TIMEST: Temporal Information Motif Estimator Using Sampling Trees

Yunjie Pan
University of Michigan
Ann Arbor, Michigan, USA
panyj@umich.edu

Omkar Bhalerao
University of California, Santa Cruz
Santa Cruz, California, USA
obhalera@ucsc.edu

C. Seshadhri
University of California, Santa Cruz
Santa Cruz, California, USA
sesh@ucsc.edu

Nishil Talati
University of Michigan
Ann Arbor, Michigan, USA
talatin@umich.edu

ABSTRACT

The mining of pattern subgraphs, known as motifs, is a core task in the field of graph mining. Edges in real-world networks often have timestamps, so there is a need for *temporal motif mining*. A temporal motif is a richer structure that imposes timing constraints on the edges of the motif. Temporal motifs have used to analyze social networks, financial transactions, and biological networks.

Motif counting in temporal graphs is particularly challenging. A graph with millions of edges can have trillions of temporal motifs, since the same edge can occur with multiple timestamps. There is a combinatorial explosion of possibilities, and state-of-the-art algorithms cannot manage motifs with more than four vertices.

In this work, we present TIMEST: a general, fast, and accurate estimation algorithm to count temporal motifs of arbitrary sizes in temporal networks. Our approach introduces a temporal spanning tree sampler that leverages weighted sampling to generate substructures of target temporal motifs. This method carefully takes a subset of temporal constraints of the motif that can be jointly and efficiently sampled. TIMEST uses randomized estimation techniques to obtain accurate estimates of motif counts.

We give theoretical guarantees on the running time and approximation guarantees of TIMEST. We perform an extensive experimental evaluation and show that TIMEST is both faster *and* more accurate than previous algorithms. Our CPU implementation exhibits an average speedup of 28× over state-of-the-art GPU implementation of the exact algorithm, and 6× speedup over SOTA approximate algorithms while consistently showcasing less than 5% error in most cases. For example, TIMEST can count the number of instances of a financial fraud temporal motif in four minutes with 0.6% error, while exact methods take more than *two days*.

PVLDB Reference Format:
Yunjie Pan, Omkar Bhalerao, C. Seshadhri, and Nishil Talati. TIMEST: Temporal Information Motif Estimator Using Sampling Trees. PVLDB, 19(1): 15 - 28, 2025.
doi:10.14778/3772181.3772183

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:10.14778/3772181.3772183

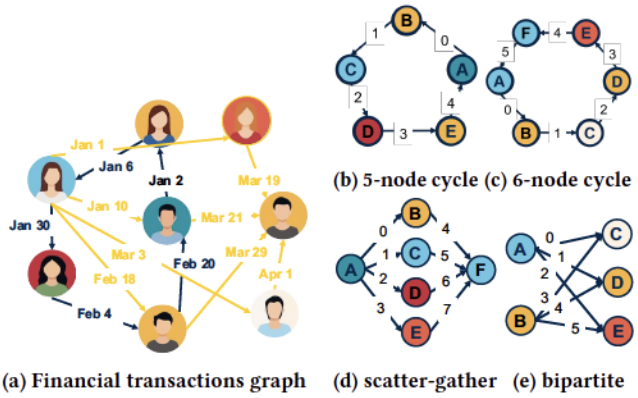


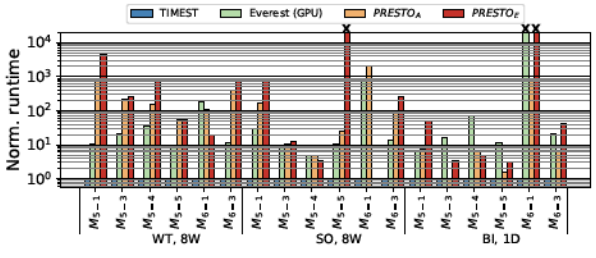
Figure 1: (a) An example of financial transactions with timestamps. Different money-laundering and gambling patterns [6, 28] including (b) 5-node simple-cycle; (c) 6-node simple-cycle; (d) scatter-gather; and (e) bipartite.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/pyjhzwh/SpanningTreeSampling/.

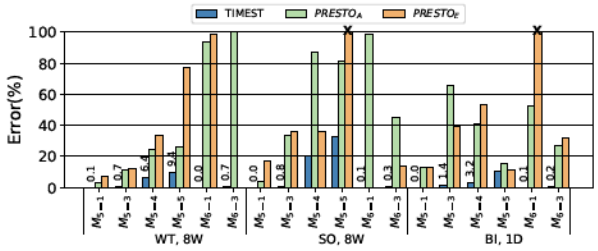
1 INTRODUCTION

A central tool in the analysis of large networks is *motif mining* [28, 33, 35, 41, 55]. A motif is a small pattern graph that indicates some special structure in a larger input graph. Motifs offer valuable insight into the graph structure, and are analogous to queries that detail a set of edge constraints. The algorithmic problem of motif mining, often called subgraph counting/finding in the algorithms literature, has a rich history in the data mining literature [5, 36, 52]. (See survey [50].)

Most real-world graphs are *temporal*, where edges come with timestamps. There is a surge of recent work on mining temporal graphs, especially searching for temporal motifs (refer to survey [14]). These works show temporal motifs can capture richer information than standard motifs [23, 38]. Temporal motifs have been used for user behavior characterization on social networks [24, 26, 41], detecting financial fraud [15, 28], characterizing function of biological networks [17, 38], and improving graph neural networks [8, 28].



(a) Runtime of prior works normalized to TIMEST. TIMEST has an average speedup of 28× compared to previous works.



(b) Relative error (%) of approximate algorithms. TIMEST is almost always the most accurate while always the fastest.

Figure 2: We show the runtime improvement and estimation errors of TIMEST versus Everest [65] (GPU) and approximate algorithms [47] ($PRESTO_A$ and $PRESTO_E$) on various datasets and motifs. Everest is tested on an NVIDIA A40 GPU, while others are evaluated on a CPU using 32 threads. "x" means run out of memory (OOM) or timeout (> 1 day).

A temporal graph can be viewed as a graph database with a rich set of attributes such as vertex/node labels and edge timestamps. Mining motifs in temporal graphs is analogous to executing queries in traditional database systems.

1.1 Problem Definition

We set some notation. The input is a multigraph $G = (V(G), E(G))$, with n vertices and m edges. Each edge in G is a tuple (u, v, t) where u and v are source and destination vertices, and t is a positive integer timestamp. Following prior work [30, 65], we assume that each tuple (u, v, t) can appear only once. However, the same u, v can have multiple edges between them with different timestamps, and there can be different tuples with the same timestamp. For a temporal edge e, let $t(e)$ to denote its timestamp. A temporal motif involves a standard graph motif with a constraint on the time window the motif occurs in, and a constraint on the order of edges. We formally define temporal motifs and temporal matches, following Paranjape *et al.* [41].

Definition 1.1. *A* temporal motif *is a triple $M = (H, \pi, \delta)$ where (i) $H = (V(H), E(H))$ is a directed pattern graph, (ii) π is an ordered set/list of the edges of H, and (iii) δ is a positive integer.*

The ordered list π specifies the time ordering of edges, and δ specifies the length of the time interval in which edges must occur.

Abusing notation, for an edge e in the list π, we use $\pi(e)$ to denote its index in the list.

Definition 1.2. *Consider an input temporal graph $G = (V(G), E(G))$ and a temporal pattern $M = (H, \pi, \delta)$. An M-*match *is a pair (ϕ_V, ϕ_E) 1-1 maps $\phi_V : V(H) \to V(G)$, $\phi_E : E(H) \to E(G)$ satisfying the following conditions.*

- *(Matching the edges) $\forall (u, v) \in E(H), \phi_E(u, v) = (\phi_V(u), \phi_V(v))$*
- *(Matching the pattern) $\forall (u, v) \in E(H), (\phi_V(u), \phi_V(v)) \in E(G)$.*
- *(Edges ordered correctly) The timestamps of the edges in the match follow the ordering π. Formally, $\forall e, e' \in E(H), \pi(e) < \pi(e')$ iff $t(\phi_E(e)) < t(\phi_E(e'))$.*
- *(Edges in time interval) All edges of the match occur within δ time range. Formally, $\forall e, e' \in E(H), |t(\phi_E(e)) - t(\phi_E(e'))| \leq \delta$.*

Figure 1a shows an example of financial transactions, represented as a graph. We also four temporal motifs, which have been explicitly mentioned as indications of money laundering [51, 55]. Note that the edges of the cycle must occur in temporal order (given the edge label), for this to represent a valid cyclic money flow. More temporal motifs are given in Figure 3. We note that a natural generalization is to impose a *partial order* of time constraints, which we discuss more in future work.

1.2 The Challenge

Temporal motif counting is particularly difficult because of *combinatorial explosion*. There can be thousands of temporal edges between the same vertices, which leads to a exponentially larger search space for motifs. For example, there are trillions of temporal 5-cliques in Bitcoin graph with 100 million edges. Exact methods based on enumeration or exploration cannot avoid this massive computation [30, 41]. Many techniques for reducing the search space using graph properties like the degeneracy and dynamic programming are tailored for simple graphs [9, 44, 49, 50]. These techniques cannot incorporate ordering constraints on edges. In general, when the motif has four vertices, no exact method is able to get results on graphs with 100M edges even in a day with commodity hardware [25, 41, 42].

There are two approaches to temporal motif counting. The usual method is to design general purpose algorithms that work for (potentially) any motif. Exact motif counting algorithms rely on explicit subgraph enumeration that suffers a massive computational explosion [30, 41, 65]. There are general purpose estimators for all kinds of temporal motifs, like IS [29], ES [59] and PRESTO [47]. However, they tend to perform poorly on larger motifs, since they rely on exact algorithm in subsampled intervals. For example, our evaluation shows that PRESTO runs for more than 5 hours estimating a 5-clique (*i.e.*, M_{5-5} in Figure 3) with a high 25% estimation error.

Other methods analyze specific motifs and designs specialized algorithms. These include 2SCENT [25] (for simple cycles), DOTT [42] (for triangles), Gao *et al.* [13] (caters to 2-, 3-node, 3-edge motifs), and TEACUPS [39] (for 4-node motifs), alongside other specialized motifs such as bi-triangle [62], bi-clique [64], and butterfly [11, 45] motifs for bipartite networks. While these often work well for the

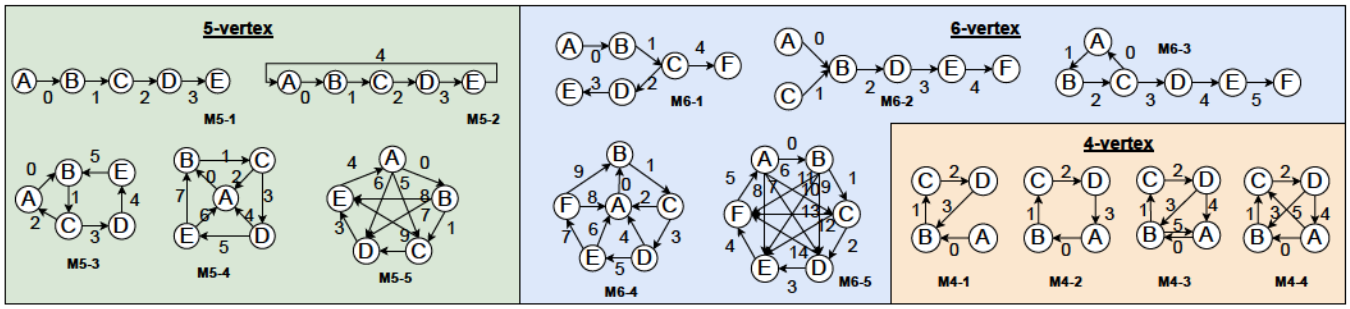


Figure 3: Connected 4-, 5- and 6-vertex temporal motifs used for evaluation.

tailored motif, they do not give general purpose algorithms. Common motifs with more than 5 nodes occurs in money laundering applications [6, 51, 55]. These motifs are shown in Figure 1, and cannot be handled efficiently by existing methods.

