The k-Trine Cohesive Subgraph and Its Efficient Algorithms

Jinyu Duan
CIAT & Huangpu Research School
Guangzhou University
Guangzhou, Guangdong, China
jduan@e.gzhu.edu.cn

Haicheng Guo
CIAT & Huangpu Research School
Guangzhou University
Guangzhou, Guangdong, China
guohaicheng@e.gzhu.edu.cn

Fan Zhang*
CIAT & Huangpu Research School
Guangzhou University
Guangzhou, Guangdong, China
zhangf@gzhu.edu.cn

Kai Wang
Antai College of Economics and Management
Shanghai Jiao Tong University
Shanghai, China
w.kai@sjtu.edu.cn

Zhengping Qian
Alibaba Cloud
Alibaba Group
Hangzhou, Zhejiang, China
zhengping.qzp@alibaba-inc.com

Zhihong Tian*
CIAT & Huangpu Research School
Guangzhou University
Guangzhou, Guangdong, China
tianzhihong@.gzhu.edu.cn

Abstract

In this paper, we introduce and study a novel cohesive subgraph model, named k-trine, to address the defects in the classical k-core and k-truss models. Our analysis shows that the k-trine is a more feasible model for capturing cohesive subgraphs by containing the strongly connected vertices. We analyze the theoretical properties of k-trine and propose efficient algorithms to compute the k-trine. Particularly, we design batch processing algorithms to update the decomposition of k-trine against highly dynamic graphs. Extensive experiments on real-world networks validate the effectiveness of the k-trine model and the efficiency of our algorithms.

CCS Concepts

• **Theory of computation → Dynamic graph algorithms**; • **Human-centered computing → Social networks**.

Keywords

Social networks; Cohesive subgraph; Dynamic algorithms.

ACM Reference Format:
Jinyu Duan, Haicheng Guo, Fan Zhang, Kai Wang, Zhengping Qian, and Zhihong Tian. 2025. The k-Trine Cohesive Subgraph and Its Efficient Algorithms. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3690624.3709174

KDD Availability Link:
The source code of this paper has been made publicly available at https://doi.org/10.5281/zenodo.14539433.

*Corresponding authors: Fan Zhang, Zhihong Tian.

1 Introduction

Graphs are widely used for modeling large-scale entities and their relations in different areas, e.g. social networks [32], the World Wide Web [17], collaboration networks [4], and biology networks [7]. As a fundamental graph problem, cohesive subgraph mining is to find a group of well-connected vertices, widely used in different applications, e.g., community discovery [21, 44, 47], predicting network collapse [30], anomaly detection [37], optimizing network resilience [38], and user engagement analysis [27, 45].

The k-core is a widely studied cohesive subgraph model, defined as a maximal subgraph in which each vertex has at least k neighbors inside [28, 35]. Although the computation cost is linear, k-core is often too relaxed and is regarded as a seedbed to find subgraphs with higher cohesiveness. Since real-world connections often have different strengths, the k-truss model is proposed by preserving some strong edges. As the strength (i.e., support) of an edge is often estimated by the number of triangles containing it [14, 16, 24, 36], the k-truss is defined as a maximal subgraph S where each edge is contained in at least $k-2$ triangles in S[10].

Although k-truss is an enhanced model compared with k-core, we think a new model is also needed to find/analyze communities from a different angle: 1) Real communities are usually vertex-oriented, i.e., the existence of an edge is dependent on the existence of its endpoints, but the k-truss is defined on edges. For instance, k-truss may remove a "weak" edge even if its endpoints belong to k-truss; 2) The removal of "weak" edges will decrease the supports of other edges excessively s.t. some strong edges are regarded as weak and then removed; and 3) Some strongly engaged nodes are excluded from k-truss due to the contagious underestimation of tie strength and iterative removal of "weak" edges.

To address the above concerns, we propose a novel cohesive subgraph model, named k-trine, defined as a maximal subgraph S in which the support of each vertex is at least k in S. The support of a vertex v in a subgraph S is the sum of the supports of v's incident edges in S, i.e., 2 times the number of triangles containing v in S.

The focus of the k-trine model is different from the k-truss model: The truss requires each vertex to have at least one strong edge, while the trine additionally captures strongly-engaged vertices including those with many weak edges. Naturally, we find each k-truss is contained by the $(k-2)(k-1)$-trine (Prop. 2). Our case study on

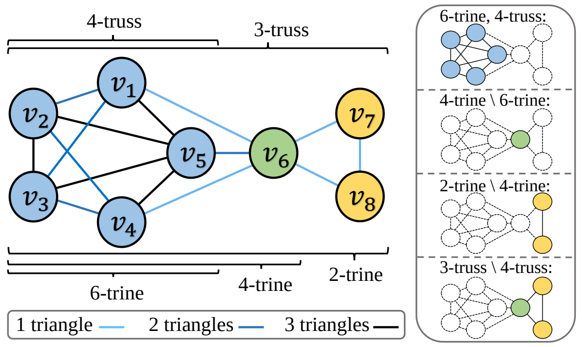


Figure 1: The Difference between k-Trine and k-Truss

DBLP shows that any author who published one paper with at least 6 authors belongs to the 6-truss. However, an author who published many papers with less than 6 authors can be overlooked by the 6-truss but included in the 20-trine (as Exp-2 of Sec. 6.1).

