the *breakpoint* is triggered, the algorithm follows Block-based DFS rules with an added constraint: successors must not exceed the starting edge's timestamp (Lines 19–20). A *visited* vector ensures path

**Figure 4: Comparison of running time.****Figure 5: Performance of B-DFS and IB-DFS.**

simplicity (Lines 10, 25), and is reset during backtracking (Lines 15, 27), ensuring correctness.

The Incremental Breakpoint-based DFS algorithm also naturally extends to support edge deletions. To handle the removal of an edge, we first identify and eliminate all cycles that include the deleted edge by initiating a search starting from it. Once all such cycles are removed, we proceed to delete the edge from the outgoing edge list E_{out} and update the corresponding offsets using a strategy similar to that used in edge insertion. The time complexity of the Incremental Breakpoint-based DFS remains $\mathcal{O}((c+1)(m+n))$, and the space complexity is $\mathcal{O}(m+n)$. We omit the detailed proof here, as it closely mirrors the analysis in Theorem 2.

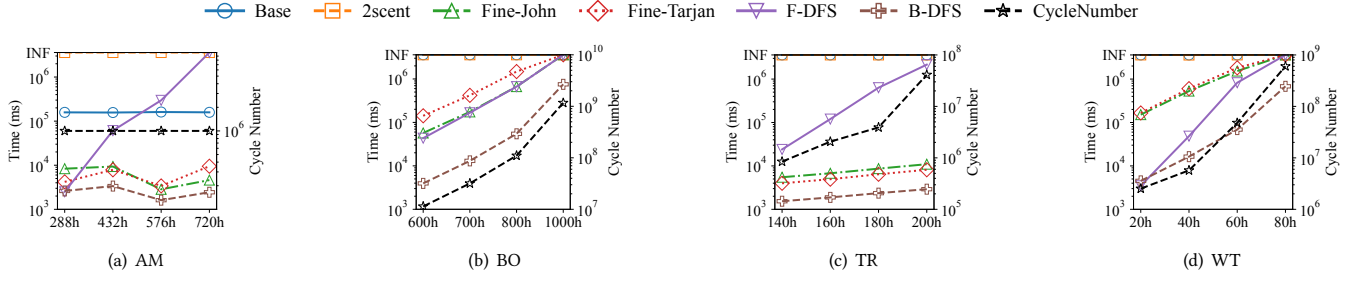
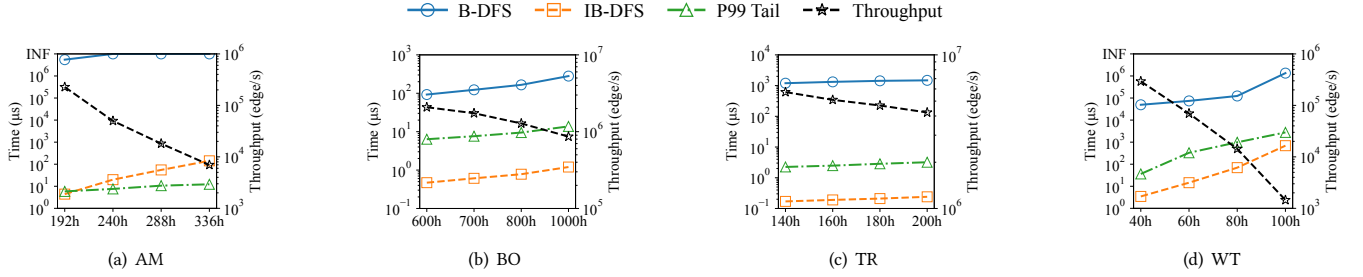
5 Experiments

5.1 Experimental Setup

We evaluate the following methods: (1) **Base** [27]: The baseline method. (2) **2SCENT** [28]: The state-of-the-art algorithm. (3) **Fine-John** [7]: The fine-grained parallel versions of the temporal Johnson algorithm. (4) **Fine-Tarjan** [7]: The fine-grained parallel versions of the temporal Read-Tarjan algorithm. (5) **F-DFS**: Our Filter-based DFS method. (6) **B-DFS**: Our Block-based DFS method. (7) **IB-DFS**: Our Incremental Breakpoint-based DFS method. For Fine-John and Fine-Tarjan, we report results using their single-threaded versions to ensure a fair comparison with our single-threaded algorithms. Table 1 summarizes the basic statistics of the real-world temporal graphs used in our experiments. Additional datasets, detailed experimental settings, and supplementary results are provided in the Appendix.

5.2 Experimental Comparisons

Comparison on Static Graphs. To ensure experimental feasibility, we impose a time limit of an hour (3.6×10^6 ms). Any query that does not complete within this limit is terminated, and its running time is reported as an hour. Figure 4 compares the runtime of different algorithms across multiple datasets, where the time window used in each experiment is indicated next to the graph name. As expected, Base is the least efficient method and consistently exceeds the one-hour time limit due to its exhaustive enumeration of temporal paths. While 2SCENT improves upon Base, it still fails to complete within

Figure 6: Comparison of running time with time window w variedFigure 7: Performance of B-DFS and IB-DFS with time window w varied

the time limit on most datasets. The parallel-inspired methods Fine-John and Fine-Tarjan are able to finish within one hour and may occasionally outperform F-DFS on specific datasets. However, our B-DFS algorithm consistently delivers better performance, with speedups reaching over one order of magnitude in the best cases. **Comparison on Dynamic Graphs.** We evaluate IB-DFS under dynamic updates using three metrics: average update time, throughput (updates per second), and 99th-percentile tail latency. For average runtime, we compare IB-DFS with B-DFS, as other baselines are prohibitively slow in dynamic settings; for throughput and tail latency, we report only IB-DFS, since B-DFS cannot meet real-time requirements. As shown in Figure 5, IB-DFS is up to over six orders of magnitude faster than B-DFS. For instance, on the AM dataset with a 192-hour window, B-DFS requires nearly 10^7 μ s (INF) per update, whereas IB-DFS completes an update in under 10 μ s. This gap stems from their fundamentally different strategies: B-DFS recomputes all cycles upon each update, while IB-DFS incrementally processes only affected regions. Consequently, IB-DFS achieves sub-millisecond 99th-percentile tail latency on most datasets and sustains a throughput above 10^4 updates per second, demonstrating its suitability for high-rate dynamic updates.

Impact of Time Window Size (Static Graphs). Figure 6 reports runtime as the time window size increases, with the number of enumerated cycles shown on the right y-axis. For the AM dataset, the window eventually covers the entire graph; due to runtime limits, we cap the output at 1M cycles. Overall, runtime increases with larger windows, as they expand the search space and produce more valid temporal cycles, consistent with the rising output counts. An exception appears on AM, where runtime decreases for large windows because the 1M cycle cap is reached earlier. Across all

datasets, B-DFS consistently delivers the best performance, with smoothly increasing runtime as the window grows, highlighting its robustness under expanding temporal constraints.