This leads to the main question of our paper: *do there exist general purpose temporal motif counting algorithms that can scale beyond current motif sizes?*

1.3 Our Results

Our main result is TIMEST, a general and efficient randomized (approximate) temporal motif mining algorithm. TIMEST is a general purpose method that works for any motif. As a concrete example, it is able to efficiently approximate counts for all the example motifs in Figure 3. We introduce a new technique of *temporal spanning tree sampling* that generalizes previous temporal results [40] and older non-temporal methods [21, 49, 60]. We list out some specific contributions of our work.

Temporal spanning tree sampling: Our primary contribution is a new technique of *temporal spanning tree sampling*, that generalizes a recent method of temporal path sampling [40]. To count temporal motifs of arbitrary size, we find an appropriate spanning tree of the motif. These spanning trees satisfy partial temporal constraints of this motif. We then sample many such trees, and estimate the fraction of them that "extend" to the motif.

A direct temporal spanning tree would simply take a subset of the edges *and* the corresponding temporal constraints. But it is not clear how to efficiently sample such a tree with these constraints. We show that by a careful relaxation of some of the temporal constraints, an efficient sampler can be constructed. This sampler quickly preprocesses the graph and can output any desired number of uniform random spanning trees.

Sample reduction techniques: To reduce the sample complexity, we design heuristics that choose the "best" temporal spanning tree. We can formally show that the temporal spanning tree with the fewest matches is the best choice, but finding this tree would itself require a large number of motif counts. We design heuristics to narrow down on the best spanning tree, which leads to lower error with the same sampling budget.

Practical efficiency of TIMEST: We perform a comprehensive evaluation across various datasets (Table 2) and a large collection of complex motifs (Figure 3), up to 6 vertices. We compare with a state-of-the-art exact algorithm that uses GPUs (Everest [65]) and versions of the best randomized method (Presto [47]). Results are in Figure 2a. TIMEST is *always* faster, with a typical 10x-100x speedup over both existing methods. We note that TIMEST is implemented on a CPU, and still has an average speedup of 28× over high-end NVIDIA GPU implementations. We do not give results for the motifs M_{6-4} and M_{6-5} since previous methods timeout even after running for a day. TIMEST is able to get approximate counts even for this motifs. On the specific money-laundering bipartite motif in Figure 1, TIMEST takes *4 minutes* on an example dataset, while the best exact method takes *two days*.

Low error of TIMEST: Despite the variety of complex constraints in Figure 3, in *all* cases, TIMEST has significantly lower error. We plot the errors of previous work and TIMEST in Figure 2b. TIMEST typically has an error of less than 5%, while previous methods have 20% error or more. The only hard motifs for TIMEST are M_{5-4} and M_{5-5}, which have more than 20% error. But other methods have twice as much (or much more) error.

Theoretical analysis of TIMEST: We give a comprehensive theoretical analysis of TIMEST, giving bounds on all running time steps, and an analysis of the sample complexity with respect to output error.

2 HIGH-LEVEL IDEAS IN TIMEST

The final TIMEST algorithm is fairly complex with multiple moving parts. In this section, we give an overview of various procedures in TIMEST. We will denote the input graph as G and the motif as M. A depiction of the TIMEST pipeline is given in Figure 4. There are two primary steps.

Preprocessing: This is the main novelty in TIMEST. We first choose an appropriate spanning tree of M and identify of a set of *relaxed temporal constraints*. This process itself requires computations involving the input graph G. Our aim is to select a temporal "submotif" T of M that can be quickly sampled. We set up the temporal constraints to allow for a dynamic programming bottom-up approach to sample uniform random instances of T. This approach requires computing a series of edge weights, called the *sampling weights*. Since there are multiple temporal edges between the same pair of vertices, the edge weights can be complicated to compute.

Sampling: These weights allow us to apply a common paradigm in approximate motif counting that involves sampling a substructure, and extending to the motif [9, 21, 40, 49, 60]. We show how the weights can be used to subsample subtrees of T, which can then be incrementally extended to a uniform random sample of T. A quick validation verifies that the sample satisfies the temporal constraints

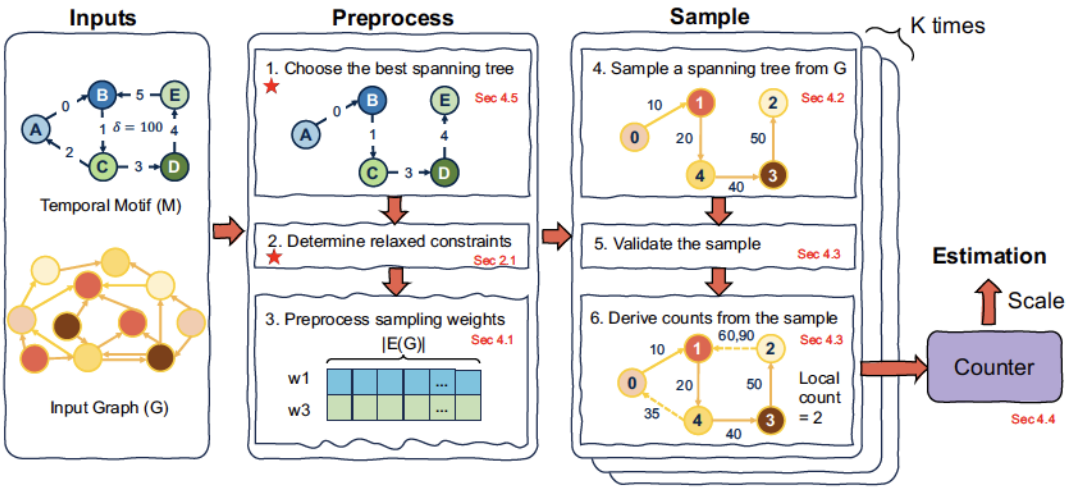
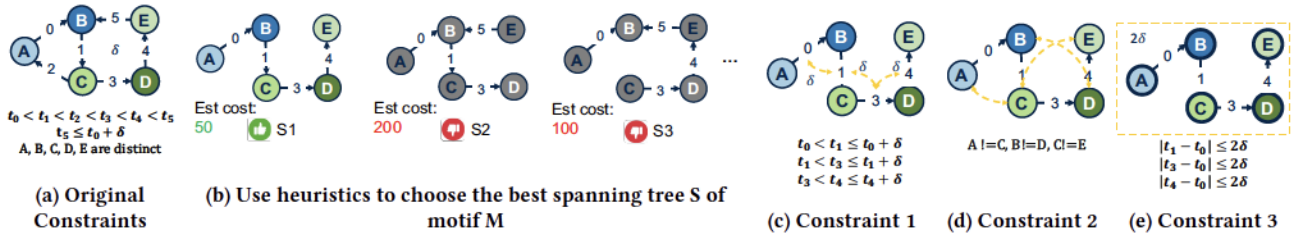


Figure 4: Overall process of estimating temporal motif M counts in the input graph G.



(a) Original Constraints | **(b) Use heuristics to choose the best spanning tree S of motif M** | **(c) Constraint 1** | **(d) Constraint 2** | **(e) Constraint 3**

Figure 5: An example of mining motif $M_{5\text{-}3}$. (a) The standard approach strictly enforces all constraints for a temporal motif match. (b) Multiple spanning trees of the motif exist, and we use a heuristic to select the optimal one that has the lowest estimated sampling cost (Est cost). We then relax specific constraints and apply three partial constraints to the spanning tree: (c) enforcing edge order and δ-window constraints for adjacent edges; (d) ensuring distinct end vertices for adjacent edges; and (e) constraining all edges to fall within a 2δ time window.

of M. The next step is to count the number of matches to M that involve the sampled match to T. Again, temporal graphs bring in challenges. In the typical setting, a single T can only extend to a small number of matches to M. Here, the sampled T could have many temporal edges going between the vertices, and hence an unbounded number of matches to M come from a *single* sampled T. We devise an algorithm using a careful binary searching over various edge lists to compute this number of matches. The sampling step is repeated for a collection of samples. Standard randomized algorithms techniques tell us how to rescale the total number of matches found, to get an estimate for the total count of M.

We go deeper into the preprocessing step, which requires many new ideas.

2.1 A Closer Look at Preprocessing

Consider mining the temporal motif $M_{5\text{-}3}$ as shown in Figure 5. We fix the specific spanning tree shown in the figure. The original motif imposes an ordering constraint on the edges and a total time window constraint. It is not clear how to sample such a spanning tree, so we relax some of the constraints.

We relax the requirements for all edges to be ordered correctly and to fall within a specified time interval from Definition 1.2. These constraints are only applied to certain pairs of *adjacent edges*. This forms the base constraint (Constraint 1), as illustrated in Figure 5c. In complex spanning trees, these constraints leave out multiple ordering conditions.

We note that most efficient mechanisms to sample trees use dynamic programming, and technically sample tree homomorphisms (where non-adjacent pattern vertices can be matched to the same vertex in G). To address this, we introduce a straightforward constraint ensuring that the end vertices of adjacent edges are distinct. This approach minimizes memory overhead and significantly reduces invalid samples, making it particularly effective in skewed graphs. This is described as Constraint 2 is illustrated in Figure 5d.

Constraints 1 and 2 focus on adjacent edges, while Constraint 3 introduces a global constraint of the time interval. For example, in the chosen spanning tree of $M_{5\text{-}3}$, the sequential timing conditions of Constraint 1 implies $t_0 < t_4 \le t_0 + 3\delta$, potentially violating the δ time interval constraint. (We use t_i to denote the timestamp the edge labeled i.) To reduce the affect of these violations, we apply a

2δ sliding window approach, partitioning the input graph into overlapping subgraphs based on timestamps. This results in intervals like $[0, 2\delta]$, $[\delta, 3\delta]$, and so on, ensuring that any temporal motif matches within a δ-time window must reside entirely within a single interval and will not span across the boundary of two intervals. This approach effectively reduces the time window length to 2δ, significantly reducing the occurrence of invalid matches due to δ-window violations. An illustration of Constraint 3 is provided in Figure 5e. Note that this approach leads to the under/over counting of some temporal motifs due to boundary conditions. Since this overcounting factor can be determined for any match, we can apply corrections to the final estimate.

All in all, these constraints create a subset of matches for the spanning tree. We prove that by a multi-pass preprocessing over the graph, we can construct an efficient sampler for these matches of the spanning tree. This process involves a dynamic program that constructs weights for various subtrees, and then computes weights for larger subtrees. The constraints discussed above allow for this dynamic programmin approach to work.

Choose a Spanning Tree: The first step of preprocessing requires selecting the best spanning tree. As shown in Figure 5b, a motif can have multiple valid spanning tree candidates. We try to determine the spanning tree that will lead to the fewest number of samples required for convergence. This is an important distinction from previous spanning tree based sampling works that fix a choice [21, 40, 49, 60]. Those methods could afford to do that because they deal with smaller and non-temporal motifs. We need to design some heuristics to select a smaller pool of spanning trees, and then exactly determine the sampling cost for these trees. By selecting the tree with the lowest sampling cost, we ensure efficient and accurate motif counting during execution. The details are elaborated in Section 4.5.