In trine decomposition, we compute the unique triness for each vertex v, i.e., the largest k with k-trine containing v. We can immediately retrieve a k-trine by the vertices with triness no less than k and the incident edges. Since the vertex support can be much larger than edge support, the k-trine has finer granularity in decomposing a graph. In existing works, the decomposition number of a node, e.g., coreness, is the "best practice" to estimate its engagement in the network [45]. Since the maximum triness of a network is significantly larger than coreness and trussness, it can categorize users into finer groups for user engagement study (as Exp-1 of Sec. 6.1).

Fig. 1 shows a toy graph G in which edge colors represent different support values in G. G itself is 3-truss or 2-trine. The k-truss can only extract a 4-truss from G where v_6 is classified as the same as v_7 and v_8 but v_6 is better engaged in G. Our k-trine model divides the graph into three parts, from 2-trine, 4-trine to 6-trine, where v_6, as a part of 4-trine, is different from v_7 and v_8.

In this paper, we also study the computational problems for k-trine, including the computation of k-trine with a given k, the trine decomposition on static graphs, and the trine maintenance to update the triness of each vertex against edge insertions/removals.

Challenges. To facilitate efficient computations, we should analyze the properties of k-trine in-depth, including its relations with k-core and k-truss. The computation on static graphs should minimize the enumeration of triangles which may dominate the cost. The challenges for the trine maintenance problem are significant: 1) We prove that the maintenance problem is unbounded for edge insertions (Def. 8 and Thm. 1); 2) The finer granularity of trine decomposition lends to the fact that a few edge insertions/removals can affect massive vertices; and 3) Batch processing is needed for processing large data streams, but it involves more complex changes compared with single-edge updates.

Our Solutions. We compute the k-trine on the subgraph of $\lfloor \sqrt{k+0.25}+0.5 \rfloor$-core based on our analysis and further remove the unqualified vertices. For trine decomposition, we iteratively remove the vertices with insufficient supports.

As for trine maintenance, we first reduce the problem to a vertex-order maintenance problem s.t. a novel framework is proposed to handle batch updates. For edge insertion, our algorithm enumerates the triangles at most twice for the vertices with potential triness updates and reposition the vertices in the order at most once. Considering that the domination cost is from triangle enumeration, we propose a pruning technique by delaying the update of each vertex's neighbor information to reduce the actual number of enumerated triangles. We also propose an edge removal algorithm that is compatible with the insertion algorithm and bounded regarding the removed edges and the changes of triness.

Contributions. The contributions of the paper are as follows.

- We formally introduce the k-trine model[1], study its properties, and validate its effectiveness in real social networks.
- The k-trine computation algorithms on static graphs are designed with the properties of k-trine. Besides, we define the k-trine maintenance problem and analyze its (un)boundedness.
- We reduce the trine maintenance to a vertex-order maintenance problem for efficient solutions. Two novel maintenance algorithms are proposed for edge insertions/removals. We optimize edge insertions by delaying the updates to reduce the cost of triangle enumeration.
- Extensive experiments on 10 real-world datasets demonstrate that the result of k-trine is more reasonable, the trine decomposition has a finer granularity, our maintenance algorithms are faster than the decomposition by $1-3$ orders of magnitude, for processing 500 inserted/removed edges.

2 Related Work

Due to the diversity of real-world scenarios, different cohesive subgraphs are proposed and studied, such as clique [2], k-core [28, 35], k-truss [10], etc. The theories, algorithms, and applications of cohesive subgraphs are surveyed in the literature, e.g., [8, 12, 26, 48].

Core-related Models and Algorithms. The k-core is a widely studied relaxation of the clique model and the seedbed for computing cohesive subgraphs [28, 35]. Core decomposition is to compute the k-core with every possible k. On static graphs, core decomposition runs in $O(m)$ time by using bin-sort [5]. On dynamic graphs, the core maintenance problem is first studied with single-edge updates [20, 33, 34] and also with batch updates [50]. Some studies propose parallel algorithms for k-core computation, e.g., [23, 42].

In addition, k-core-related models are extensively studied in different graphs and scenarios [3, 15, 22, 25, 46, 47]. The (k, h)-core is studied in [3] with maintenance algorithms on temporal graphs. The (α, β)-core is a variant of the k-core on bipartite graphs and the maintenance algorithms are proposed in [22]. Some other models such as k-peak [15] and (k, r)-core [47] are mainly studied on static graphs. In contrast to the above models, k-truss and k-trine further consider tie strength in computing cohesive subgraphs.

Truss-related Models and Algorithms. As triangles represent stable and strong vertex relations [43], k-truss is defined as a maximal subgraph in which each edge is contained in at least $k-2$ triangles in the subgraph [10]. The static truss decomposition runs

[1]As discussed in Sec. 2, although k-trine is a special case of (k, Φ)-core, we are the first to study the k-trine as a cohesive subgraph model including its effectiveness and the efficient algorithms. Note that (k, Φ)-core is used solely for optimizing densest subgraph computation [13].