Impact of Time Window Size (Dynamic Graphs). Figure 7 reports performance as the time window length w varies, measured by average running time, throughput, and 99th-percentile tail latency. As expected, the average running time of both B-DFS and IB-DFS increases approximately linearly with w due to the expanding temporal search space. For IB-DFS, tail latency also increases with w , accompanied by a corresponding decrease in throughput. Despite this, IB-DFS exhibits strong scalability and robustness: on most datasets, both average latency and 99th-percentile tail latency remain below 1 ms even for large window sizes, confirming its suitability for dynamic temporal graphs.

Scalability Analysis. We evaluate scalability on the FS dataset. For static algorithms, we cap the output at 1M cycles due to runtime limits; no cap is applied in the dynamic setting. As shown in Figure 8(a), Base, 2SCENT, Fine-John, Fine-Tarjan, and F-DFS all fail to enumerate 1M cycles: Base, Fine-John, and Fine-Tarjan exceed the 256 GB memory limit, 2SCENT and F-DFS cannot reach the cap within one hour. In contrast, B-DFS scales stably, with runtime growing approximately linearly as graph size increases. Figure 8(b) reports the dynamic performance of IB-DFS. Both average and tail latency increase linearly with graph size, while throughput degrades smoothly without sharp drops. These results confirm the robustness and scalability of IB-DFS on large dynamic graphs.

Memory Cost. Figure 9 reports the memory consumption of all evaluated algorithms. For methods that do not finish within one hour, we report their memory usage at termination. Overall, on the majority of datasets, F-DFS and B-DFS incur lower memory

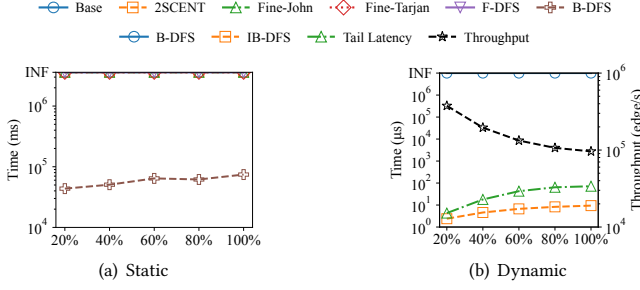


Figure 8: Performance on FS having different sizes

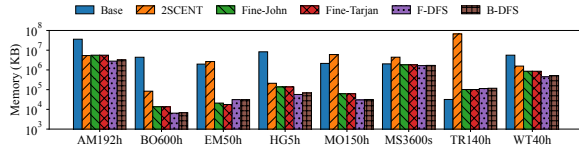


Figure 9: Comparison of memory consumption (KB)

overhead than existing methods. These results demonstrate that our methods are both computation and memory efficient.

5.3 Ablation Experiments

BDFS-NF denotes a variant that applies the blocking mechanism without using edge offsets. Figure 10 shows that B-DFS consistently achieves the best performance, as it simultaneously eliminates redundant temporal constraint checks and avoids repeated exploration of fruitless paths. In contrast, F-DFS only removes redundant temporal checks, while BDFS-NF only prevents repeated useless exploration. This study clearly demonstrates that both techniques are individually effective and, more importantly, complementary—together enabling the superior efficiency of B-DFS.

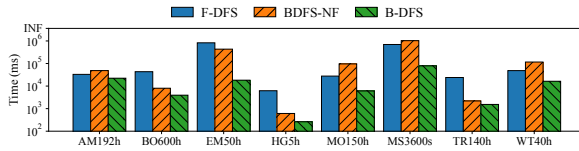


Figure 10: Ablation experiment

5.4 Case Study

Figure 11 illustrates a transaction subgraph extracted from IBM financial transaction data, where the red edges highlight a suspicious laundering flow. Each edge is annotated with its transaction time (e.g., 09/01 04:04 denotes 04:04 on September 1, 2022). The intended takeaway is not merely that these red edges form a temporal cycle, but that such a cycle captures a meaningful laundering behavior: funds originate from Account A, pass through a chain of intermediary accounts (B,C,D,E,F) in strictly increasing temporal order, and eventually return to A within a bounded time window. This closed, time-respecting circulation is consistent with the layering behavior commonly used to obscure fund provenance. Importantly,

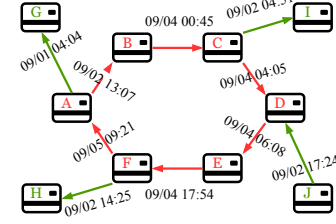


Figure 11: A time-respecting money laundering cycle on financial transaction data

this pattern cannot be directly replaced by static motifs or cliques, which ignore temporal order and therefore fail to distinguish a genuine time-ascending fund flow from a set of unordered transactions. This case study demonstrates why enumerating simple temporal cycles is practically meaningful for financial monitoring, and motivates the need for efficient STCE algorithms in real-world dynamic transaction networks.

6 Conclusion

For the simple temporal cycle enumeration problem, we propose a novel edge-centric framework that treats temporal edges to eliminate redundant temporal checks. Based on this design, we develop an efficient constraint-based DFS algorithm with polynomial delay and no detection phase overhead. To handle dynamic graphs, we further introduce an incremental update algorithm that selectively explores affected paths using a breakpoint mechanism. Experiments show over an order-of-magnitude speedup on static graphs and up to six orders-of-magnitude improvement for dynamic updates, with most updates completing within 1 ms.

Acknowledgments

The work of Dian Ouyang was supported by the National Natural Science Foundation of China No. 62502105, the Guangdong Basic and Applied Basic Research Foundation 2023A151110592 and Science and Technology Guangzhou Education Department Foundation (No. 2023KQNCX057). The work of Fan Zhang was supported by the National Natural Science Foundation of China (62572140, 62536007), the Guangdong Basic and Applied Basic Research Foundation (2024A151011501). The work of Xuemin Lin was supported by the National Natural Science Foundation of China (NSFC) under Grant U2241211.