3 RELATED WORK

Static Motif Mining. There have been a lot of research in static motif mining, including exact counting and enumeration approaches [12, 19, 20, 31, 32, 53, 56, 61], and approximate algorithms [3, 10, 18, 21, 21, 29, 43, 47–49, 57–60, 62–64] Although static motif mining can serve as a first step for temporal motif matching, as illustrated by Paranjape *et al.* [41], this often results in orders of magnitude redundant work [30, 65] due to the presence of temporal constraints.

Exact Temporal Motif Mining. Temporal motif mining was first formally introduced by Paranjape *et al.* [41], who suggested an algorithm to list and count instances of temporal motifs. A newer exact counting method [30] introduced a backtracking algorithm by listing all instances on edges sorted chronologically. Everest [65] further refines this approach, employing system-level optimizations on GPUs to enhance performance substantially. There are also numerous algorithms designed for general temporal motif counting [34, 54] and others tailored to specific motifs [7, 13, 25, 42]. Despite these advancements, scalability remains a significant issue.

Approximate Temporal Motif Mining. To approximate temporal motif counts, various sampling-based methods balance efficiency and accuracy. There are several general frameworks for estimating temporal motifs. Our work differs fundamentally from those methods such as IS [29], PRESTO [47], and ES [59] in how it handles motif structure and temporal constraints. IS [29] partitions the timeline into non-overlapping intervals and performs exact motif counting on a sampled subset of intervals. Its performance is highly sensitive to interval size and may miss cross-interval patterns. PRESTO [47] improves upon IS by avoiding rigid partitioning and instead using uniform sampling across the entire graph. While more flexible, it still treats motif instances in a uniform sampling space and does not incorporate structural information. ES [59] samples individual edges uniformly and enumerates all local motif instances around them. It does not exploit the motif's topology beyond local neighborhoods, leading to inefficiencies and high variance for larger motifs. In contrast, TIMEST introduces a spanning tree–based sampling framework that leverages the motif's structural backbone and performs weighted sampling along constrained subspaces.

There are methods that rely on an exact counting for some sampled graph, such as Ahmed *et al.* [4] and OdeN [46]. These are often less efficient and have large estimator variance. There are methods tailored to the unique characteristics of specific motifs. Cai *et al.* [11] and Pu *et al.* [45] specialize in butterfly motifs in bipartite graphs, Oettershagen *et al.* [37] explore 2- and 3-node motifs and Pan *et al.* [40] focus on 4-node motifs.

Table 1: Summary of notations

Symbol	Definition
δ	maximum time window
$M = (H, \pi, \delta)$	temporal motif
G	input graph with $m = \lvert E(G)\rvert$ edges and $n = \lvert V(G)\rvert$ vertices
ϕ_V, ϕ_E	vertex map $\phi_V : V(H) \to V(G)$ and edge map $\phi_E : E(H) \to E(G)$
$t(e)$	time stamp of edge e
S	spanning tree of a temporal motif
P	sampled edges from G that is mapped to S
$w_{s,e}$	sampling weight of edge e in G that is mapped to s in S
W	total sampling weights
$C, \hat{C}$	ground truth and estimated motif count

4 THE ALGORITHMIC DETAILS OF TIMEST

In line with previous work [30, 65], we store the graph G as a sorted list of incoming and outgoing edges by timestamp for efficient binary search operations. For ease of algorithm explanation, we adjust the timestamps of the edges in G to begin at time 0. We summarize the important symbols in Table 1.

We start with some important definitions for temporal graphs.

Definition 4.1 (Temporal outlists and degrees). *For a vertex v and timestamp t and t', the* temporal outlist $\Lambda_v^+[t, t']$ *is the set of outedges (v, w, t''), where w is any vertex and $t'' \in [t, t']$. The* temporal inlist $\Lambda_v^-[t, t']$ *is defined similarly.*

With $\alpha \in \{+, -\}, \beta \in \{<, >\}$ and fixed δ, we use the notation $\Lambda_v^\alpha[t, (\beta, \delta)]$ to represent $\Lambda_v^\alpha[t - \delta, t]$ when β is $<$, and $\Lambda_v^\alpha[t, t + \delta]$ when β is $>$.

The temporal out-degree $d_v^+[t, t'] = |\Lambda_v^+[t, t']|$. *And the* temporal in-degree $d_v^-[t, t'] = |\Lambda_v^-[t, t']|$.

Definition 4.2 (Multi-edge list). *We define multi-edge list $El_{u,v}[t, t']$ as the collection of edges $e = (u, v, t'')$ with $t'' \in [t, t']$.*

To simplify the notation with $\alpha \in \{+, -\}, \beta \in \{<, >\}$ and δ, we use $El_{u,v}^\alpha[t, (\beta, \delta)]]$ to present different cases. Here, α represents the edge direction: $+$ means $u \to v$ direction, and $-$ means $v \to u$ direction. β specifies the time range: the timestamp must fall in $[t - \delta, t]$ $(<)$ or fall in $[t, t + \delta]$ $(>)$.

Definition 4.3 (Multiplicity). *Given a pair u, v of distinct vertices and timestamps $t < t'$, the* multiplicity $\sigma_{u,v}[t, t']$ *is the number of edges (u, v, t'') which satisfy $t'' \in [t, t']$. $\sigma_{u,v}[t, t'] = |El_{u,v}[t, t']|$. We denote the maximum δ-multiplicity as σ_δ, defined as $\max_{u,v,t} \sigma_{u,v}[t, t + \delta]$.*

Let S be a spanning tree of the motif we wish to count. We first define the notion of a leaf edge. An edge $s \in S$ is called a leaf edge if any one of its endpoints has 0 indegree and outdegree.

We introduce the process of assigning weights to edges S. It will be convenient to root the tree at an edge (rather than a vertex), and we define parent/children using this rooting. Note that this rooting is independent of the actual directions of the edges. The height of a leaf edge is set to zero. The height of an edge is the length of the longest path down the tree to a leaf. (Equivalently, the height of an edge is one plus the maximum height of its children.)

Definition 4.4. *The dependency list of an edge $s \in E(S)$ consists of triples $\langle s', \alpha, \beta \rangle$, where $s' \neq s$ is a child of s, the parameter α (+ or -) determines the direction (incoming or outgoing) of s' with respect to the vertex at which they meet and β (<, >) gives the relative time-order between s' and s. We denote this list by $D(s)$.*

Consider the tree S in Figure 4, where the edges s_i is the edge labeled i. Then, $D(s_1) = \{\langle s_0, +, < \rangle\}$.

Definition 4.5. *A spanning tree of a temporal motif M is represented as a tuple $S = (L, O, D)$, where (i) L is the list of edges in $E(H)$ that form the directed spanning tree of M; (ii) O is an ordering of the edges in L; (iii) D is the dependency of the edges in L.*

The order O can be any topological sorting based on the dependency D. Now we introduce the notion of a *partial match* to a tree in G, which will be central throughout the paper.

Definition 4.6. *A subgraph H of G is a δ-partial match to tree S if there exists a pair $\phi = (\phi_V, \phi_E)$ of mappings such that $\phi_V : V(S) \to V(G)$ and $\phi_E : E(S) \to E(G)$ with the following properties:*

1. *$\phi_E(u_i, u_j) = (\phi_V(u_i), \phi_V(u_j))$*
2. *$(u_i, u_j) \in E(S)$ if and only if $(\phi_V(u_i), \phi_V(u_j)) \in E(G)$, with $\phi_V(u) \neq \phi_V(v)$*
3. *Suppose $e_1 = (u, v)$ and $e_2 = (x, y)$ are dependent edges in S. Assume that the height of $e_1 >$ height of e_2. Then, if $t(u, v) < t(x, y)$, then $t(\phi_E(x, y)) \in [t(\phi_E(u, v)), t(\phi_E(u, v)) + \delta]$. Otherwise, $t(\phi_E(x, y)) \in [t(\phi_E(u, v)) - \delta, t(\phi_E(u, v))]$.*
4. *Further $|\phi_E(u, v) \cap \phi_E(x, y)| = 1$*

In particular, a partial match ϕ is a homomorphism (property (2)) of S into G, which respects the relative time order between pairs of adjacent edges along every path from the root of S to any of its leaves (property (3), corresponds to Constraint 1). It also ensures that matches to adjacent edges intersect only in one vertex (property (4), corresponds to Constraint 2). Given a partial match ϕ, an edge $e = (u, v, t) \in E(G)$, and an edge $s = (x, y) \in E(S)$, we say that ϕ matches s to e if $\phi_E(x, y) = (u, v)$. We will often refer to e as the *match* of s in G under ϕ.

Overview: Figure 4 illustrates the overall process of estimating temporal motif counts for a given motif M in an input temporal graph G. The algorithm begins by selecting the best spanning tree of motif M to optimize sampling efficiency (Section 4.5), then determines relaxed temporal constraints (Section 2.1) and precomputes edge sampling weights (Section 4.1). In the sampling phase, it repeatedly samples spanning trees from G (Section 4.2), validates them against relaxed constraints, and derives local counts using dynamic programming (Section 4.3). The final estimate is obtained by scaling the aggregated local counts from K samples (Section 4.4). We explain the spanning tree selection at the end, since it requires many concepts from the other components.

4.1 Preprocess Sampling Weights

During the preprocessing phase, for each edge of the input graph, we compute a set of $|E(S)|$ non-negative weights, one for every one of the $|E(S)|$ edges in the spanning tree S. We also refer to the root edge of S as the center edge.

Furthermore, for an edge $s \in S$, v_s denotes the lower-level endpoint of the edge s and T_s to denote the subtree rooted at v_s.

Definition 4.7 (s-weight of edge e $w_{s,e}$). *Given an edge $e = (u, v, t) \in E(G)$ and an edge $s \in E(S)$, the s-weight of e is defined as the number of partial matches (in G) ϕ to the subtree T_s which are rooted at $\phi_V(v_s)$, where v_s the end-point of s with lower height. We will denote the s-weight of an edge e by $w_{s,e}$.*

Consider S in Figure 4. Suppose $e = (v_2, v_3, 40) \in E(G)$ is a match to $s_1 = (B, C)$. Then $\phi_v(B) = v_2$, $\phi_V(C) = v_3$. In this case, w_{e,s_1} is the number of partial matches in G to the subtree T_B that are rooted at v_2. Since the subtree T_B consists of a single edge (A, B) that happens before (B, C), w_{e,s_1} assume a value equal to the number the inedges of v_2 with timestamps in the interval $[40 - \delta, 40]$.

More generally, suppose $e = (u, v, t) \in E(G)$ is a match to the edge $s = (x, y)$ under ϕ_E. Consider an arbitrary triple $\langle s_1, \alpha_1, \beta_1 \rangle$ in the list $D(s)$. Assume w.l.o.g. $\phi_V(v_s) = u$. We claim that any match to s_1 in G under ϕ_E will either be an in-edge or out-edge of u (depending on α_1), that does not intersect v, and whose timestamp is either in $[t - \delta, t]$ or $[t, t + \delta]$, depending on β_1. We denote this list by L_{e,s,s_1}, which is formally defined below.

Claim 4.8. *The partial match ϕ_E will map s_1 to an edge in $\Lambda_u^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{uv}[t, (\beta_1, \delta)]$ if $\phi_V(v_s) = u$.*
The partial match ϕ_E will map s_1 to an edge in $\Lambda_v^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{vu}[t, (\beta_1, \delta)]$ if $\phi_V(v_s) = v$.

Thus, when looking for the potential matches to s_1, it suffices to restrict to the edge list given in Claim 4.8. Let us denote this list $L_{e,s,s_1} = \Lambda_u^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{uv}[t, (\beta_1, \delta)]$ (when $\phi_V(v_s) = u$) or $\Lambda_v^{\alpha_1}[t, (\beta_1, \delta)] \setminus El_{vu}[t, (\beta_1, \delta)]$ (when $\phi_V(v_s) = v$) for simplicity.