Table 1: Summary of Notations

Notation	Definition
G	an unweighted and undirected graph
$V(G); E(G)$	the vertex/edge set of G
$G[V]$	the subgraph of G induced by vertex set V
$\triangle(u, v, w)$	a triangle containing vertices u, v and w
$N_G(v)$	the neighbor set of vertex v in G
$deg_G(v)$	the degree of vertex v in G
$sup_G(u, v)$	the edge support of (u, v) in G, i.e. $sup_G(u, v) = \lvert\{\triangle(u, v, w) \mid w \in V(G)\}\rvert$
$sup_G(v)$	the vertex support of v in G, i.e. $sup_G(v) = \sum_{u \in N_G(v)} sup_G(u, v)$
$T_k(G)$	the k-trine of G
$t(v)$	the triness of vertex v in G
$\preceq$	a vertex set ordered by the vertex deletion sequence in trine decomposition, or the sequence of vertex deletion in trine decomposition
$\preceq_v$	a subset of $\preceq$ formed by vertex v and the vertices behind v in $\preceq$, i.e., $\preceq_v = \{u \mid v \preceq u \vee u = v\}$
Φ_k	a subset of $\preceq$ formed by every vertex $v \in \preceq$ with $t(v) = k$, i.e., $\Phi_k = \{v \mid v \in \preceq \wedge t(v) = k\}$
$rem(v)$	the remaining support of v, i.e., $rem(v) = sup_{G[\preceq_v]}(v)$
$ext(v)$	the extra support of v, i.e., $ext(v) = sup_{G'[\preceq_v]}(v) - rem(v)$ where G' is the updated graph
$ts(v)$	the support of v in $sup_{T_{t(v)}}(v)$, i.e., $ts(v) = sup_{T_{t(v)}}(v)$

in $O(m^{1.5})$ time by using the bin-sort [41]. Truss maintenance algorithms are proposed in [16] to handle single-edge updates. Batch-update algorithms are proposed in [49] which is the state-of-the-art for truss maintenance. There are also some truss-related models, such as the (k_c, k_f)-truss on directed graphs [39] and the (k, s)-core considering weak ties [46]. Nevertheless, the truss-related models still face some issues: 1) some models are not vertex-oriented, 2) the inaccurate strength estimation after removing all the weak ties, and 3) the deletion of some nodes with factually strong ties.

k-Trine for Densest Subgraph Discovery. Due to the efficiency, some cohesive subgraphs are used to discover the densest subgraph [29, 31]. To solve the k-clique densest subgraph problem proposed by [40], the (k, Φ)-core decomposition [13] (i.e., k-clique core decomposition mentioned in [11]) is introduced as a technique to speed up the computation. Although k-trine can be regarded as a special case of (k, Φ)-core when Φ is 3-clique, our paper is the first to study the effectiveness of k-trine other than computing the densest subgraph, and the efficient algorithms on computing the k-trine with a given k and maintain the triness values.

3 Basic Concepts

3.1 Definitions of k-Trine

Let $G = (V, E)$ denote a simple, undirected, and unweighted graph, with $n = |V|$ vertices and $m = |E|$ edges (assume $m > n$). Given a node $v \in V(G)$, the neighbor set of v in G is denoted as $N_G(v) = \{u \mid u \in V(G) \wedge (u, v) \in E(G)\}$, and the degree of v is denoted as $deg_G(v) = |N_G(v)|$. Let $\triangle(u, v, w)$ be a triangle in graph G, i.e., a cycle with length 3, formed by the edges (u, v), (v, w) and (w, u) in $E(G)$. The notions used in the paper are summarized in Tab. 1. *We may omit G in notations when the context is clear.*

Definition 1. ***Edge Support.*** *The support of an edge $e = (u, v)$ in G, denoted by $sup_G(u, v)$ or $sup_G(e)$, is the number of triangles in G that contain (u, v), i.e., $sup_G(u, v) = |\{\triangle(u, v, w) \mid w \in V(G)\}|$.*

The support of an edge (u, v) well estimates the strength of the edge, e.g., the relation of u and v is strong if they have multiple common neighbors [14].

Definition 2. ***Vertex Support.*** *The support of a vertex u in G, denoted by $sup_G(u)$, is the sum of supports from all the edges incident to u, i.e., $sup_G(u) = \sum_{v \in N_G(u)} sup_G(u, v)$.*

The support of a vertex u is equivalent to twice the number of triangles in G which contain u. So, the following observation holds.

Observation 1. *We have $sup_G(u) = \sum_{v \in N_G(u)} sup_G(u, v) = 2 \cdot |\{\triangle(u, v, w) \mid v, w \in V(G)\}|$ for each $u \in V(G)$.*

Definition 3. ***k-Trine.*** *Given a graph G and an integer k, a subgraph S is the k-trine of G, denoted by $T_k(G)$, if (i) the support of each vertex in S is at least k, i.e., $\forall u \in V(S), sup_S(u) \geq k$; and (ii) S is maximal, i.e., any supergraph of S is not a k-trine except S itself.*

The k-trine has the following containment relation.

Observation 2. *For every integer k, the k-trine of a graph G is a subgraph of the $(k-1)$-trine of G.*

According to Obs. 2, each vertex has a unique triness value.

Definition 4. ***Triness.*** *Given a graph G, the triness of a vertex $u \in G$, denoted by t(u), is the largest k such that u is in the $T_k(G)$ but not in $T_{k+1}(G)$, i.e., $t(u) = \arg\max_{k \in \mathbb{N}} u \in T_k(G)$.*

The triness of a vertex u determines which k-trine contains u.

Observation 3. *A vertex $u \in T_k(G)$ if and only if $k \leq t(u)$.*

3.2 Properties of k-Trine

In this subsection, we introduce the properties of k-trine including the relations with k-core and k-truss models, and the diameter limit. The proof of all the properties in this section is given in Appx. A.1.

Lemma 1. *$\forall u \in G, deg(u) \times (deg(u) - 1) \geq sup(u) \geq t(u)$.*

Given a graph G, k-core is defined as the maximal subgraph in which each vertex has at least k neighbors in the subgraph [28, 35]. The relation between k-trine and k-core is as follows.

Property 1. *Each k-trine of G is a subgraph of the $\lfloor\sqrt{k + 0.25} + 0.5\rfloor$-core of G.*

Given a graph G, k-truss is defined as the maximal subgraph in which each edge is contained in at least $k - 2$ triangles in the subgraph [10]. The relation between k-trine and k-truss is as follows.