References

- [1] Florian Adriaens, Cigdem Aslay, Tijl De Bie, Aristides Gionis, and Jeffrey Lijffijt. 2019. Discovering Interesting Cycles in Directed Graphs. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. ACM, Beijing China, 1191–1200.
- [2] David Aleja, Julio Flores, Eva Primo, and Miguel Romance. 2024. Time-dependent personalized PageRank for temporal networks: Discrete and continuous scales. *Chaos: An Interdisciplinary Journal of Nonlinear Science* 34, 8 (2024).
- [3] Erik Altman, Jovan Blanuša, Luc Von Niederhäusern, Béni Egressy, Andreea Anghel, and Kubilay Atasu. 2023. Realistic synthetic financial transactions for anti-money laundering models. *Advances in Neural Information Processing Systems* 36 (2023), 29851–29874.
- [4] Michael A Bender, Jeremy T Fineman, Seth Gilbert, and Robert E Tarjan. 2015. A new approach to incremental cycle detection and related problems. *ACM Transactions on Algorithms (TALG)* 12, 2 (2015), 1–22.

- [5] Elisabetta Bergamini, Henning Meyerhenke, and Christian I. Staudt. 2014. Approximating betweenness centrality in large evolving networks. In *2015 Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 133–146.
- [6] Sayan Bhattacharya and Janardhan Kulkarni. 2020. An improved algorithm for incremental cycle detection and topological ordering in sparse graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2509–2521.
- [7] Jovan Blanuša, Kubilay Atas, and Paolo Ienne. 2023. Fast parallel algorithms for enumeration of simple, temporal, and hop-constrained cycles. *ACM Transactions on Parallel Computing* 10, 3 (2023), 1–35.
- [8] Hanqing Chen, Shuai Ma, Junfeng Liu, and Lizhen Cui. 2025. Discovery of Temporal Network Motifs. *IEEE Transactions on Knowledge and Data Engineering* (2025).
- [9] Davi de Andrade, Jãelino Ara-Áejo, Allen Ibiapina, Andrea Marino, Jason Schoeters, and Ana Silva. 2025. Temporal Cycle Detection and Acyclic Temporization. *arXiv preprint arXiv:2503.02694* (2025).
- [10] Chao-Yi Dong, Dongkwan Shin, Sunghoon Joo, YoonKey Nam, and Kwang-Hyun Cho. 2012. Identification of feedback loops in neural networks based on multi-step Granger causality. *Bioinformatics* 28, 16 (2012), 2146–2153.
- [11] Jessica Enright, Kitty Meeks, and Hendrik Molter. 2025. Counting temporal paths. *Algorithmica* (2025), 1–47.
- [12] Rebecca Fussone, Roberto Dominguez, Salvatore Cannella, and Jose M. Framinan. 2024. Bullwhip Effect in Closed-Loop Supply Chains with Multiple Reverse Flows: A Simulation Study. *Flexible Services and Manufacturing Journal* 36, 1 (March 2024), 250–278.
- [13] Roberto Grossi, Andrea Marino, and Luca Versari. 2018. Efficient algorithms for listing k disjoint st-paths in graphs. In *LATIN 2018: Theoretical Informatics: 13th Latin American Symposium, Buenos Aires, Argentina, April 16–19, 2018, Proceedings* 13. Springer, 544–557.
- [14] Anshul Gupta and Toyotaro Suzumura. 2021. Finding All Bounded-Length Simple Cycles in a Directed Graph.
- [15] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E Tarjan. 2012. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Transactions on Algorithms (TALG)* 8, 1 (2012), 1–33.
- [16] László Hajdu and Miklós Krész. 2020. Temporal network analytics for fraud detection in the banking sector. In *International Conference on Theory and Practice of Digital Libraries*. Springer, 145–157.
- [17] Kongzhang Hao, Long Yuan, and Wenjie Zhang. 2021. Distributed hop-constrained st simple path enumeration at billion scale. *Proceedings of the VLDB Endowment* 15, 2 (2021), 169–182.
- [18] F Hoffmann and D Krasle. [n. d.]. Fraud detection using network analysis, 2015. *EP Patent App. EP20 140* ([n. d.]).
- [19] Weishu Hu, Haitao Zou, and Zhiguo Gong. 2015. Temporal pagerank on social networks. In *Web Information Systems Engineering–WISE 2015: 16th International Conference, Miami, FL, USA, November 1–3, 2015, Proceedings, Part I* 16. Springer, 262–276.
- [20] Md Nazrul Islam, SM Rafizul Haque, Kaji Masudul Alam, and Md Tarikuzzaman. 2009. An approach to improve collusion set detection using MCL algorithm. In *2009 12th International Conference on Computers and Information Technology*. IEEE, 237–242.
- [21] Jarosław Jankowski, Radosław Michalski, and Piotr Bródka. 2017. Spreading processes in multilayer complex network within virtual world.
- [22] Zhi-Qiang Jiang, Wen-Jie Xie, Xiong Xiong, Wei Zhang, Yong-Jie Zhang, and Wei-Xing Zhou. 2013. Trading networks, abnormal motifs and stock manipulation. *Quantitative Finance Letters* 1, 1 (2013), 1–8.
- [23] Donald B Johnson. 1975. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4, 1 (1975), 77–84.
- [24] Leskovec Jure. 2014. SNAP Datasets: Stanford large network dataset collection. Retrieved December 2021 from <http://snap.stanford.edu/data> (2014).
- [25] Steffen Klamt and Axel von Kamp. 2009. Computing paths and cycles in biological interaction graphs. *BMC bioinformatics* 10 (2009), 1–11.
- [26] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. 2011. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment* 2011, 11 (2011), P11005.
- [27] Rohit Kumar and Toon Calders. 2017. Finding simple temporal cycles in an interaction network. In *TD-LSG@ PKDD/ECML, Skopje, Macedonia*. 3–6.
- [28] Rohit Kumar and Toon Calders. 2018. 2scent: An efficient algorithm to enumerate all simple temporal cycles. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1441–1453.
- [29] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.
- [30] Yung-Keun Kwon and Kwang-Hyun Cho. 2007. Analysis of feedback loops and robustness in network evolution based on Boolean models. *BMC bioinformatics* 8 (2007), 1–9.
- [31] Zhengmin Lai, You Peng, Shiyu Yang, Xuemin Lin, and Wenjie Zhang. 2021. Pefp: Efficient k -hop constrained st simple path enumeration on fpga. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1320–1331.
- [32] Baochen Li, Alfiya Abinova, and Shouwei Li. 2026. Persistent Cycles and Network Resilience: A Hypernetwork-Based Framework for Temporal Graph Analysis. *Scientific Reports* 16, 1 (March 2026), 14506.
- [33] Xiangfeng Li, Shenghua Liu, Zifeng Li, Xiaotian Han, Chuan Shi, Bryan Hooi, He Huang, and Xueqi Cheng. 2020. Flowscope: Spotting money laundering based on graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 4731–4738.
- [34] Penghang Liu, Valerio Guarrasi, and Ahmet Erdem Saryüce. 2021. Temporal network motifs: Models, limitations, evaluation. *IEEE Transactions on Knowledge and Data Engineering* 35, 1 (2021), 945–957.
- [35] Laishui Lv, Kun Zhang, Ting Zhang, Dalal Bardou, Jiahui Zhang, and Ying Cai. 2019. PageRank centrality for temporal networks. *Physics Letters A* 383, 12 (2019), 1215–1222.
- [36] Prabhakar Mateti and Narsingh Deo. 1976. On algorithms for enumerating all circuits of a graph. *SIAM J. Comput.* 5, 1 (1976), 90–99.
- [37] Nav Mathur. 2017. *Graph Technology for Financial Services*. Technical Report. Technical Report. Neo4J. 1–14 pages. Retrieved from <https://neo4j.com/use...>
- [38] Steven Noel, Eric Harley, Kam Him Tam, Michael Limiero, and Matthew Share. 2016. CyGraph: graph-based analytics and visualization for cybersecurity. In *Handbook of statistics*. Vol. 35. Elsevier, 117–167.
- [39] Girish Keshav Palshikar and Manoj M Apte. 2008. Collusion set detection using graph clustering. *Data mining and knowledge Discovery* 16 (2008), 135–164.
- [40] {Min Jia} Pan, {Rong Hua} Li, {Yu Hai} Zhao, and {Guo Ren} Wang. 2020. Fast Temporal Cycle Enumeration Algorithm on Temporal Graphs. *Journal of Software* 31, 12 (2020), 3823–3835.
- [41] Raj Kumar Pan and Jari Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* 84, 1 (2011), 016105.
- [42] Christos I. Papanagnou. 2022. Measuring and Eliminating the Bullwhip in Closed Loop Supply Chains Using Control Theory and Internet of Things. *Annals of Operations Research* 310, 1 (March 2022), 153–170.
- [43] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the tenth ACM international conference on web search and data mining*. 601–610.
- [44] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Hop-constrained st Simple Path Enumeration: Towards Bridging Theory and Practice. *Proc. VLDB Endow.* 13, 4 (2019), 463–476.
- [45] Jacob Ponsstein. 1966. Self-avoiding paths and the adjacency matrix of a graph. *SIAM J. Appl. Math.* 14, 3 (1966), 600–609.
- [46] Alexandra Porter, Baharan Mirzasoleiman, and Jure Leskovec. 2022. Analytical models for motifs in temporal networks. In *Companion Proceedings of the Web Conference 2022*. 903–909.
- [47] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1876–1888.
- [48] VV Bapswara Rao and VGK Murti. 1969. Enumeration of all circuits of a graph. *Proc. IEEE* 57, 4 (1969), 700–701.
- [49] Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. 2014. Efficiently listing bounded length st-paths. In *International Workshop on Combinatorial Algorithms*. Springer, 318–329.
- [50] Polina Rozenshtein and Aristides Gionis. 2016. Temporal pagerank. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2016, Riva del Garda, Italy, September 19–23, 2016, Proceedings, Part II* 16. Springer, 674–689.
- [51] Aditya Singh, Anubhav Gupta, Hardik Wadhwa, Siddhartha Asthana, and Ankur Arora. 2021. Temporal Debiasing Using Adversarial Loss Based GNN Architecture for Crypto Fraud Detection. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Pasadena, CA, USA, 391–396.
- [52] Shixuan Sun, Yuhang Chen, Bingsheng He, and Bryan Hooi. 2021. Pathenum: Towards real-time hop-constrained st path enumeration. In *Proceedings of the 2021 international conference on management of data*. 1758–1770.
- [53] John Tang, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. 2009. Temporal distance metrics for social network analysis. In *Proceedings of the 2nd ACM workshop on Online social networks*. 31–36.
- [54] Robert Tarjan. 1973. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.* 2, 3 (1973), 211–216.
- [55] James C Tiernan. 1970. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM* 13, 12 (1970), 722–726.
- [56] Fei Wang, Peng Cui, Jian Pei, Yangqiu Song, and Chengxi Zang. 2020. Recent advances on graph analytics and its applications in healthcare. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3545–3546.
- [57] Jim Webber. 2021. *Powering real-time recommendations with graph database technology*. Technical Report. Technical Report. Neo4J. 1–7 pages. Retrieved from <https://neo4j.com/use...>