Suppose the list $D(s)$ contains k triples $\langle s_1, \alpha_1, \beta_1 \rangle, \cdots, \langle s_k, \alpha_k, \beta_k \rangle$. Then any partial match ϕ to the subtree T_{v_s} can be obtained by first picking a match to each of the k edges $s_1, s_2, \ldots, s_k$, and then extend them recursively into partial matches to the subtrees $T_{s_1}, \cdots T_{s_k}$.

Claim 4.9. *Suppose the list $D(s)$ contains triples $< s_i, \alpha_i, \beta_i >$ for $1 \leq i \leq k$. Then*

$$w_{s,e} = \Big(\sum_{e \in L_{e,s,s_1}} w_{s_1,e}\Big)\Big(\sum_{e \in L_{e,s,s_1}} w_{s_2,e}\Big) \ldots \Big(\sum_{e \in L_{e,s,s_k}} w_{s_k,e}\Big)$$

Proof. Let Match$(e = (u, v, t), s)$ denote the collection of partial matches to the subtree T_s, which are rooted at $\phi_V(v_s)$ in G. Here $\phi_v(v_s) \in \{u, v\}$. Then Match$(e, s) = \{(\phi_1, \ldots, \phi_k)\}$, where ϕ_i is a partial match to the subtree T_{s_i} in G. Every such match is rooted at a suitable end-point of some edge $e_i \in L_{e,s,s_i}$. This endpoint

Algorithm 1 : PREPROCESS(S)

Input: Spanning tree $S = (L, O, D)$
Output: Total sampling weight W; W_i for each subgraph G_i; $w_{i,s,e}$ for e in G_i that are mapped to s.

1: num_{subg} = time span of G / δ
2: **for** $i \in [0, num_{\text{subg}})$ **do** ▷ partition to subgraphs
3: build subgraph G_i whose timestamp in $[i\delta, (i+2)\delta]$
4: $W_i, w_{i,s,e}$ = PREPROCESSSUBGRAPH(S, G_i)
5: **return** $W, W_i, w_{i,s,e}$

Algorithm 2 : PREPROCESSSUBGRAPH(S, G_i)

Input: Spanning tree $S = (L, O, D)$, and current subgraph G_i
Output: W for the current subgraph G_i; $w_{s,e}$ for edge $e \in G_i$ mapped to edge $s \in L$.

1: **for** $h \in [1, H]$ **do** ▷ skip leaf edges in S
2: **for** $e = (u, v, t) \in E(G_i)$ **do**
3: **for** $s \in O[h]$ **do**
4: calculate $w_{s,e}$ according to Claim 4.9
5: **if** s is the "center" edge in S **then**
6: W += $w_{s,e}$
7: **return** $W, w_{s,e}$

is given by $\phi_{Vi}(v_{s_i})$. Note that the s weight of e is precisely the size of the set Match(e, s) i.e. $w_{s,e} = |\text{Match}(e, s)|$. Recall that the number of partial matches to the subtree T_{s_i}, which can be obtained from an edge e in the list L_{e,s,s_i}, is exactly equal to e's s_i weight. Hence the number of partial matches ϕ_i with the given property is $\sum_{e \in L_{e,s,s_i}} w_{s_i,e}$.

As a result, we can conclude that the number of tuples in the set Match(e, s) is given by $\Big(\sum_{e \in L_{e,s,s_1}} w_{s_1,e}\Big)\Big(\sum_{e \in L_{e,s,s_1}} w_{s_2,e}\Big) \dots \Big(\sum_{e \in L_{e,s,s_k}} w_{s_k,e}\Big)$. □

Claim 4.9 allows us to use dynamic programming to efficiently compute the $w_{s,e}$ values, where the choices for s vary in the increasing order of their height in S.

Finally, let W denote the total number of partial-matches to S in G and c be the center edge of S. Then,

CLAIM 4.10. *$W = \sum_{e \in E(G)} w_{c,e}$, where c is the center edge in S*

PROOF. Since any edge in G could potentially be the center edge in a match to S, it follows that the total number of matches to S in G is $\sum_{e \in E(G)} w_{c,e}$. This completes the proofs. □

We detail the PREPROCESS procedure in Algorithm 1. First, we segment the entire timespan of the input graph G into overlapping intervals of length 2δ. Let $I = \{[0, 2\delta], [\delta, 3\delta], \dots, [(q-2)\delta, q\delta]\}$ denote the resulting collection of intervals. Here $q = T/\delta$ where T is the time span of G. Each interval I_i corresponds to a subgraph G_i, consisting of the edges in G which fall within the interval I_i. Next, we invoke the subroutine PREPROCESSSUBGRAPH to get the overall sampling weight W_i for each subgraph G_i, as well as the s-weights for the edges in each subgraph. Here s varies across the edges of S.

Algorithm 3 : SAMPLESUBG$(S, G_i, W, w_{s,e})$

Input: Spanning tree $S = (L, O, D)$, sampling weights W for current subgraph G_i, and $w_{s,e}$ for edges in $E(G_i)$
Output: Sampled edges P where $P[i]$ is the edge in $E(G_i)$ that is mapped to s_i in S, partial match vertex map ϕ_V and edge map ϕ_E.

1: c is the center edge in S
2: Sample center edge e with sampling probability $p_e = w_{c,e}/W_i$
3: $P[c] = e$, Update ϕ_V and ϕ_E
4: **for** $h \in [H, 0]$ **do**
5: **for** $s \in O[H]$ **do**
6: **for** $\langle s', \alpha', \beta' \rangle \in D(s)$ **do**
7: Use binary search to find the $L_{e,s,s'}$
8: $W_x = \sum_{e \in L_{e,s,s'}} w_{s',e}$
9: Compute $p_{s',e} = w_{s',e}/W_x$ for $e \in L_{e,s,s'}$
10: Sample $e_{s'}$ using sampling weight $p_{s',e}$
11: $P[s'] = e_{s'}$, update ϕ_V and ϕ_E
12: **return** P, ϕ_V and ϕ_E

In the PREPROCESSSUBGRAPH procedure (in Algorithm 2), we compute the s-weight for every edge $e \in E(G)$ and every $s \in E(S)$. These weights are stored in a 2D array and are computed in the order given by O. More concretely, for every $e \in E(G)$, first we compute its s-weight for every $s \in E(S)$ which is at *height 1* in S. (Leaf edges with 0 height are skipped because the weights are constant 1s.) Next, for each edge, we compute its associated s-weight for every $s \in E(S)$ which is at *height 2*. We use the relation given by Claim 4.9 to do so. We repeat these steps until we reach the center edge in S. The total sampling weight for the current subgraph is then calculated by Claim 4.10.

4.2 Sample

The goal of the SAMPLE procedure is to sample a partial match to spanning tree S from graph G. First, we sample an interval from I with probability proportional to its overall weight W_i. Next, we construct the graph G_i by including those edges of G whose timestamp belongs to the sampled interval I_i.

Then we invoke the SAMPLESUBG function on G_i (Algorithm 3). In this subroutine, first we sample a match $e = (u, v, t) \in E(G_i)$ to the center edge c of S. This is done by sampling an edge $e \in E(G)$ with probability proportional to $w_{c,e}$ (line 2). Next, for every $s \in D(c)$, we use binary search to get the list of potential matches to s in G. For each $s \in D(c)$, we sample its match in G by picking an edge from the list of candidate matches to s with probability proportional their associated s-weight. Having done this for every $s \in D(c)$, we recursively build a match to S, now by sampling matches to the edges in $D(s)$ for every $s \in D(c)$. Note that the order in which we sample a match to the spanning tree is exactly the opposite of how we compute the edge weights.

Let $\mathcal{G} := \{G_1, G_2, \dots, G_{|I|}\}$ be the family of subgraphs of G obtained from the intervals in I. For a fixed $G_k \in \mathcal{G}$ containing ϕ, our algorithm outputs a uniform random partial match to S.

LEMMA 4.11. *Fix a partial match ϕ to S in G. If this partial match is present in N_ϕ-many subgraphs of $\mathcal{G}$, then the subroutine SAMPLESUBG will output ϕ with probability N_ϕ/W.*

Algorithm 4 : ValidateAndDeriveCnt(M, P, ϕ_E, ϕ_V)

Input: Temporal motif M, sampled spanning tree P, partial vertex map ϕ_V and edge map ϕ_E
Output: Number of motif instances *cnt* that extend from $\phi_E(S)$

1: Verify if ϕ_V is a 1-1 map. If not, **return** 0
2: Verify if edges in $\phi_E(S)$ are in δ time range. If not, **return** 0
3: Verify if edges in $\phi_E(S)$ have the correct edge order. If not, **return** 0
4: **return** DeriveCnt$(M, P, \phi_E, \phi_V)/N_\phi$

Algorithm 5 : DeriveCnt$(M, P, \phi_E, \phi_V))$

Input: Temporal motif M, sampled spanning tree P, partial vertex map ϕ_V and edge map ϕ_E
Output: Number of motif instances *cnt* that extend from $\phi_E(S)$

1: For every edge e of M that's not in the sampled spanning tree P, use binary search to find the lists (in G) of potential matches
2: Number these lists as $L_1, L_2, \cdots L_l$ in the time ordering.
3: Use the ListCount algorithms (Alg. 4 of [39]) to count all possible combinations without enumeration.
4: **return** cnts

Note that $N_\phi \leq 2$ for every partial match ϕ to S. For a fixed $G_k \in \mathcal{G}$ containing ϕ, our algorithm outputs a uniform random partial match to S.

4.3 Validate and Derive the Motif Counts

After sampling a partial match ϕ to S, we verify if it respects all ordering and temporal constraints. If the sampled match fails to meet any required conditions, we set its count to 0. Otherwise, we obtain the number of instances of the target motif induced by it. This can be done by a combination of the merge-technique used in the merge-sort algorithm and dynamic programming DeriveCnt (Algorithm 5), which uses the ListCount algorithm (Alg. 4 from Pan *et al.* [39]). Given a collection of time ordered edge lists $L_1, L_2, \cdots L_l$, it counts the number of combinations $(e_1, e_2, \ldots, e_l)$ where for all i, $e_i \in L_i$, and $t(e_i) < t(e_{i+1})$. This is done with pointer traversals and dynamic programming. Note that by modifying the DeriveCnt function to list the sampled motif instance to reconstruct individual motif instances, our algorithm can be adapted to support random motif listings. Finally, we rescale this count to ensure that our estimate is unbiased at the end. The full procedure is detailed in Algorithm 4.

4.4 Overall Estimate Procedure

Integrating the procedures described previously, we outline the Estimate process in Algorithm 6. First, analyze the motif and use heuristics to choose a spanning tree S of the motif. Given S, for every $s \in E(S)$ and $e \in E(G)$, we compute the weight $w_{s,e}$, as given by the Preprocess subroutine. Next, we invoke the SampleSubg procedure to sample a partial match to S and feed it as input to the ValidateAndDeriveCnt subroutine, which in turn returns the number of matches to the target motif induced by the partial match to S. We run the procedure multiple times, average the individual counts, rescale the average and return the final result.