Property 2. *Each k-truss of G is a subgraph of the $(k-2) \times (k-1)$-trine of G.*

The above properties are tight since in some cases the k-truss and $(k-1)$-core can be all the same as the $(k-2)(k-1)$-trine. Besides, we cannot derive bounds from the opposite direction since for any non-trivial function $f(x)$, we cannot ensure that x-core is always a subgraph of $f(x)$-trine. And this also applies to k-truss. The above properties imply that the structural cohesion of the k-trine subgraph lies between a k'-core and a k''-truss.

Then, we give the *diameter limit* of k-trine.

```
Algorithm 1: Compute $k$-Trine
  Input: a graph $G = (V, E)$, an integer $k$
  Output: the $k$-trine $T_k(G)$
  $G' \leftarrow \lfloor\sqrt{k+0.25}+0.5\rfloor$-core;
  for each $v \in V(G')$ do
      compute $sup_{G'}(v) = 2 \times |\{\triangle(u,v,w) \mid (u,w) \in G'\}|$;
  $Del \leftarrow \{v \mid v \in V(G') \wedge sup_{G'}(v) < k\}$;
  while $Del \neq \emptyset$ do
      $v \leftarrow$ a vertex in $Del$;
      $Del \leftarrow Del - \{v\}$;
      for each $(u,w) \in \{(u,w) \mid u,w \in N_{G'}(v) \wedge (u,w) \in E\}$ do
          $sup_{G'}(u) \leftarrow sup_{G'}(u) - 2$ if $sup_{G'}(u) \geq k$;
          $sup_{G'}(w) \leftarrow sup_{G'}(w) - 2$ if $sup_{G'}(w) \geq k$;
      $Del \leftarrow Del \cup \{u \mid u \in N_{G'}(v) \wedge sup_{G'}(u) < k\}$;
      $E(G') \leftarrow E(G') - \{(u,v) \mid u \in N_{G'}(v)\}$;
      $V(G') \leftarrow V(G') - \{v\}$;
  Return: $G'$
```

```
Algorithm 2: TrineD
  Input: a graph $G = (V, E)$
  Output: the triness of each vertex $t(\cdot)$ in $G$
  $G^* \leftarrow G; k \leftarrow 0$;
  for each $v \in V(G^*)$ do
      compute $sup_{G^*}(v) = 2 \times |\{\triangle(u,v,w) \mid (u,w) \in G^*\}|$;
  while $V(G^*) \neq \emptyset$ do
      $v \leftarrow \arg\min_{u \in V(G^*)} sup_{G^*}(u)$;
      $k \leftarrow max\{k, sup_{G^*}(v)\}$;
      $t(v) \leftarrow k$;
      for each $(u,w) \in \{(u,w) \mid u,w \in N_{G^*}(v) \wedge (u,w) \in E\}$ do
          $sup_{G^*}(u) \leftarrow sup_{G^*}(u) - 2$ if $sup_{G^*}(u) > k$;
          $sup_{G^*}(w) \leftarrow sup_{G^*}(w) - 2$ if $sup_{G^*}(w) > k$;
      $E(G^*) \leftarrow E(G^*) - \{(u,v) \mid u \in N_{G^*}(v)\}$;
      $V(G^*) \leftarrow V(G^*) - \{v\}$;
  Return: $t(\cdot)$
```

Property 3. *Given a k-trine with n vertices, the diameter of the k-trine, denoted by d, is at most $\left\lfloor \frac{3n}{3+\lfloor\sqrt{k+0.25}-0.5\rfloor} \right\rfloor - 1$, i.e., $n \geq d + 1 + \lceil \frac{d+1}{3} \rceil \times \lfloor\sqrt{k+0.25}-0.5\rfloor$.*

4 Trine Compution on Static Graphs

4.1 Compute k-Trine

Definition 5. ***k-Trine Computation Problem.*** *Given a graph G and an integer k, compute the k-trine of G.*

We firstly compute the $\lfloor\sqrt{k+0.25}+0.5\rfloor$-core, because it contains the k-trine by Prop. 1 and the k-core computation runs in linear time. Then, we can compute the k-trine by iteratively removing each vertex with insufficient support value.

Alg. 1 shows the pseudo-code of k-trine computation. First, we quickly exclude the non-k-trine vertices by $\lfloor\sqrt{k+0.25}+0.5\rfloor$-core computation and re-construct the adjacency list of each vertex (Line 1). Then, we compute the support of each vertex by triangle counting and store the to-delete vertices in Del (Lines 2-4). Next, we recursively delete vertices with insufficient supports and update supports of other vertices for the deletion, until the remaining subgraph satisfies the definition of k-trine (Lines 5 - 13).
Analysis. The dominated time cost of Alg. 1 is from the triangle enumeration in Lines 3 and 8, which is effectively restricted by the computation of the $\lfloor\sqrt{k+0.25}+0.5\rfloor$-core. Therefore, by using hash table in triangle mantenance, the time complexity of Alg. 1 is $O(m + |E(\lfloor\sqrt{k+0.25}+0.5\rfloor\text{-core})|^{1.5})$.

4.2 Trine Decomposition

Definition 6. ***Trine Decomposition Problem***[2]. *Given a graph G, compute the triness of every vertex in G.*

Due to Obs. 2 and 3, the trine decomposition can be naturally computed by computing the k-trines with an increasing input k. The framework is proposed in [13] for finding the densest subgraph.