- [58] Danni Wu, Yuanyuan Xu, Xuemin Lin, Wenjie Zhang, and Ying Zhang. 2026. Understanding Evolving Graph Structures for Large Discrete-Time Dynamic Graph Representation. *Proceedings of the VLDB Endowment* 19, 5 (2026), 862–875.
- [59] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.
- [60] Ye Wu, Changsong Zhou, Jinghua Xiao, Jürgen Kurths, and Hans Joachim Schellnhuber. 2010. Evidence for a bimodal distribution in human communication. *Proceedings of the national academy of sciences* 107, 44 (2010), 18803–18808.
- [61] Yuanyuan Xu, Danni Wu, Xuemin Lin, Dong Wen, Wenjie Zhang, Lei Chen, and Ying Zhang. 2026. Exploring Sequential Dynamics on Temporal Graphs via Composite Filtering. In *Proceedings of the ACM Web Conference 2026*. 487–498.
- [62] Yuanyuan Xu, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2025. Unidyg: a unified and effective representation learning approach for large dynamic graphs. *IEEE Transactions on Knowledge and Data Engineering* (2025).
- [63] Yuanyuan Xu, Wenjie Zhang, Ying Zhang, Maria Orlowska, and Xuemin Lin. 2024. TimeSGN: Scalable and effective temporal graph neural network. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 3297–3310.
- [64] Yuanyuan Xu, Wenjie Zhang, Ying Zhang, Xiwei Xu, and Xuemin Lin. 2025. Fast and accurate temporal hypergraph representation for hyperedge prediction. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*. 1727–1738.
- [65] Jijiang Zhang, Shiyu Yang, Dian Ouyang, Fan Zhang, Xuemin Lin, and Long Yuan. 2023. Hop-constrained ST simple path enumeration on large dynamic graphs. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 762–775.

A Appendix

A.1 Theorem and Proof

Theorem. Both the space and time complexity of computing the edge offsets are $O(m)$.

Proof. Each edge stores two offset values in addition to its own information, resulting in a total space complexity of $O(m)$. During execution, Algorithm 1 iterates over each incoming edge exactly once. To compute the corresponding start and end offsets, it scans through the outgoing edges of the relevant vertex, where each outgoing edge is examined at most twice, once for validating the start position and once for the end position. As a result, the overall time complexity is also $O(m)$.