Algorithm 6 Estimate(M, G, k)

Input: Temporal motif M, input graph G, and sample number k
Output: Motif count estimate $\widehat{C}$

1: Use heuristic to choose a spanning tree S
2: $W, W_i, w_{i,s,e}$ = Preprocess(S)
3: num_{subg} = time span of G / δ
4: init sample counts $samp_i = 0$ for every subgraph G_i, $cnt = 0$
5: **for** $j \in [1, k]$ **do** ▷ sample subgraphs
6: Sample G_i with probability $p_{i,\delta} = W_i/W$
7: $samp_i$ += 1
8: **for** $i \in [1, num_{\text{subg}}]$ **do**
9: **for** $j \in [1, samp_i]$ **do** ▷ sample edge in subgraphs
10: P, ϕ_E, ϕ_V = SampleSubg $(S, G_i, W_i, w_{i,s,e})$
11: cnt += ValidateAndDeriveCnt(M, P, ϕ_E, ϕ_V)
12: **return** $\widehat{C} = (cnt/k) \cdot W$

Next, we show that the estimate produced by our algorithm is unbiased and bounded.

Lemma 4.12. *Let $\widehat{C}$ denote the estimate produced by TIMEST and C be the true motif count. Then* $\mathbf{E}[\widehat{C}] = C$.

Proof. Let $\mathcal{S}$ denote the set of partial matches to the spanning tree S in G. For every $1 \leq i \leq k$, let ϕ_i denote the partial match to S output by the subroutine SampleSubg when invoked for the i-th time. For every $\phi \in \mathcal{S}$, let M_ϕ be the number of instances of the target motif H that extend from ϕ, N_ϕ be the number of subgraphs in $\mathcal{G}$ which contain ϕ and C be the total number of occurrences of H in G. Further, let the random variable Y_i denote the outcome of DeriveCnt when invoked on ϕ_i and let $Y = \sum_{i \leq k} Y_i$.

$\mathbf{E}[Y] = \mathbf{E}[\sum_{i \leq k} Y_i] = \sum_{i \leq k} \mathbf{E}[Y_i]$.

Note that $\mathbf{E}[Y_i] = \sum_{\phi \in \mathcal{S}} \mathbf{E}[Y_i | \phi_i = \phi] \Pr(\phi_i = \phi)$.

From Lemma 4.11, $\Pr(\phi_i = \phi) = N_\phi/W$. Further, $\mathbf{E}[Y_i | \phi_i = \phi] = M_\phi/N_\phi$. This is because once we obtain a partial match ϕ, we determine the number of subgraphs in $\mathcal{G}$ that contain ϕ i.e. N_ϕ. Subsequently, we divide the value returned by the DeriveCnt subroutine when invoked on ϕ with N_ϕ.

Therefore, $\mathbf{E}[Y_i] = \sum_{\phi \in \mathcal{S}} (M_\phi/N_\phi)(N_\phi/W) = \sum_{\phi \in \mathcal{S}} (M_\phi/W) = C/W$. Note that $\sum_{\phi \in \mathcal{S}} M_\phi = C$ because every instance H_G of H in G has a unique spanning tree T_{H_G} such that the subroutine DeriveCnt will account for H_G only when it is invoked with T_{H_G} as its input.

Therefore, $\mathbf{E}[Y] = kM/W$. So, $\widehat{C} = WY/k$, is an unbiased estimate of C. □

We now prove the following result. Let B denote the maximum number of matches to the target motif M resulting from matches to S in G.

Theorem 4.13. *Let $Y_1, Y_2, \ldots, Y_k$ be k random variables, where Y_i is the outcome of the subroutine ValidateAndDeriveCnt on the i-th sampled partial match to S. Suppose $k = (3B/\varepsilon^2)(W/C)\ln(2/\gamma)$ for any $\varepsilon, \gamma \in (0, 1)$. Then with probability at least $1 - \gamma$, $\widehat{C} \in [(1 - \varepsilon)C, (1 + \varepsilon)C]$.*

Proof. Recall that the random variables Y_i denote the outcome of the DeriveCnt when invoked on partial match ϕ_i and $Y =$

$\sum_{i \le k} Y_i$. First, note that the k random variables $Y_1, \ldots, Y_k$ are independent of each other. Also, observe that the random variables $Y_1, \ldots, Y_k$ may not necessarily assume values in the interval $[0, 1]$. Hence we cannot directly use the Chernoff Bound. However, dividing them by B ensures that $0 \le Y_i/B \le 1$ for all $1 \le i \le k$.

Before applying the Chernoff Bound, observe that the event $|Y/B - kM/BW| \ge (\varepsilon kM/WB)$ is identical to $|WY/k - C| \ge \varepsilon C$. Similarly, since $\widehat{C} = WY/K$, the events $|Y/B - kM/WB| \ge (\varepsilon kM/WB)$ and $|\widehat{C} - C| \ge \varepsilon C$ are also identical.

Note that $\mathrm{E}[Y/B] = \sum_{i \le k} \mathrm{E}[Y_i]/B = Mk/WB$. Using Chernoff Bound, for any $\varepsilon \in (0, 1)$, $\Pr(|Y/B - (kM/WB)| \ge (\varepsilon kM/WB)) \le 2\exp(-(\varepsilon^2/3)(kM/WB))$. Since $k = (3B/\varepsilon^2)(W/C)\ln(2/\gamma)$, it follows that $\Pr(|Y/B - (kM/BW)| \ge (\varepsilon kM/BW)) \le 2\exp(-(\varepsilon^2/3)(3B/\varepsilon^2)(W/C)(\ln(2/\gamma))(C/WB)) = \gamma$. □

We present the time and space complexity as follows.

Claim 4.14. *The* Preprocess *procedure takes* $O(|E(S)|^2 m d_{\max})$ *time. The* SampleSubg *procedure takes* $O(|E(S)|^2 d_{\max})$ *time per sample. The storage complexity is* $O(m|V(S)|)$. *Here,* m *is the number of edges in* G, *where* $d_{\max}$ *is the maximum number of simple edges incident on any vertex in* G.

Proof. PreprocessSubgraph computes the s-weights ($w_{s,e}$) for every edge $e \in E(G)$ and every $s \in E(S)$. To obtain $w_{s,e}$, we perform binary searches on the edges incident to one endpoint of e (say u) to locate candidate matches for edges in S, forming the set $D(s)$, in $O(\log m)$ time. For every $s' \in D(s)$, we retrieve the precomputed s'-weights of the matches incident on u (in $O(1)$ time each), and multiply them to get the s-weight of e. Thus, computing a single $w_{s,e}$ takes $O(|E(s)|d_{\max} + \log m)$ time. Repeating this for all $e \in E(G)$ and $s \in E(S)$ yields a total complexity of $O(|E(S)|^2 m d_{\max})$.

During the sampling phase, once we have sampled an edge e that matches s, we use binary search to determine potential matches to every $s' \in D(s)$, fetch their corresponding s'-weights and subsequently sample matches to every s' from their corresponding list of potential matches using distributions defined using s'-weight values. So the first step takes $O(|E(S)| \log m)$ time, the second step takes $O(|E(S)|d_{\max})$ and final step takes $O(|E(S)| \log m)$ time. We do these steps for every s in $E(S)$, so the overall time to sample an entire partial match is $O(|E(S)|^2 d_{\max})$ per sample.

Regarding the space complexity, we store the sampling weight $w_{s,e}$ in a vector for each edge in the graph G, whose size equal to the total edge count (m) multiplying the number of edges in the spanning tree of the motif ($|V(S)|$). □

4.5 Choosing the Spanning Tree

Figure 5b shows that a motif can have many spanning trees. Our goal is to select the spanning tree that minimizes runtime while having the best estimation quality. The estimation error when choosing 54 different spanning trees of the M_{6-4} motif is shown in Figure 6. The results reveal large variation in accuracy: the worst-case estimation error exceeds 350% while the best error is less than 1%.

In Theorem 4.13, given the motif M and input graph G, the number of samples k to take to reach convergence is proportional to the total sampling weight W. This weight W is exactly the number of instances of chosen spanning tree S in the graph. A smaller W indicates a rarer spanning tree, reducing the search space. For example,

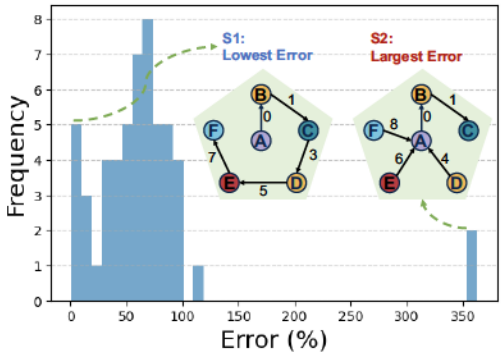


Figure 6: Histogram of estimation errors (%) across different spanning tree choices for motif M_{6-4} and with 1e8 samples. Two highlighted examples, S1 and S2, correspond to the lowest and highest observed errors, emphasizing that the choice of spanning tree significantly impacts estimation accuracy.

Algorithm 7 : GetBestSPTree(M, G, n_c)

Input: Temporal motif M, input graph G, number of candidate spanning trees n_c
Output: Selected spanning tree S_{best}

DFS to enumerate all spanning trees of M in an array $S_{all}[]$
$C\_losseness$ = GetConstraintLooseness(S_{all}, M)
S_{topk} = SelectTopK($C\_losseness, S_{all}, n_c$)
Init an array $est\_runtime[n_c]$
for $i \in [0, n_c)$ **do**
    $curW$ = PreprocessSubgraph($S_{topk}[i], G$)
    $est\_runtime[i] = curW$
S_{best} = tree with min $est\_runtime$
return S_{best}

Algorithm 8 : GetConstraintLooseness(S_{all}, M)

Input: all spanning trees $S_{all}[]$
Output: Constraint looseness $C\_lossness[]$

Initialize $C\_lossness[]$ with 0
for $i \in [0, len(S_{all}))$ **do**
    /*For every node, compute the absdiff of the edges incident to the node*/
    for $u \in nodes(M)$ **do**
        $E(u) \leftarrow$ edges if M that are incident on u
        if $|E(u)| < 2$ **then**
            continue
        /*Consider all pairs of incident edges*/
        for each unordered pair (e_1, e_2) in $E(u)$ **do**
            $C\_lossness[i] += |t(e_1) - t(e_2) - 1|$
return $C\_lossness$

consider the candidate spanning trees S_1 and S_2 of motif M_{5-3} in Figure 5b, or those of motif M_{6-4} in Figure 6. S_1 follows a path-like structure and S_2 has a more tree-like structure. The experiments in Table 7 show that the sampling weight W for S_2 is around 5-10

Table 2: Temporal graph datasets used in the evaluation.

Dataset	$\|V\|$	$\|E_{temporal}\|$	Time span (year)
wiki-talk (WT)	1.1M	7.8M	6.4
stackoverflow (SO)	2.6M	63.5M	7.6
bitcoin (BI)	48.1M	113.1M	7.1
reddit-reply (RE)	8.4M	636.3M	10.1

times larger than that for S_1. This indicates that S_1 appears less frequently in the input graph and introduces stricter constraints compared to S_2, so we prefer S_1 to S_2.

However, calculating W for *all* spanning trees is time-consuming. Each calculation takes approximately 10% of the overall runtime. We propose a heuristic that approximates how tightly a spanning tree enforces temporal constraints, which serves as a rough proxy for W without accessing the graph. This proxy, denoted as *C_looseness*, is computed in Algorithm 8. It examines all edge pairs incident to each node and computes how tightly the spanning tree orders them. A lower value indicates stricter constraints and is thus preferred.

Overall, our heuristics have the following steps.

- Finding Candidates: We use DFS to find all spanning trees and evaluate them based on motif-only information. Then we select the top n_c candidates with the smallest constraint losseness (GetConstraintLooseness).
- Evaluating Candidates: For the selected candidate spanning trees, we calculate the actual sampling weight using the input graph (PreprocessSubgraph) . A smaller W means fewer samples are required for convergence (indicated by Theorem 4.13).
- Choosing the Best Tree: We use the sampling weight W (which affects the number of samples) to estimate the total runtime. We select the spanning tree with the smallest estimated runtime, making it applicable across different graphs and motifs.