[2]The problem is the same as (k,Φ)-core decomposition in [13] and k-clique core decomposition in [11, 40] when the basic unit is the triangle.

To ease the understanding of the maintenance algorithms in Sec. 5, we show the pseudo-code of trine decomposition (named TrineD) in Alg. 2. Let G^* be a subgraph of G and k be the lower bound of triness for vertices in the current G^*. The algorithm initializes with $G^* = G$, $k = 0$ and calculates the support of every vertex in G^* by triangle counting (Lines 1 - 3). Then, the algorithm recursively chooses the vertex v with the smallest $sup_{G^*}(\cdot)$ in G^*, and increases k to $max\{k, sup_{G^*}(v)\}$ (Lines 5 - 6). We assign $t(v) = k$ (Line 7) because i) the vertices with triness less than k have been deleted in previous iterations, ii) G^* now is a subgraph of the k-trine of G that contains v, and iii) $sup_{G^*}(v)$ is no larger than k now. Next, we update the supports of other vertices for the deletion of v (Lines 8-10), and v will be removed from G^* (Lines 11-12).

Since $sup_{G^*}(\cdot)$ is only used to compare with k (Line 6), we do not need to update $sup_{G^*}(\cdot)$ if it is already no larger than k.
Analysis. As discussed in Appx A.3, Alg. 2 runs in $O(m^{1.5})$ time.

5 Trine Maintenance on Dynamic Graphs

Definition 7. ***Trine Maintenance Problem.*** *Given a graph G, the result of trine decomposition on G, and a set of edges E^+ (resp. E^-), the problem of trine maintenance is to update the triness of each vertex in G after inserting E^+ to G (resp. removing E^- from G).*

Inspired by the studys on k-core and k-truss [49, 50], we first show the challenge of the problem by proving the (un)boundedness. (Sec. 5.1). Then, we introduce the edge batch insertion algorithm (Sec. 5.2) and an effective pruning technique (Sec. 5.3). We also propose a batch deletion algorithm for the problem (Sec. 5.4).

5.1 Theoretical Analysis

Definition 8. ***Boundedness.*** *The trine maintenance problem is bounded if and only if there exists at least one locally persistent algorithm that can solve the problem in $O(f(||CHANGE||_c))$ time, where $CHANGE$ is the set of vertices whose trinenesses needs to re-compute, or who are endpoints of $e \in E^+$ (or E^-), $||CHANGE||_c$ denotes the size of $CHANGE$'s c-hop neighbors for a constant positive integer c, and $f(\cdot)$ is a polynomial function.*

Theorem 1. *The trine maintenance problem is bounded for edge removals but is unbounded for edge insertions.*

The unboundedness of edge insertions. To prove this, we can construct a graph G and two specific updates ΔG_1 and ΔG_2 such that (i) inserting either ΔG_1 or ΔG_2 to G results in $O(1)$ changes of triness, i.e., the corresponding $CHANGE$ is of constant size; (ii) inserting both two Δ results in changes different with inserting one; and (iii) the distance between ΔG_1 and ΔG_2 is at least l, where l is not a constant but related to G.

The first point states that assuming the problem is bounded, there should be a locally persistent algorithm that can insert ΔG_1 or ΔG_2 in $O(1)$ time since $O(f(||CHANGE||_c)) = O(1)$. However, because of the second point, when ΔG_2 is inserted, the locally persistent algorithm has to spend extra time to confirm whether ΔG_1 has already been inserted before ΔG_2, even if ΔG_1 is indeed not inserted. Since the third point, the above extra time overhead causes the time complexity of any locally persistent algorithm to insert ΔG_2, denoted as $O(\mathcal{A}(\Delta G_2))$, alone to be at least $O(l) - O(\mathcal{A}(\Delta G_1))$, i.e., $O(f(||CHANGE||_c))$ insertion cannot hold for both $\mathcal{A}(\Delta G_1)$ and $\mathcal{A}(\Delta G_2)$. Thus, the trine maintenance problem is unbounded.

In Appx. A.2, we further give an instance to support the above analysis.

The boundedness of edge removal. We present a batch-edge-removal algorithm in Sec. 5.4 and show it is bounded.

5.2 Edge Insertion

The intuition of the edge-insertion algorithm is transforming the problem into a vertex-order maintenance problem. Specifically, for edge insertions, the propagation of triness change is dependent on the deletion sequence of vertices, i.e. T-order, in TrineD (Alg. 2).

Definition 9. ***T-order.*** *Given a graph G, the T-order, denoted by $\preceq$, is defined as an ordered vertex set $V(G)$ that satistifies the following conditions: (i) $\forall u, v \in V(G)$, $u \preceq v \rightarrow t(u) \leq t(v)$; and (ii) Let $\preceq_v = \{u \mid v \preceq u \vee u = v\}$ be a subset of $V(G)$, then $\forall v \in V(G)$, $sup_{G[\preceq_v]}(v) \leq t(v)$ holds. We may also use $\preceq$ to represent the sequence of vertex ordering satisfying the above conditions.*

The vertex deletion sequence in TrineD is a $\preceq$, and the vertices with equal triness occur consecutively in $\preceq$. Regarding T-order, there are three key notations used in our algorithm:

- $\preceq_v$ denotes a subset of $\preceq$ formed by vertex v and the vertices behind v in $\preceq$, i.e., $\preceq_v = \{u \mid v \preceq u \vee u = v\}$;
- $\Phi_k(G)$ denotes a subset of $\preceq$ formed by every vertex $v \in \preceq$ with $t(v) = k$, i.e., $\Phi_k = \{v \mid v \in \preceq \wedge t(v) = k\}$;
- $rem(v)$ denotes 2 times the number of triangles formed by v and the vertices in $\preceq_v$, i.e., $rem(v) = sup_{G[\preceq_v]}(v)$.