Proof of Theorem 1. The core idea of Block-based DFS is to perform a truncated depth-first search: all valid paths are explored except those pruned by blocked vertices. Vertices may be blocked in two cases. First, if a vertex appears in the current path, revisiting it would violate the simplicity constraint—a standard DFS pruning rule. Second, a vertex may be blocked during backtracking, even if not currently in the path. Suppose we explore $v \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_k$, and determine after exploring v_k 's subtree that it cannot lead to a valid cycle. This occurs when: (1) v_k cannot reach the starting vertex v within the time window w ; or (2) any path from v_k to v revisits an earlier vertex (e.g., v_i), violating simplicity. In the latter case, v_k can only be unblocked if v_i is unblocked, which happens only after v_i 's subtree is fully explored. This dependency-aware blocking and unblocking mechanism guarantees that all and only valid simple temporal cycles are explored.

Proof of Theorem 2. The time complexity primarily depends on the number of output cycles and the cost of edge exploration and unblocking. Note that a vertex can only be unblocked through a call to the UNBLOCK procedure, which is invoked exclusively when a valid cycle is found and reported. Consider a valid cycle $v \xrightarrow{t_1} v_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} v_k \xrightarrow{t_{k+1}} v$. Once this cycle is output, the algorithm invokes UNBLOCK in reverse along the prefix path: first on v_k , then v_{k-1} , and so on, down to v_1 . Crucially, each call to

UNBLOCK only unblocks vertices that were blocked solely due to the current vertex. This implies that each edge can be unblocked at most once between two cycle outputs. Consequently, each edge may be involved in at most two operations between any two outputs: one for being unblocked, and one for being explored. In the worst case, where no further cycles are found, the algorithm explores all reachable edges until all vertices are blocked, costing $O(n(m+n))$ times. Hence, the overall time complexity is $O((c+n)(m+n))$. As for space complexity, the algorithm maintains a time limit for each vertex, which requires $O(n)$ space. Each vertex also stores an unblock list to record dependencies, with the total cost bounded by $O(m)$. Hence, the total space complexity is $O(m+n)$.

Proof of Lemma 1. According to Observation 1, if there exists at least one valid path from v to the starting vertex s , but no valid simple cycle is discovered after fully exploring the search tree rooted at v , then the failure must be attributed to violations of the simplicity constraint. As a result, any potential cycle involving an edge explored after reaching v must traverse one or more of the conflict vertices. Since the cycle must be simple, the subpath passing through the conflict vertices must be part of the valid simple cycle.

Proof of Lemma 2. There are two possible reasons why a path cannot reach the starting vertex within the time window: (1) There exists no path to the source vertex; or (2) The path can reach the source vertex, but its arrival time falls outside the specified time window. In the first case, the conclusion is immediate. In the second case, if the path violates the time window constraint when starting from a later timestamp, then starting from an earlier timestamp can only further shorten the available arrival time budget, making it even less likely to satisfy the constraint.

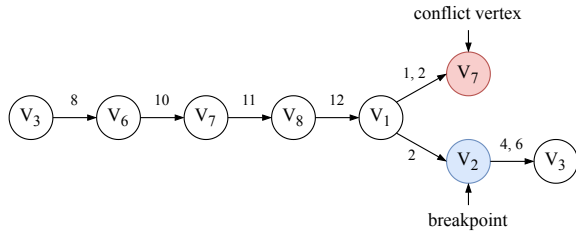
A.2 Other Related Works

s-t Path (or Cycle) Enumeration. Recent studies have primarily focused on hop-constrained s-t simple path enumeration. Gupta and Suzumura [14] study bounded-length simple cycle enumeration in static directed graphs, but their setting does not consider temporal ordering or time-window constraints. Several theoretical works [13, 49] achieve polynomial delay. Peng et al. [44] propose BC-BFS with a “never repeat mistakes” strategy, improving on earlier methods. Sun et al. [52] also propose an index-based method for real-time enumeration of hop-constrained s-t simple paths. HP-index [47] maintains paths between hop vertices whose degrees exceed a specified threshold, enabling real-time enumeration of hop-constrained cycles in large dynamic graphs. For dynamic graphs, Zhang et al. [65] design an efficient partial-path index. Other works include FPGA-based solutions [31] and distributed algorithms [17]. These methods, however, are designed specifically for hop-constrained settings and are not applicable to temporal cycle enumeration.

The enumeration of all simple s-t paths or cycles in static graphs has been extensively studied [23, 36, 45, 48, 54, 55]. Notably, Johnson's algorithm [23] efficiently enumerates simple cycles in directed graphs using DFS with vertex blocking, achieving $O((c+1)(m+n))$ time. However, these classic methods are not applicable to temporal graphs as they do not account for temporal constraints. [40] proposes an algorithm for enumerating hop-constrained temporal cycles. When the hop constraint is removed, their method degenerates to the 2SCENT. As our work focuses on unconstrained simple

Table 2: Characteristics of Datasets ($K = 10^3, M = 10^6$)

Name	Dataset	$ \mathcal{V} $	$ \mathcal{E} $	Time Span (Days)
BA	bitcoinalpha	3.7K	24K	1901
CO	CollegeMsg	1.9K	59K	193
AU	askubuntu	159K	964K	2613
SM	SMS	44K	548K	338
NL	wiki-dynamic-nl	1.1M	20M	3602
SU	superuser	194K	1.4M	2773
FR	friends	624K	12M	1826
SO	StackOverflow	2.4M	17M	2774

**Figure 12: The search branch starting at the new edge**

temporal cycle enumeration, and 2SCENT already serves as the representative baseline for this setting, we do not include this method in our experimental comparison. Blanuša et al. [7] present scalable parallelizations of Johnson and Read-Tarjan algorithms for enumerating simple, temporal, and hop-constrained cycles. We use single-threaded versions to ensure a fair comparison with our single-threaded algorithms. Furthermore, several studies [4, 6, 15] address cycle detection in dynamic graphs rather than full enumeration, and thus fall outside the scope of our problem setting. There also exist studies on counting temporal or static cycles [9, 11]. Adriaens et al. [1] study discovering interesting cycles in directed graphs, where the goal is to retrieve cycles by an interestingness measure rather than exhaustively enumerate all valid temporal cycles. However, such approaches are not directly applicable to STCE and are therefore excluded from our experimental comparison.

Temporal Graphs. In recent years, temporal graphs have been studied from various perspectives [58, 61–64]. One common approach extends classical concepts from static graph theory—such as PageRank [2, 19, 50], shortest paths [41, 53, 59], and centrality measures [5, 35, 50]—to the temporal domain, accompanied by efficient algorithms tailored to these adaptations. Other studies focus on gaining a deeper understanding of the structure and dynamics of temporal graphs. In particular, recent work leverages temporal motifs [8, 26, 34, 43, 46] and their frequency distributions to analyze and characterize temporal networks. These works typically focus on patterns with a fixed number of vertices or edges. To apply such algorithms for cycle discovery, one would need to execute them separately for each possible cycle length. Although feasible in theory, this approach has several practical limitations. First, the range of cycle lengths of interest is generally unknown in advance. Second, motif-based algorithms are designed to enumerate or count

all patterns of a given size, rather than cycles specifically, resulting in substantial overhead from irrelevant patterns. Moreover, many of these studies focus on frequency counting or statistical summarization instead of exact enumeration. As a result, temporal motif algorithms are not well-suited for solving the STCE problem, and we do not include them in our experimental comparison, as they address a fundamentally different task.