5 EVALUATION

In this section, we comprehensively evaluate the performance of TIMEST across different graphs, motifs, and δ values. We assess both the accuracy and the runtime of TIMEST and compare them with existing methods, highlighting its accuracy and efficiency.

5.1 Experiment Setup

Benchmarks. We evaluate the temporal motif counts across a spectrum of datasets, encompassing medium to large-scale graphs such as wiki-talk (WT), stackoverflow (SO) [27], bitcoin (BI) [22], and reddit-reply (RE) [16], detailed in Table 2. To showcase the versatility of TIMEST, we mine temporal motifs encompassing 5-vertex and 6-vertex connected graphs in Figure 3. We set δ as $8W$, $16W$ for WT and SO, and as $1D$ for BI and RE, where H represents hour, D represents day, and W represents week.

Baselines. For exact algorithms, we use the state-of-the-art Everest [65], implemented in [2], which builds on the BT algorithm [30] with GPU-specific optimizations. We exclude the original BT CPU version due to its significantly lower performance (over 20× slower than Everest).

For approximate algorithms on 4-vertex motifs, we compare against PRESTO [47], ES [59] and IS [29]. PRESTO is a sampling framework that uniformly samples intervals of length $c\delta$ and performs exact temporal motif counting on these sampled intervals to derive estimated counts. We explore two versions of PRESTO, referred to as *PRESTO-A* and *PRESTO-E*. For 5- for 6-vertex motifs, we omit the comparison with IS [29] as PRESTO was already shown to outperform it in their study. And we do not provide a quantitative comparison with ES [59] because the public code [1] is limited to motifs with 4 or fewer edges.

TIMEST The number of samples k affects the accuracy and runtime of TIMEST which varies from 10 million to 1 trillion. For the detailed settings per case, please refer to the `reproduce.py` in https://github.com/pyjhzwh/SpanningTreeSampling/. For a fair comparison, the runtime of TIMEST include all steps in Figure 4, which consists of preprocess and sample procedure.

Accuracy Metrics The accuracy of approximate algorithms is defined as $\frac{|C-\hat{C}|}{C}$, where C and $\hat{C}$ are the exact count and the estimated count, respectively. And error = 1- accuracy. We run it 5 times to get the average(avg) and standard deviation(std) of error.

Discussion Of The Samples TIMEST uses more samples than prior work, but the runtime per sample is extremely small (i.e., 0.1 microsecond per sample). PRESTO samples time windows and then runs an expensive algorithm on this sample. Each sample in TIMEST is a spanning tree that processes few edges. So, even including preprocessing time, TIMEST is faster.

Hardware Platforms. Both TIMEST and PRESTO run on a CPU platform with 32 threads. Specifically, the CPU platform utilized is an AMD EPYC 7742 64-core CPU with 64MB L2 cache, 256MB L3 cache, and 1.5TB DRAM memory. Everest [2] runs on a single NVIDIA A40 GPU with 10k CUDA cores and 48GB memory.

5.2 Results

TIMEST is fast. In Table 3, we list the runtime of Everest (GPU) and TIMEST (CPU) on counting the 5-vertex and 6-vertex temporal motifs in Figure 3. We focus on the cases where Everest runs more than 60 seconds. For RE dataset with δ=1 day, Everest is faster than TIMEST. With large number of nodes and edges in input graph and motifs, Everest takes a large amount of runtime, and even timeout (>1 day of execution time) in some cases. On the other hand, TIMEST completes execution in less than 30 minutes in all cases and exhibits a geomean speedup of 28× over Everest.

Further, we also compare TIMEST and Everest (GPU) on the bipartite money laundering pattern of Figure 1 [6]. We use the WT graph with δ set to 4W. Everest takes two days to complete, while TIMEST runs in *four minutes*. The error observed is less than 0.6%.

TIMEST is accurate. Our comparison between TIMEST and PRESTO [47] focuses on runtime performance and relative error, with findings summarized in Table 4. Despite PRESTO's runtime being 6× slower compared to TIMEST, our results show that TIMEST consistently outperforms PRESTO in accuracy for all evaluated temporal motifs involving 5 and 6 vertices.

PRESTO employs an approximation strategy that involves uniform sampling across graph partitions within a time window of size $c\delta$ ($c > 1$ is a parameter), and applying an exact algorithm on these partitions. PRESTO's inefficiency is because of the reliance on uniform sampling and the use of the slow backtracking (BT) algorithm [30] as the subroutine on the sampled time window.

Table 3: Runtime (in second) of Everest (GPU) and TIMEST (CPU), the speedup of TIMEST over Everest, and the estimation error (%) of TIMEST. We set the timeout limit as 1 day. We mark the speedup and error as ***No Exact*** for the case that Everest timeout. The error(%) is represented as avg ± std.

Graph	δ	Motif	Everest (GPU)	TIMEST (CPU)	Speedup (×)	Error(%)
WT	8W	$M_{5\text{-}1}$	9.9E1	9.3E0	10.6	0.1±0.0
		$M_{5\text{-}2}$	1.7E3	8.8E0	193.3	1.3±0.7
		$M_{5\text{-}3}$	1.5E3	7.1E1	21.1	0.7±0.5
		$M_{5\text{-}4}$	2.5E3	7.1E1	34.9	6.4±4.9
		$M_{5\text{-}5}$	3.4E3	4.3E2	8.0	9.4±5.3
		$M_{6\text{-}1}$	1.8E4	1.0E2	176.1	0.0±0.0
		$M_{6\text{-}2}$	1.2E4	1.4E1	852.9	0.1±0.1
		$M_{6\text{-}3}$	4.1E3	3.3E2	11.8	0.7±0.6
		$M_{6\text{-}4}$	5.5E4	1.3E3	42.5	7.4±3.8
		$M_{6\text{-}5}$	6.8E4	1.3E3	51.8	72.3±16.7
SO	8W	$M_{5\text{-}1}$	9.0E2	3.0E1	30.5	0.0±0.0
		$M_{5\text{-}2}$	7.7E3	2.9E1	265.0	0.9±0.3
		$M_{5\text{-}3}$	6.6E2	8.0E1	8.2	0.8±0.4
		$M_{5\text{-}4}$	4.0E2	8.1E1	5.0	20.4±12.6
		$M_{5\text{-}5}$	6.7E3	6.4E2	10.5	32.7±16.7
		$M_{6\text{-}1}$	3.1E4	4.2E1	734.8	0.1±0.1
		$M_{6\text{-}2}$	9.4E4	3.2E1	2980.1	0.2±0.0
		$M_{6\text{-}3}$	2.1E3	1.5E2	13.6	0.3±0.2
		$M_{6\text{-}4}$	2.9E3	1.0E3	2.8	33.1±18.5
		$M_{6\text{-}5}$	1.5E5	1.0E3	140.1	100.0±0
BI	1D	$M_{5\text{-}1}$	3.7E2	5.4E1	6.8	0.0±0.0
		$M_{5\text{-}2}$	2.1E3	4.9E1	43.1	2.2±0.7
		$M_{5\text{-}3}$	1.9E3	1.1E2	16.7	1.4±0.6
		$M_{5\text{-}4}$	7.7E3	1.1E2	68.3	3.2±3.1
		$M_{5\text{-}5}$	1.0E4	8.9E2	11.3	10.0±5.4
		$M_{6\text{-}1}$	5.1E4	6.2E1	812.7	0.1±0.0
		$M_{6\text{-}2}$	timeout	4.7E1	No Exact	No Exact
		$M_{6\text{-}3}$	3.0E3	1.4E2	20.9	0.2±0.2
		$M_{6\text{-}4}$	timeout	8.3E2	No Exact	No Exact
		$M_{6\text{-}5}$	timeout	8.2E2	No Exact	No Exact
RE	1D	$M_{5\text{-}1}$	6.3E1	3.9E2	0.2	0±0
		$M_{5\text{-}2}$	3.4E2	3.9E2	0.9	0.6±0.4
		$M_{5\text{-}3}$	5.5E2	4.1E2	1.4	0.4±0.5
		$M_{5\text{-}4}$	9.6E3	4.2E2	23.1	2.0±1.3
		$M_{5\text{-}5}$	9.9E3	1.3E3	7.8	19.9±10.6
		$M_{6\text{-}1}$	5.1E2	5.1E2	1.0	0.1±0.0
		$M_{6\text{-}2}$	3.1E3	3.9E2	7.9	0.0±0.0
		$M_{6\text{-}3}$	1.0E3	8.5E2	2.2	0.1±0.0
		$M_{6\text{-}4}$	timeout	8.5E2	No Exact	No Exact
		$M_{6\text{-}5}$	timeout	8.7E2	No Exact	No Exact

Table 4: Runtime (in second) and relative error (%) of approximate algorithms on various datasets and temporal motifs. The lowest error is highlighted in green block. All run in 32-thread. We set the timeout limit as (1 day). If the program runs out of memory (OOM), it will be killed. TIMEST is the fastest and the most accurate in most cases.

Dataset	Motif	PRESTO-A		PRESTO-E		TIMEST	
		Time (s)	Error	Time (s)	Error	Time (s)	Error
WT $\delta = 8W$	$M_{5\text{-}1}$	8.0E3	3.0%	4.0E4	7.1%	1.0E1	**0.1%**
	$M_{5\text{-}2}$	1.5E4	28.1%	1.6E4	2.2%	8.8E0	**1.3%**
	$M_{5\text{-}3}$	1.5E4	11.5%	1.9E4	12.0%	6.7E1	**0.7%**
	$M_{5\text{-}4}$	1.1E4	24.1%	5.1E4	33.6%	6.4E1	**6.4%**
	$M_{5\text{-}5}$	2.4E4	25.8%	2.4E4	77.1%	4.8E2	**9.4%**
	$M_{6\text{-}1}$	1.1E4	93.8%	2.0E3	98.8%	6.4E1	**0.0%**
	$M_{6\text{-}2}$	2.5E3	96.1%	timeout	N/A	1.1E1	**0.1%**
	$M_{6\text{-}3}$	1.4E5	107.5%	2.6E5	10.4%	2.0E2	**0.7%**
	$M_{6\text{-}4}$	1.3E4	89.5%	timeout	N/A	6.9E2	**7.4%**
	$M_{6\text{-}5}$	timeout	N/A	timeout	N/A	6.9E2	72.3%
SO $\delta = 8W$	$M_{5\text{-}1}$	4.9E3	3.8%	2.3E4	17.2%	2.6E1	**0.0%**
	$M_{5\text{-}2}$	OOM	N/A	OOM	N/A	2.5E1	**0.9%**
	$M_{5\text{-}3}$	8.0E2	33.4%	1.0E3	17.2%	6.6E1	**0.8%**
	$M_{5\text{-}4}$	4.0E2	87.3%	2.8E2	35.7%	6.9E1	**20.4%**
	$M_{5\text{-}5}$	1.5E4	81.5%	OOM	N/A	4.8E2	**32.7%**
	$M_{6\text{-}1}$	8.6E4	98.4%	timeout	N/A	3.2E1	**0.1%**
	$M_{6\text{-}2}$	OOM	N/A	OOM	N/A	2.9E1	**0.2%**
	$M_{6\text{-}3}$	1.4E4	45.0%	4.0E4	13.88%	1.2E2	**0.3%**
	$M_{6\text{-}4}$	1.3E3	81.4%	3.9E3	270.8%	8.7E2	**33.1%**
	$M_{6\text{-}5}$	OOM	N/A	OOM	N/A	8.8E2	100%
BI $\delta = 1D$	$M_{5\text{-}1}$	4.1E2	12.9%	2.7E3	13.2%	3.9E1	**0.0%**
	$M_{5\text{-}2}$	8.2E2	47.3%	1.3E3	41.7%	3.6E1	**2.2%**
	$M_{5\text{-}3}$	1.2E2	65.9%	3.8E2	39.4%	4.1E1	**1.4%**
	$M_{5\text{-}4}$	7.7E2	41.1%	5.4E2	53.3%	9.6E1	**3.2%**
	$M_{5\text{-}5}$	1.4E3	15.3%	2.8E3	11.5%	7.8E2	**10.0%**
	$M_{6\text{-}1}$	5.5E4	52.4%	timeout	N/A	5.2E1	**0.1%**
	$M_{6\text{-}2}$	timeout	N/A	timeout	N/A	5.2E1	NE
	$M_{6\text{-}3}$	8.5E2	26.5%	5.9E3	32.2%	1.2E2	**0.2%**
	$M_{6\text{-}4}$	NE	NE	NE	NE	7.9E2	NE
	$M_{6\text{-}5}$	NE	NE	NE	NE	8.1E2	NE
RE $\delta = 1D$	$M_{5\text{-}1}$	3.2E2	102.5%	1.6E3	14.4%	3.2E2	**0.0%**
	$M_{5\text{-}2}$	3.7E2	92.8%	2.4E3	45.8%	2.9E2	**0.6%**
	$M_{5\text{-}3}$	1.8E4	925.0%	1.9E3	95.5%	4.2E2	**0.4%**
	$M_{5\text{-}4}$	6.8E4	90.6%	5.6E4	97.6%	3.3E2	**2.0%**
	$M_{5\text{-}5}$	1.1E4	98.6%	2.5E4	25.8%	1.2E3	**19.9%**
	$M_{6\text{-}1}$	1.1E4	98.6%	4.5E4	99.9%	1.2E3	**0.1%**
	$M_{6\text{-}2}$	3.6E4	40.4%	1.3E5	6.0%	3.5E2	**0.0%**
	$M_{6\text{-}3}$	3.4E3	76.5%	1.1E3	timeout	4.5E2	**0.1%**
	$M_{6\text{-}4}$	NE	NE	NE	NE	7.9E2	NE
	$M_{6\text{-}5}$	NE	NE	NE	NE	8.1E2	NE