Let G' be the updated G after inserting a set of edges. The initial $\preceq$ may no longer satisfy the definition of T-order in G', since $sup_{G'[\preceq_v]}(v)$ of a vertex v may increase and become larger than $t(v)$ in G'. Therefore, we need to update the T-order $\preceq$ (along with Φ_k, $rem(\cdot)$), which essentially maintains the trinesses. We denote the homonymous notions on the updated G' by $\preceq'$, $t'(\cdot)$ and $rem'(\cdot)$.

The main idea of our algorithm (Alg. 3) is maintaining the trinesses of vertices by maintaining their positions in $\Phi_k(G)$.

Theorem 2. *The trine maintenance problem can be reduced to maintaining the T-order which is to update each $\Phi_k(G)$.*

Algorithm 3: BatchInsertion

```
Input: $G$, $E^+$, T-order $\preceq$, $rem(\cdot)$ : $\forall v \in \preceq, rem(v) = sup_{G[\preceq_v]}(v)$
Output: the triness of each vertex after inserting $E^+$
1  According to Def. 9-12, $\preceq^* \leftarrow \preceq$; $\mathcal{P} \leftarrow \emptyset$; $ext(\cdot), ubr(\cdot) \leftarrow \{0\}$;
2  for each $e = (u, v) \in E^+$ do
3      for each $\triangle(u, v, w) \in G \cup E^+$ do
4          $x \leftarrow$ the vertex in $\{u, v, w\}$ with $x \preceq u \wedge x \preceq v \wedge x \preceq w$;
5          $ext(x) \leftarrow ext(x) + 2$;
6  $v^* \leftarrow$ the first vertex in $\preceq$ s.t. $ext(v^*) \neq 0$;
7  while $v^* \neq tail \vee \mathcal{P} \neq \emptyset$ do
8      $k \leftarrow min\{t(v^*), min_{v \in \mathcal{P}} ubr(v)\}$
       /* Reposition Some Vertices in $\mathcal{P}$ to $\preceq^*$ */
9      for each $v \in \mathcal{P}$ with $ubr(v) \leq k$ do
10         move $v$ from $\mathcal{P}$ to $\Phi^*_k$ at the position before $v^*$;
11         $t(v) \leftarrow k$; $rem(v) \leftarrow ubr(v)$; $ubr(v) \leftarrow 0$;
12         for each $\triangle(u, v, w) \in G'[\preceq^*_v \cup \mathcal{P}]$ do
13             $ubr(u) \leftarrow ubr(u) - 2$ if $u \in \mathcal{P}$;
14             $ubr(w) \leftarrow ubr(w) - 2$ if $w \in \mathcal{P}$;
15             $ext(u) \leftarrow ext(u) - 2$ if $u \in \preceq^*$;
16             $ext(w) \leftarrow ext(w) - 2$ if $w \in \preceq^*$;
17     if $k = t(v^*)$ then
           /* Check $v^*$ for the Pending Set */
18         if $ext(v^*) + rem(v^*) > k$ then
               /* case-1: move $v^*$ from $\preceq^*$ to $\mathcal{P}$ */
19             for each $\triangle(u, v^*, w) \in G[\preceq^*_{v^*}]$ do
20                 $ext(\cdot) \leftarrow ext(\cdot) + 2$ for $u$ and $w$;
21             $ubr(v^*) \leftarrow ext(v^*) + rem(v^*)$;
22             $ext(v^*) \leftarrow 0$; $rem(v^*) \leftarrow 0$;
23             move $v^*$ from $\Phi_k(G)$ to $\mathcal{P}$;
24         else if $ext(v^*) + rem(v^*) \leq k$ then
               /* case-2: $v^*$ stays its position in $\preceq^*$ */
25             $rem(v^*) \leftarrow ext(v^*) + rem(v^*)$, $ext(v^*) \leftarrow 0$;
26             for each $\triangle(u, v^*, w) \in G[\preceq^*_{v^*} \cup \mathcal{P}]$ with
                 $u \in \mathcal{P} \vee w \in \mathcal{P}$ do
27                 Update $ubr(\cdot)$ and $ext(\cdot)$ by Lines 13-16;
28         $v^* \leftarrow$ the next vertex in $\preceq^*$ s.t. $ext(v^*) \neq 0$;
29         Execute Lines 9 - 16 with above $v^*$ if case-2 was invoked;
Return: $t(\cdot)$
```

Proof. Whenever v is moved from $\Phi_k(G)$ to $\Phi_{k'}(G')$ in the T-order, we can update $t(v)$ to k'. Since $\preceq'$ is a vertex deletion sequence of TrineD(G'), the trineness are correctly maintained. □

Let $\preceq^*$ be an intermediate state between $\preceq$ and $\preceq'$ in the maintenance of T-order (Similarly for Φ^*_k). The key points of a fast T-order maintenance are to quickly identify every vertex v that satisfies $sup_{G'[\preceq^*_v]}(v) > t(v)$, move it out of $\preceq^*$, and reposition it to the correct place in $\preceq^*$. The challenge is, in addition to the new triangles produced by E^+, the move of the vertices in front of v to the behind will also affect $sup_{G'[\preceq^*_v]}(v)$.