A.3 Example of IB-DFS Method

Figure 12 illustrates the search process of Incremental Breakpoint-based DFS starting from the newly added edge $(V_3, V_6, 8)$, with a time window $w = 10$. The algorithm incrementally explores the path $V_3 \xrightarrow{8} V_6 \xrightarrow{10} V_7 \xrightarrow{11} V_8 \xrightarrow{12} V_1$. Since no breakpoint is encountered, the search continues to the successors of V_1 , namely V_7 and V_2 . The edge $(V_1, V_7, 1)$ is pruned due to exceeding the time window ($12 - 1 > 10$), and $(V_1, V_7, 2)$ is skipped as V_7 has already been visited. The algorithm then follows $(V_1, V_2, 2)$, where a breakpoint is detected since the timestamp (2) is earlier than the current time (12), violating the temporal ordering. The exploration continues through valid edges such as $(V_2, V_3, 4)$ or $(V_2, V_3, 6)$, while ignoring $(V_2, V_3, 1)$, and eventually returns to the starting vertex V_3 .

A.4 Experimental Setup

Table 2 shows the basic statistics of the additional datasets. The SM dataset is obtained from [60], while the FR, MS, and TR graphs are from [21]. The NL graph is sourced from [29], the AM graph from [3], and the remaining datasets are collected from [24]. Overall, the number of vertices in our datasets ranges from approximately 1,000 to 65 million, and the number of edges spans from tens of thousands to over 1 billion. The temporal coverage of these datasets varies from 7 days to more than 3,000 days.

We obtain the source code of all baselines from their original authors. All methods are implemented in C++ and compiled with g++ using the -O3 optimization flag. Experiments are conducted on a Linux machine with a 2.40 GHz Intel Xeon Silver 4210R CPU and 256 GB RAM. Our evaluation focuses on in-memory processing, and disk I/O time is not included in the reported results.

A.5 Edge Offsets Cost

Table 3 reports the time cost of computing edge offsets on all evaluated datasets. Overall, the results demonstrate that the overhead of edge-offset computation is lightweight and negligible compared to the overall enumeration process.

Specifically, our edge-offset computation runs in linear time, i.e., $O(m)$, and completes efficiently even on large-scale graphs with tens of millions of edges. This confirms that the preprocessing required by our framework is simple and scalable, and does not introduce any practical bottleneck. In contrast, prior solutions such as 2SCENT rely on an expensive detection phase with a worst-case time complexity of $O(m(m + c)W)$, which becomes prohibitive on large graphs or under long time windows. As shown in our experiments, such detection phase often fails to complete within a reasonable time on large datasets, whereas our edge-offset computation consistently finishes quickly across all settings.

These results highlight a key advantage of our approach: edge offsets can be computed efficiently with minimal overhead, making our method well suited for large-scale and dynamic temporal graphs, where heavy preprocessing is impractical.

A.6 Performance on Static Graphs

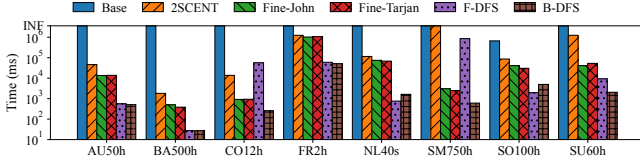


Figure 13: Comparison of running time

Figure 13 reports the running time of different algorithms on additional datasets. On smaller graphs or under narrow time windows (e.g., SO with 100h), F-DFS can outperform B-DFS due to its lower overhead. In these settings, the number of valid temporal cycles is limited, and the lightweight traversal strategy of F-DFS is sufficient to achieve good performance.

However, as either the graph size or the time window length increases, B-DFS consistently demonstrates clear advantages. By leveraging its blocking mechanism, B-DFS effectively prunes redundant and unproductive paths, preventing repeated exploration. For example, on SO with a 250h window, B-DFS significantly outperforms F-DFS. Unlike F-DFS, which may revisit the same fruitless branches multiple times, B-DFS ensures that each such branch is explored at most once, leading to substantial efficiency gains.

Importantly, even in scenarios where B-DFS does not outperform F-DFS, it still achieves performance that is markedly better than prior methods. This highlights the robustness of B-DFS: while it may incur slightly higher overhead in favorable cases for F-DFS, it maintains stable and competitive performance across a wide range of graph sizes and window lengths.

The performance trends under varying time window sizes are shown in Figure 14, which are consistent with the observations reported in Section 5.2. Overall, these results demonstrate that F-DFS may excel in lightweight scenarios, whereas B-DFS provides superior scalability and robustness for more challenging settings.

A.7 Performance on Dynamic Graphs

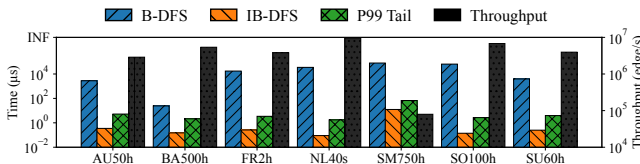


Figure 15: Comparison of running time

The overall comparison between B-DFS and IB-DFS is shown in Figure 15, with performance under different time window lengths reported in Figure 16. The results clearly demonstrate the effectiveness of IB-DFS in dynamic settings. Compared with B-DFS,

IB-DFS achieves dramatically lower latency. In nearly all cases, update queries are completed within 1 ms, highlighting the practical feasibility of IB-DFS for real-time temporal graph analysis.

As the time window length increases, the latency of IB-DFS exhibits a mild upward trend. This behavior is expected, since a larger window expands the temporal search space; nevertheless, the increase remains approximately linear and well controlled. Moreover, the reported P99 tail latency and throughput further confirm that IB-DFS maintains stable performance under varying window sizes and graph dynamics. Overall, these results indicate that IB-DFS is well suited for dynamic scenarios, efficiently supporting high-frequency graph updates while preserving low latency.