Both PRESTO and TIMEST struggle to accurately estimate counts for the most complex 6-clique motif ($M_{6\text{-}5}$), showing low convergence and accuracy. Despite generating approximately 1 billion samples, with around 100 million being valid (i.e., not violating the constraints defined in Definition 1.2), only 1 - 100 of these valid samples contribute to the final count. The 6-clique encompasses more constraints (such as edge mapping and ordering) than any chosen spanning tree can cover. Developing accurate algorithms for such complex motifs would require new ideas.

For smaller motifs with 4-vertex in Figure 3, we compare TIMEST with PRESTO, ES and IS baselines regarding runtime and relative error. The result is shown in Table 5. TIMEST is always faster by an order of magnitude with comparable or lower error.

Improved performance with additional constraints. Introducing specific constraints on sampled edges helps reduce the chances of them violating the requirements of Definition 1.2. In Table 6, we show how the three constraints in Section 2 reduce invalid samples and improve performance. We categorize three distinct invalid sample types because of violating: a) 1-1 vertex map ϕ_V b) δ time interval c) edge orders.

Our experimental approach employs Constraint 1 as the foundational condition, with Constraints 2 and 3 subsequently integrated in a phased manner. These experiments were conducted on the $M_{5\text{-}5}$ motif (5-cliques), utilizing the WT graph with $\delta = 8W$, and the RE graph with $\delta = 1D$. The objective is to achieve high accuracy with high efficiency, striving for an increase in the rate of valid samples while minimizing runtime for an equivalent number of samples.

From the table, we can see that adding Constraint 2 (distinct end vertices on dependent edges) reduces the number of invalid samples due to violating 1-1 vertex map. For RE graph, it significantly reduces the percentage of invalid samples due to violating the vertex map from 75% to 2%. Adding Constraint 3 (2δ window) reduces the invalid samples due to violating the δ constraints, increasing the valid sample rate by around 5%. Integrating all three constraints, we observe that the runtime remains largely unchanged, suggesting that the additional computational overhead introduced by these new constraints is minimal. More crucially, this integration boosts the valid sample rate from 1% to 33% for the RE graph, concurrently decreasing the error rate dramatically from 95% to 4%. *This demonstrates not only the feasibility of applying these constraints without significant efficiency losses but also their effectiveness in enhancing the precision of the sampling process.*

Table 5: Runtime (s) and relative error (%) of all algorithms on each dataset. The lowest error is highlighted in green block. Because not all sampling algorithms support multi-threading, for a fair comparison, we report the runtime for single-threaded implementations. Because TIMEST output count for $M_{4\text{-}1}$-$M_{4\text{-}4}$ at once, we consider the runtime of TIMEST as the total runtime for $M_{4\text{-}1}$-$M_{4\text{-}4}$. TIMEST is always the fastest.

Dataset	Motif	PRESTO-A		PRESTO-E		IS		ES		TIMEST		
		Time (s)	Error	Time (s)	Error	Time (s)	Error	Time (s)	Error	Time (s)	Speedup over PRESTO-A (×)	Error
WT, δ=8W	M4-1	3.65E+03	31.08%	3.62E+03	8.65%	1.32E+04	1.62%	2.22E+02	3.64%	5.16E+01	292	**0.26%**
	M4-2	3.76E+03	4.69%	4.04E+03	7.92%	1.44E+04	0.80%	2.71E+02	2.42%			**0.63%**
	M4-3	3.65E+03	75.88%	3.96E+03	34.45%	1.32E+04	**0.57%**	2.35E+02	8.34%			2.35%
	M4-4	3.99E+03	77.56%	3.74E+03	27.54%	1.45E+04	12.63%	5.01E+02	**2.27%**			4.30%
ST, δ=8W	M4-1	3.69E+03	13.16%	4.10E+03	19.17%	7.99E+04	0.68%	1.87E+03	0.75%	3.70E+02	41	**0.24%**
	M4-2	3.88E+03	86.36%	3.91E+03	12.84%	> 1D	-	1.50E+03	0.50%			**0.37%**
	M4-3	3.62E+03	38.68%	4.01E+03	20.29%	8.59E+04	1.45%	2.24E+02	**0.95%**			1.30%
	M4-4	3.85E+03	**0.52%**	4.07E+03	10.05%	8.12E+04	0.56%	3.12E+02	1.76%			4.74%
BI, δ=1D	M4-1	3.62E+03	5.33%	3.60E+03	19.10%	1.80E+03	1.81%	8.66E+03	2.12%	4.60E+02	32	**0.07%**
	M4-2	3.65E+03	18.50%	3.61E+03	8.94%	2.01E+03	**0.23%**	1.22E+03	1.48%			0.53%
	M4-3	3.64E+03	25.39%	3.67E+03	19.33%	1.79E+03	20.56%	5.24E+02	6.23%			**1.84%**
	M4-4	3.67E+03	41.71%	3.63E+03	31.99%	1.69E+03	16.61%	6.17E+02	**6.32%**			7.60%
RE, δ=1D	M4-2	3.60E+03	26.83%	3.61E+03	23.17%	3.67E+03	38.34%	1.85E+03	8.90%	2.42E+03	6	**0.22%**
	M4-3	3.60E+03	160.88%	3.60E+03	53.66%	5.91E+03	21.37%	1.08E+03	1.45%			**0.44%**
	M4-5	3.61E+03	89.03%	3.61E+03	46.35%	4.22E+03	33.11%	6.04E+02	2.96%			**1.92%**
	M4-7	3.60E+03	104.42%	4.54E+03	344.62%	7.29E+03	151.53%	1.01E+03	27.94%			**4.57%**

Table 6: The percentage of invalid and valid sample rate, relative error, and runtime when enforcing different constraints (C) for estimating temporal motif count of $M_{5\text{-}5}$.

Graph, δ		WI, δ=8W			RE, δ=1D		
Constraints		C1	C1+2	C1+2+3	C1	C1+2	C1+2+3
Invalid sample rate (%)	vertex map	42.9%	36.3%	36.6%	75.5%	2.0%	1.9%
	δ interval	14.6%	17.1%	10.3%	0.8%	23.1%	13.6%
	edge order	34.4%	35.8%	39.8%	23.0%	45.4%	51.1%
Valid samples rate (%)		8.1%	10.8%	13.3%	0.8%	29.5%	33.4%
Error (%)		5.1%	0.9%	2.9%	95.92%	12.5%	4.0%
Runtime (s)		320.1	332.7	361.0	1259.2	1221.7	1148.1

Table 7: Sampling weight W, estimation error (%) and runtime (s) for choosing different spanning tree (S_1 to S_3) when estimating motif count for $M_{5\text{-}3}$.

G	δ	W			Error			Runtime (s)		
		S_1	S_2	S_3	S_1	S_2	S_3	S_1	S_2	S_3
WT	4W	1.1E12	8.9E12	1.4E12	**4.9%**	9.5%	14.5%	**9.9**	12.4	11.7
	8W	6.5E12	5.2E13	8.0E12	**0.8%**	13.4%	3.4%	**14.2**	16.4	17.7
SO	4W	3.1E12	2.6E13	4.0E12	**3.1%**	33.1%	5.8%	25.4	**19.8**	27.0
	8W	1.6E13	1.5E14	2.0E13	**4.4%**	21.8%	11.4%	30.0	**20.7**	31.1
BI	1D	2.7E13	1.3E15	1.2E14	**1.2%**	110.6%	3.9%	42.9	**33.6**	48.1
RE	1D	7.3E11	4.9E12	5.3E12	**0.3%**	3.5%	6.8%	327.3	**251.5**	336.0

Effect of different spanning trees on performance. We examined using three distinct spanning trees for motif $M_{5\text{-}3}$, using 20 million samples each for our analysis, as seen in Table 7. The trees S_1 and S_2 are displayed in Figure 5b. The total sampling weight W is similar for S_1 and S_3 in certain graphs, but S_2 has a W value about 10 times higher. Theorem 4.13 states that for a given graph and motif, the number of samples k needed to reach convergence is proportional to W. Smaller W implies the same error with fewer samples, which is corroborated by Table 7.

Runtime comparison, given equal sample numbers, reveals minor differences between the trees, with S_3 having slightly longer runtimes than S_1 and S_2. Because the ValidateAndDeriveCnt() function has time complexity linear to the overall number of potential candidates (see Section 4.3). Here the candidates refer to the non-spanning-tree edges of the motif.

Table 8: Peak memory usage (GB) for PRESTO and TIMEST.

Dataset	Motif	PRESTO-A	PRESTO-E	TIMEST
WT $\delta = 8W$	$M_{5\text{-}1}$	1.5	3.7	2.9
	$M_{5\text{-}3}$	15.9	37.2	2.8
	$M_{6\text{-}3}$	2.7	6.9	3.0
SO $\delta = 8W$	$M_{5\text{-}1}$	12.4	13.1	20.1
	$M_{5\text{-}3}$	98.1	111.6	20.1
	$M_{6\text{-}3}$	26.8	30.8	21.0
BI $\delta = 1D$	$M_{5\text{-}1}$	4.2	5.1	40.3
	$M_{5\text{-}3}$	8.3	19.2	40.3
	$M_{6\text{-}3}$	5.7	11.9	42.1

Peak memory usage In Table 8, we show the maximum RAM memory used for various algorithms in a single run. We can see that TIMEST and PRESTO have comparable peak memory usage. Notice that the peak memory usage of TIMEST does not grow with the motif's complexity or the number of samples. Because most of the memory is used to store the preprocessing weights.