To overcome the above challenge, we propose a novel framework to maintain the T-order (Alg. 3), which ensures that each vertex will be moved at most once, even if the insertion includes multiple edges. The algorithm first calculates the direct effect of inserted E^+ on

$sup_{G'[\preceq_v^*]}(v)$ by enumerating each edge (Lines 1 - 5). Next, consider that all the to-move vertices will only be moved backward in T-order, the algorithm uses the variable v^* to traverse each vertex in $\preceq^*$ from front to back s.t. a longer prefix of $\preceq^*$ is gradually computed and eventually it becomes $\preceq'$ (Lines 6 - 29).

During the maintenance, the algorithm repeatedly performs two procedures: a) determining whether v^* should be removed from its original position (Lines 18 - 27), and if so, moving it to a pending set $\mathcal{P}$ (Def. 10) (Line 23); and b) determining whether vertices in $\mathcal{P}$ now can be put back into the position before v^* without violating the constraint of T-order (Lines 9 - 16 and 29).

Definition 10. ***Pending Set.*** *Given G and E^+, the algorithm maintains a Pending Set, denoted by $\mathcal{P}$, during the traversal of $\preceq$, which stores the vertices that have been removed from $\preceq$ and pending for being inserted to the correct positions, in the current iteration.*

Check v^* for the Pending Set. The following notion is to determine whether v^* should be removed and put into the pending set.

Definition 11. ***Extra Vertex Support.*** *Given a graph $G' = G \cup E^+$, when the algorithm is pointing to the position of v^*, the Extra Vertex Support of a vertex $v \in \preceq_{v^*}^*$, denoted by $ext(v)$, is defined as $ext(v) = sup_{G'[\preceq_v^* \cup \mathcal{P}]}(v) - rem(v)$.*

Since $rem(v) = sup_{G[\preceq_v]}(v)$, by comparing the definitions of $rem(\cdot)$ and $ext(\cdot)$, it can be seen that $ext(v)$ comes from two sources: a) the triangle formed by v with E^+; and b) the triangle formed by v with the vertices in $\mathcal{P}$. The specific maintenance of $ext(\cdot)$ is described later. When the algorithm completes the reposition from Lines 8 - 16 and then visits Line 17, for the v^* currently visited by the algorithm, we have $rem(v^*) + ext(v^*) = sup_{G'[\preceq_{v^*}^* \cup \mathcal{P}]}(v^*) = sup_{G'[\preceq'_{v^*}]}(v^*)$, because the vertices in $\mathcal{P}$ will behind v^* when they are repositioned into $\preceq^*$. Thus, v^* should be removed from $\preceq^*$ if and only if $rem(v^*) + rex(v^*) > t(v^*)$ (Line 18).

Reposition Some Vertices in $\mathcal{P}$ to $\preceq^*$. The following notion is used to check whether $v \in \mathcal{P}$ can be repositioned before v^*.

Definition 12. ***The Upper Bound of*** $rem'(\cdot)$. *Given a graph $G' = G \cup E^+$, when the algorithm is pointing to the position of v^*, the Upper Bound of $rem'(v)$ for a vertex $v \in \mathcal{P}$, denoted by $ubr(v)$, is defined as $ubr(v) = sup_{G'[\preceq_{v^*}^* \cup \mathcal{P}]}(v)$.*

$ubr(\cdot)$ is an upper bound of $rem'(\cdot)$ since for each v^*, $\forall v \in \mathcal{P}$ we have $\preceq'_v \subseteq (\preceq_{v^*}^* \cup \mathcal{P})$. When $ubr(v) \le k \le t(v^*)$, we can put v into the last position in Φ_k^* and front of v^* since repositioning v here can make $\preceq'$ satisfy the definition of T-order (Lines 9 - 10). k is gradually increased from $min_{v \in \mathcal{P}} ubr(v)$ to $t(v^*)$ to ensure multiple vertices that need to be repositioned are repositioned in the correct order and their $t'(\cdot)$ are maintained correctly. When v is repositoned, we have $rem'(v) = ubr(v)$, since the vertices still in $\mathcal{P}$ will only be put in the positions behind v in later iterations.

Maintain the Above Functions. The initial value of $ext(\cdot)$ is computed from Lines 1-5. Since $\mathcal{P} = \emptyset$ and $\preceq^* = \preceq$ (Line 1), $ext(\cdot)$ contains only the new triangles generated by E^+, i.e. $ext(v) = sup_{G'[\preceq_v]}(v) - sup_{G[\preceq_v]}(v)$. The initial value of $ubr(\cdot)$ is computed from Line 21. Comparing the definitions of $ext(\cdot)$ and $ubr(\cdot)$, it is easy to conclude that $ubr(v^*)$ after v^* is just removed from $\preceq^*$ is equal to $rem(v^*) + ext(v^*)$ before it is removed from $\preceq^*$.

1) When the algorithm moves v^* to $\mathcal{P}$, $ext(\cdot)$ of the vertices in $\preceq_{v^*}$ sharing triangles with v^* should be updated (Lines 19-20), because we assume v^* will be moved to a position behind its neighbors;

2) When the algorithm moves v from $\mathcal{P}$ to the position before v^* in $\preceq$, i.e., the final position of v, for the vertices that share triangles with v, the $ext(\cdot)$ of the vertices in $\preceq_{v^*}$ and the $ubr(\cdot)$ of the vertices in $\mathcal{P}$ should be updated (Lines 12-16).

Furthermore, if the current v^* pointed by the algorithm satisfies $rem(v^*) + ext(v^*) \le k$ (Line 24), some triangles for $ext(v^*)$ may contain the vertices in $\mathcal{P}$. In this case, these triangles should be eliminated from $ubr(\cdot)$ of the corresponding vertices (Lines 26-27) before the algorithm points to the next vertex.