A.8 Robust Test

To further evaluate the robustness of IB-DFS, we conduct a stress test on the AM and WT datasets. In this setting, each update edge originates from one of the top-2000 vertices ranked by in-degree and targets one of the top-10 vertices by out-degree, with more than one thousand edge updates issued per second. This configuration models highly skewed and high-frequency update workloads commonly observed in real-world financial and communication networks. Figure 17 reports the results of this stress test. Even under such adversarial conditions, IB-DFS consistently maintains low latency and high throughput, significantly outperforming B-DFS. These results demonstrate that IB-DFS remains stable and efficient under extreme update rates, confirming its suitability for large-scale dynamic temporal graphs with long time windows.

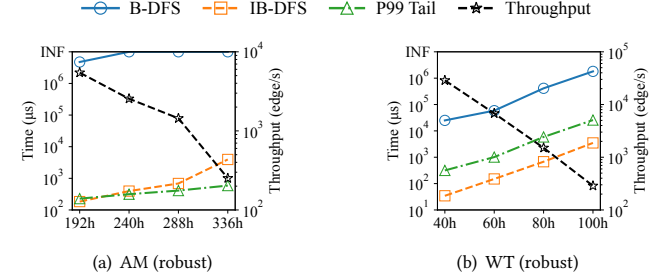


Figure 17: Robust test of IB-DFS on AM and WT

A.9 Memory Cost

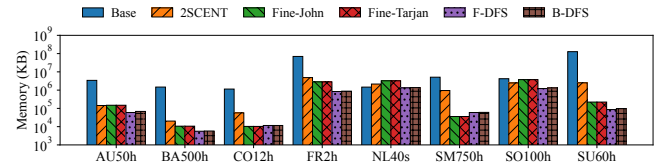
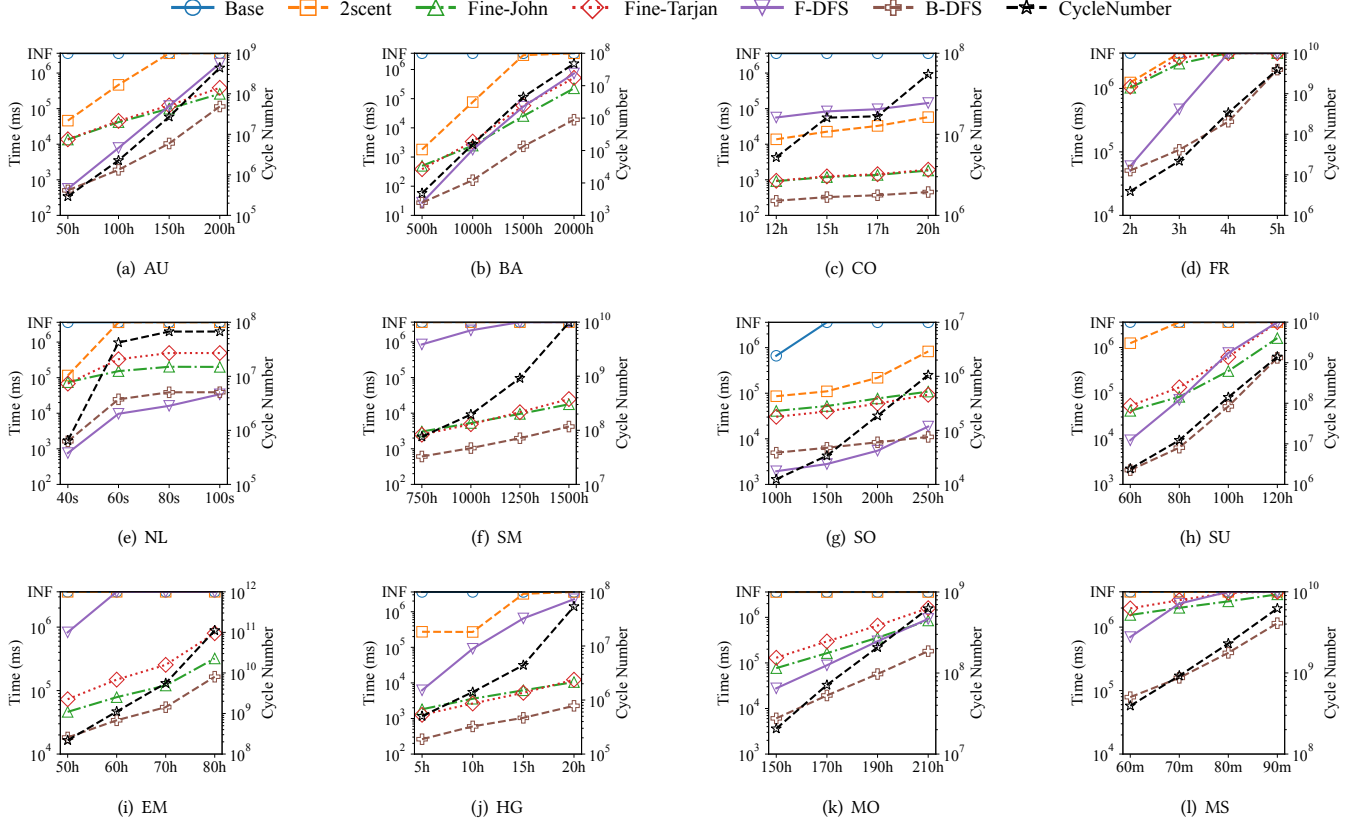


Figure 18: Comparison of memory consumption

Figure 18 reports the memory consumption of all evaluated algorithms on additional datasets. In general, Base incurs the highest memory overhead, as it needs to materialize all potential temporal paths within the given time window. In some cases, its reported

Table 3: Running time of computing the edge offsets

Dataset	HG5h	TR140h	NL40s	AU50h	BA500h	BO600h	CO12h	EM50h	MO150h	SM750h	SU60h	FR2h	WT40h	SO100h	MS3600s	AM192h
Time (ms)	28.66	25.96	809.36	37.68	1.38	1.47	2.18	11.82	16.39	21.92	58.16	587.41	323.40	1011.48	1090.26	3004.49

**Figure 14: Comparison of running time with time window w varied**

memory usage appears relatively low; however, this is misleading, since Base is terminated after one hour due to its poor runtime performance, preventing it from reaching peak memory consumption. Overall, our proposed methods exhibit stable and moderate memory usage, demonstrating favorable memory efficiency in practice.