6 CONCLUSION AND FUTURE WORK

This paper presents TIMEST—a general algorithm for counting motifs in temporal graphs that is both fast and accurate. TIMEST works for any motif with arbitrary size and shape. The key idea behind TIMEST is to estimate the motif counts based on an underlying spanning tree structure. To improve estimation accuracy and achieve high performance, we show how to intelligently design a constrained algorithm and present the heuristics for choosing spanning trees. TIMEST is designed under the assumption of total temporal orderings following temporal motif mining problems in prior works [30]. A natural next step is to try to count motifs with partial order time constraints. In some special cases, the spanning tree of TIMEST already suffices, but fully supporting arbitrary partial orderings remains an interesting direction for future work.

7 ACKNOWLEDGEMENTS

Both OB and CS are supported by NSF grants CCF-1740850, CCF-2402572, and DMS-2023495.

REFERENCES

[1] 2021. ES Code Repository. https://github.com/jingjing-cs/Temporal-Motif-Counting.

[2] 2023. Everest Code Repository. https://github.com/yichao-yuan-99/Everest.

[3] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1446–1455.

[4] Nesreen K Ahmed, Nick Duffield, and Ryan A Rossi. 2021. Online sampling of temporal networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 4 (2021), 1–27.

[5] Uri Alon. 2007. Network motifs: theory and experimental approaches. *Nature Reviews Genetics* 8, 6 (2007), 450–461.

[6] Erik Altman, Jovan Blanuša, Luc Von Niederhäusern, Béni Egressy, Andreea Anghel, and Kubilay Atasu. 2024. Realistic synthetic financial transactions for anti-money laundering models. *Advances in Neural Information Processing Systems* 36 (2024).

[7] Hanjo D Boekhout, Walter A Kosters, and Frank W Takes. 2019. Efficiently counting complex multilayer temporal motifs in large-scale networks. *Computational Social Networks* 6, 1 (2019), 8.

[8] Giorgos Bouritsas, Fabrizio Frasca, Stefanos P Zafeiriou, and Michael Bronstein. 2022. Improving graph neural network expressivity via subgraph isomorphism counting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

[9] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2018. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 12, 4 (2018), 1–25.

[10] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: fast motif counting via succinct color coding and adaptive sampling. 12, 11 (2019).

[11] Xinwei Cai, Xiangyu Ke, Kai Wang, Lu Chen, Tianming Zhang, Qing Liu, and Yunjun Gao. 2024. Efficient Temporal Butterfly Counting and Enumeration on Temporal Bipartite Graphs. *Proc. VLDB Endow.* 17, 4 (mar 2024), 657–670. https://doi.org/10.14778/3636218.3636223

[12] Xuhao Chen et al. 2022. Efficient and Scalable Graph Pattern Mining on {GPUs}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 857–877.

[13] Zhongqiang Gao, Chuanqi Cheng, Yanwei Yu, Lei Cao, Chao Huang, and Junyu Dong. 2022. Scalable motif counting for large-scale temporal graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2656–2668.

[14] Aris Gionis, Lutz Oettershagen, and Ilie Sarpe. 2024. Mining Temporal Networks. https://miningtemporalnetworks.github.io/.

[15] 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.

[16] Jack Hessel, Chenhao Tan, and Lillian Lee. 2016. Science, askscience, and badscience: On the coexistence of highly related communities. In *Proceedings of the international AAAI conference on web and social media*, Vol. 10. 171–180.

[17] Yuriy Hulovatyy, Huili Chen, and Tijana Milenković. 2015. Exploring the structure and function of temporal networks with dynamic graphlets. *Bioinformatics* 31, 12 (2015), i171–i180.

[18] Shweta Jain and C Seshadhri. 2017. A fast and provable method for estimating clique counts using turán's theorem. In *Proceedings of the 26th international conference on world wide web*. 441–449.

[19] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: a pattern-aware graph mining system. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[20] Kasra Jamshidi and Keval Vora. 2021. A deeper dive into pattern-aware subgraph exploration with peregrine. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 1–10.

[21] Madhav Jha, C Seshadhri, and Ali Pinar. 2015. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proceedings of the 24th international conference on world wide web*. 495–505.

[22] Dániel Kondor, István Csabai, János Szüle, Márton Pósfai, and Gábor Vattay. 2014. Inferring the interplay between network structure and market effects in Bitcoin. *New Journal of Physics* 16, 12 (2014), 125003.

[23] 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.

[24] Lauri Kovanen, Kimmo Kaski, János Kertész, and Jari Saramäki. 2013. Temporal motifs reveal homophily, gender-specific patterns, and group talk in call sequences. *Proceedings of the National Academy of Sciences* 110, 45 (2013), 18070–18075.

[25] 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.

[26] Mayank Lahiri and Tanya Y Berger-Wolf. 2007. Structure prediction in temporal networks using frequent subgraphs. In *2007 IEEE Symposium on computational intelligence and data mining*. IEEE, 35–42.

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[28] Penghang Liu, Rupam Acharyya, Robert E Tillman, Shunya Kimura, Naoki Masuda, and Ahmet Erdem Sarıyüce. 2023. Temporal Motifs for Financial Networks: A Study on Mercari, JPMC, and Venmo Platforms. *arXiv preprint arXiv:2301.07791* (2023).

[29] Paul Liu, Austin R Benson, and Moses Charikar. 2019. Sampling methods for counting temporal motifs. In *Proceedings of the twelfth ACM international conference on web search and data mining*. 294–302.

[30] Patrick Mackey, Katherine Porterfield, Erin Fitzhenry, Sutanay Choudhury, and George Chin. 2018. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE international conference on big data (big data)*. IEEE, 3972–3979.

[31] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2019. Graphzero: Breaking symmetry for efficient graph mining. *arXiv preprint arXiv:1911.12877* (2019).

[32] Daniel Mawhirter and Bo Wu. 2019. Automine: harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 509–523.

[33] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[34] Seunghwan Min, Jihoon Jang, Kunsoo Park, Dora Giammarresi, Giuseppe F Italiano, and Wook-Shin Han. 2023. Time-Constrained Continuous Subgraph Matching Using Temporal Information for Filtering and Backtracking. *arXiv preprint arXiv:2312.10486* (2023).

[35] What Is Data Mining. 2006. Data mining: Concepts and techniques. *Morgan Kaufinann* 10, 559-569 (2006), 4.

[36] Mark EJ Newman. 2003. The structure and function of complex networks. *SIAM review* 45, 2 (2003), 167–256.

[37] Lutz Oettershagen, Nils M. Kriege, Claude Jordan, and Petra Mutze. 2023. *A Temporal Graphlet Kernel For Classifying Dissemination in Evolving Networks*. 19–27. https://doi.org/10.1137/1.9781611977653.ch3

[38] Raj Kumar Pan and Jari Saramäki. 2011. Path lengths, correlations, and centrality in temporal networks. *Physical Review E* 84, 1 (2011), 016105.

[39] Yunjie Pan, Omkar Bhalerao, C Seshadhri, and Nishil Talati. 2024. Accurate and Fast Estimation of Temporal Motifs using Path Sampling. *arXiv preprint arXiv:2409.08975* (2024).

[40] Yunjie Pan, Omkar Bhalerao, C. Seshadhri, and Nishil Talati. 2024. Accurate and Fast Estimation of Temporal Motifs Using Path Sampling. In *International Conference on Data Mining (ICDM)*. 809–814.

[41] 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.

[42] Noujan Pashanasangi and C Seshadhri. 2021. Faster and generalized temporal triangle counting, via degeneracy ordering. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1319–1328.

[43] Aduri Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and sampling triangles from a graph stream. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1870–1881.

[44] Ali Pinar, Comandur Seshadhri, and Vaidyanathan Vishal. 2017. Escape: Efficiently counting all 5-vertex subgraphs. In *Proceedings of the 26th international conference on world wide web*. 1431–1440.

[45] Jiaxi Pu, Yanhao Wang, Yuchen Li, and Xuan Zhou. 2023. Sampling Algorithms for Butterfly Counting on Temporal Bipartite Graphs. *arXiv preprint arXiv:2310.11886* (2023).

[46] Ilie Sarpe and Fabio Vandin. 2021. OdeN: simultaneous approximation of multiple motif counts in large temporal networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1568–1577.

[47] Ilie Sarpe and Fabio Vandin. 2021. Presto: Simple and scalable sampling techniques for the rigorous approximation of temporal motif counts. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 145–153.

[48] T. Schank and D. Wagner. 2005. Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study. In *Experimental and Efficient Algorithms*. Springer, 606–609.

[49] C Seshadhri, Ali Pinar, and Tamara G Kolda. 2014. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7, 4 (2014), 294–307.

[50] C Seshadhri and Srikanta Tirthapura. 2019. Scalable subgraph counting: The methods behind the madness: WWW 2019 tutorial. In *Proceedings of the Web Conference (WWW)*, Vol. 2. 75.

[51] Shiva Shadrooh and Kjetil Nørvåg. 2024. SMoTeF: Smurf money laundering detection using temporal order and flow analysis. *Applied Intelligence* (2024), 1–18.

[52] Shai S Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. 2002. Network motifs in the transcriptional regulation network of Escherichia coli. *Nature genetics* 31, 1 (2002), 64–68.

[53] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. Graphpi: High performance graph pattern matching through effective redundancy elimination. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[54] Xiaoli Sun, Yusong Tan, Qingbo Wu, Baozi Chen, and Changxiang Shen. 2019. Tm-miner: Tfs-based algorithm for mining temporal motifs in large temporal network. *IEEE Access* 7 (2019), 49778–49789.

[55] Toyotaro Suzumura and Hiroki Kanezashi. 2021. Anti-Money Laundering Datasets: InPlusLab Anti-Money Laundering DataDatasets.

[56] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: a system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 425–440.

[57] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 837–846.

[58] Ata Turk and Duru Turkoglu. 2019. Revisiting wedge sampling for triangle counting. In *The World Wide Web Conference*. 1875–1885.

[59] Jingjing Wang, Yanhao Wang, Wenjun Jiang, Yuchen Li, and Kian-Lee Tan. 2020. Efficient sampling algorithms for approximate temporal motif counting. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1505–1514.

[60] Pinghui Wang, Junzhou Zhao, Xiangliang Zhang, Zhenguo Li, Jiefeng Cheng, John CS Lui, Don Towsley, Jing Tao, and Xiaohong Guan. 2017. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *IEEE Transactions on Knowledge and Data Engineering* 30, 1 (2017), 73–86.

[61] Yihua Wei and Peng Jiang. 2022. STMatch: accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[62] Yixing Yang, Yixiang Fang, Maria E Orlowska, Wenjie Zhang, and Xuemin Lin. 2021. Efficient bi-triangle counting for large bipartite networks. *Proceedings of the VLDB Endowment* 14, 6 (2021), 984–996.

[63] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongzhi Chen, and Guoren Wang. 2022. Lightning Fast and Space Efficient k-clique Counting. In *Proceedings of the ACM Web Conference 2022*. 1191–1202.

[64] Xiaowei Ye, Rong-Hua Li, Qiangqiang Dai, Hongchao Qin, and Guoren Wang. 2023. Efficient Biclique Counting in Large Bipartite Graphs. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–26.

[65] Yichao Yuan, Haojie Ye, Sanketh Vedula, Wynn Kaza, and Nishil Talati. 2024. Everest: GPU-Accelerated System For Mining Temporal Motifs. In *50th International Conference on Very Large Databases (VLDB 2024)*. ACM.