A.10 Ablation Experiment

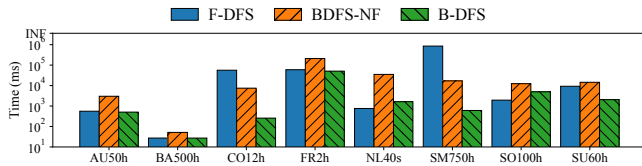
**Figure 19: Ablation experiment**

Figure 19 further compares the running time of F-DFS, BDFS-NF, and B-DFS on additional datasets. Consistent with previous observations, B-DFS consistently achieves the best performance across all

datasets. This is because B-DFS integrates both pruning strategies: edge-offset-based pruning, which eliminates redundant temporal constraint checks, and block-based pruning, which prevents repeated exploration of fruitless paths. In contrast, F-DFS applies only the former and still suffers from repeated traversal of invalid branches, while BDFS-NF relies solely on block-based pruning and incurs additional overhead due to frequent temporal checks. These results clearly demonstrate that the two pruning techniques are complementary, and their combination is crucial for achieving high efficiency in simple temporal cycle enumeration.

A.11 Case Study

Table 4 summarizes the detailed information of the transaction subgraph extracted from the IBM financial transaction dataset. All transactions are sorted chronologically by their occurrence time. Each vertex in the graph represents an account, which is uniquely identified by the combination of a bank identifier and an account identifier; the corresponding account labels are shown in parentheses in Figure 11.

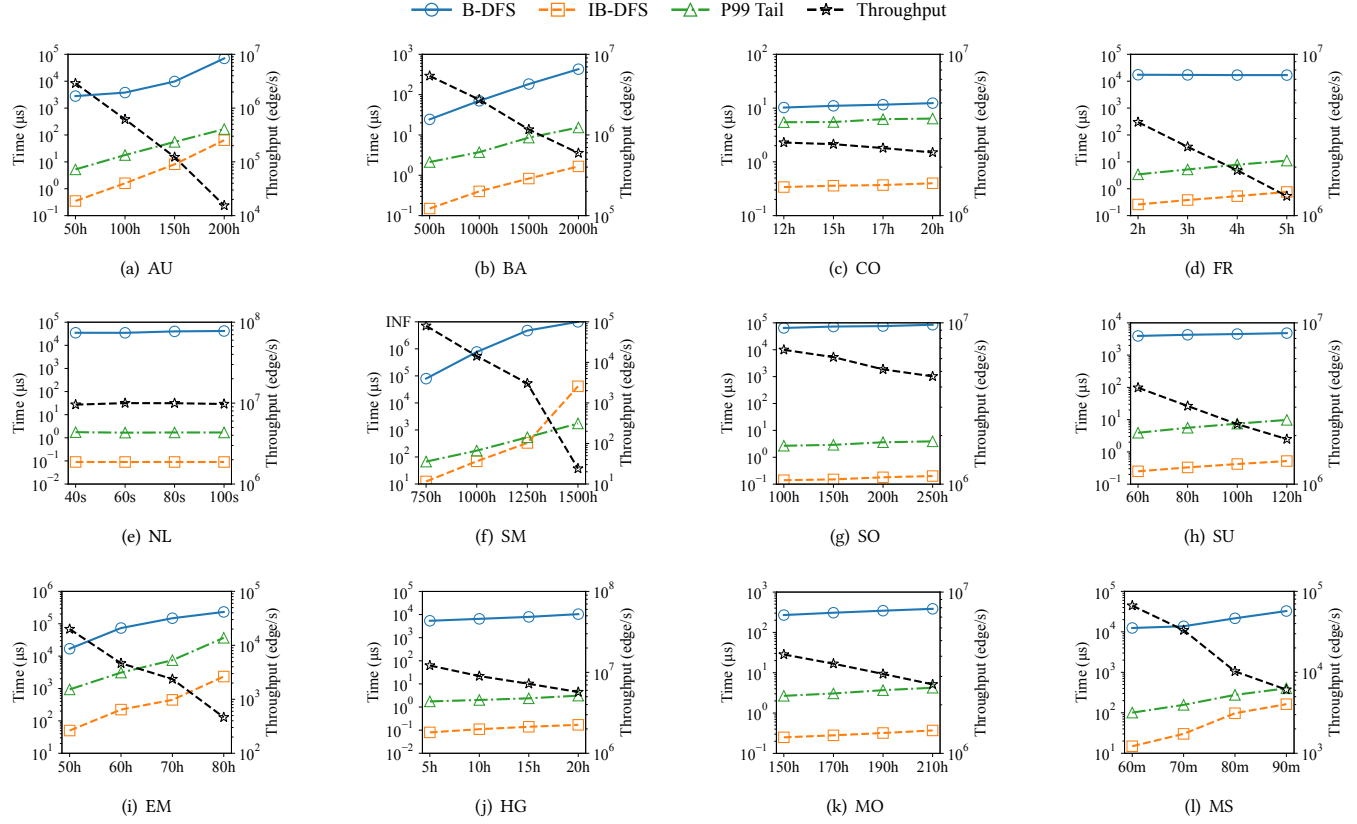
Figure 16: Performance of B-DFS and IB-DFS with time window w varied.

Table 4: The transaction information on IBM financial transaction data

Trans	Timestamp	From Bank	Account	To Bank	Account	Amount	Currency	Payment Format	Is Laundering
1	2022/09/01 04:04	15231	803A568D0 (A)	5175	8077A9EB0 (G)	2423.56	Euro	Cheque	No
2	2022/09/02 04:51	23	8102CA2C0 (C)	219449	81271E150 (I)	4908.71	Euro	Cheque	No
3	2022/09/02 13:07	15231	803A568D0 (A)	1522	8012F9500 (B)	12633.54	Euro	ACH	Yes
4	2022/09/02 14:25	29	80CF37D80 (F)	110	80D5E9570 (H)	85087.47	Brazil Real	Cash	No
5	2022/09/02 17:24	310041	8040F6160 (J)	9679	8040FAF80 (D)	43506.35	Yen	Cheque	No
6	2022/09/04 00:45	1522	8012F9500 (B)	23	8102CA2C0 (C)	11619.65	Euro	ACH	Yes
7	2022/09/04 04:05	23	8102CA2C0 (C)	9679	8040FAF80 (D)	1352779.07	Yen	ACH	Yes
8	2022/09/04 06:08	9679	8040FAF80 (D)	25960	80AC28E60 (E)	10953.15	Euro	ACH	Yes
9	2022/09/04 17:54	25960	80AC28E60 (E)	29	80CF37D80 (F)	72471.22	Brazil Real	ACH	Yes
10	2022/09/05 09:21	29	80CF37D80 (F)	15231	803A568D0 (A)	11766.38	Euro	ACH	Yes

For each transaction, the From (To) Bank and Account fields denote the source and destination accounts, respectively. Amount records the transaction value, Currency specifies the currency type, and Payment Format indicates the transaction method. The Is Laundering attribute labels whether the transaction is associated with a

laundering activity. This temporal transaction network provides a realistic case study illustrating how money laundering behaviors emerge as time-respecting cyclic flows—patterns that are difficult to detect using other motifs or cliques. Similar observations have also been reported in [16].