A running example of the algorithm is given in Appx. B.1.

Analysis. Let $\mathcal{P}_{his}$ be the set of vertices that were moved into $\mathcal{P}$ throughout the whole process (regardless of the move-out) and $||\mathcal{P}_{his}||_c$ be the number of c-hop neighbors of $\mathcal{P}_{his}$. The time complexity of Alg. 3 is $O((||\mathcal{P}_{his}||_1 \cup ||E^+||_1) log(||\mathcal{P}_{his}||_1 \cup ||E^+||_1) + ||\mathcal{P}_{his}||_2)$. The proof is given in Appx. A.3.

Since the trinesses of some vertices in $\mathcal{P}_{his}$ do not change, we have $E^+ \cup P_{his} \nsubseteq CHANGE_c$ for any constant c. Thus, Alg. 3 is unbounded with $||CHANGE||_c$, consistent with Thm. 1.

5.3 Pruning for Edge Insertion

Delaying Impact. Alg. 3 may have redundant computations in enumerating the triangles of each vertex. As the final position of v^* is not determined during the Pending Set check, the algorithm enumerates v^*'s every neighbor u and 2-hop neighbor w satisfying $v^* \preceq u, w$ to determine whether $\triangle(u, v^*, w)$ exists (Lines 17 - 18) and update their $ext(\cdot)$. Similarly, when v moves from $\mathcal{P}$ to $\preceq^*$, a process is used to eliminate the impact of v on $ubr(\cdot)$ or $ext(\cdot)$ for every $u \in N_{G'}(v) \cap (\preceq_v^* \cup \mathcal{P})$ (Lines 12 - 16). Thus, we propose a novel pruning technique, named Delaying Impact, to reduce the enumeration of triangles by delaying the update of $ext(\cdot)$ for a part of v^*'s neighbors when the algorithm moves v^* from $\preceq^*$ to $\mathcal{P}$.

The pseudo-code (Algo 5) and running example of the algorithm equipped with the Delaying Impact pruning is shown in Appx. B.

When the algorithm moves v^* from $\preceq^*$ to $\mathcal{P}$, it does not immediately update $ext(\cdot)$ for all neighbors (which belong to $\preceq_{v^*}^*$ and share triangles with v^*), but only those belong to $\Phi_{t(v^*)}^*$. In contrast, the effect for other neighbors (their $t(\cdot) > t(v^*)$) is updated with a delay. That is, only when k is changed, the algorithm calculates $\mathcal{P}$'s effect on the $ext(\cdot)$ of their neighbors belonging to Φ_k^*. In this way, only the vertices belonging to $\preceq_{v^*}^* \cap (\bigcup_{k \le t'(u^*)} \Phi_k(G))$, as well as the triangles they share with v^*, are enumerated in the combo of case-1 of Pending Set Check and Delay Impact. Similarly, each time v is moved from $\mathcal{P}$ to $\preceq^*$, the algorithm only enumerates the vertices belonging to $\Phi_{t'(v)}(G')$ and the triangles they share with v to eliminate the impact of v on $ext(\cdot)$.

Even though we delay updating a part of $ext(\cdot)$, the whole algorithm still executes correctly, since whenever the algorithm executes to Pending Set Check, for v^* now, $ext(v^*)$ complies with Def. 11. This ensures the accuracy of $ubr(\cdot)$ and algorithmic flow.

5.4 Edge Removal

Trine maintenance against edge removals is relatively simpler because the algorithm can be bounded (Sec. 5.1). Unlike the insertion

Algorithm 4: BatchRemoval

```
   Input: $G$, $E^-$, T-order $\preceq$, $ts(\cdot)$, $rem(\cdot)$
   Output: the triness of each vertex after removing $E^-$
1  $Q \leftarrow$ an empty queue;
2  for each $e = (u, v) \in E^-$ do
3      for each $\triangle(u, v, w) \in G$ do
4          for each $x \in u, v, w$ do
5              $ts(x) \leftarrow ts(x) - 2$ if $t(x) \leq min(t(u), t(v), t(w))$
6              if $ts(x) < t(x) \wedge x \notin Q$ then
7                  $Q.push(x)$;
8      $G \leftarrow G \setminus \{e\}$;
9  while $Q \neq \emptyset$ do
10     $v = Q.top(); Q.pop()$;
       /* $|\triangle_v^k| \leftarrow sup_{G[\{v\} \cup \{x | t(x) \geq k\}]}(v)$ */
11     $t_{new} \leftarrow \max\{k \mid |\triangle_v^k| \geq k\}$;
12     for $\triangle(u, v, w) \in G[\{x \mid t(x) > t_{new}\}]$ with
         $t(u) \leq t(v) \vee t(w) \leq t(v)$ do
13         $ts(u) \leftarrow ts(u) - 2$ if $t(u) \leq t(v)$;
14         $ts(w) \leftarrow ts(w) - 2$ if $t(w) \leq t(v)$;
15         $Q.push(u)$ if $ts(u) < t(u) \wedge u \notin Q$;
16         $Q.push(w)$ if $ts(w) < t(w) \wedge w \notin Q$;
17     move $v$ to the tail of $\Phi_{t_{new}}(G)$;
18     $t(v) \leftarrow t_{new}(v)$;
19     $ts(v) \leftarrow |\triangle_v^{t(v)}|$;
   Return: $t(\cdot)$
```

algorithm, we can directly identify the vertices with triness changes by $ts(\cdot)$ defined in the following, without relying on T-order.

Definition 13. ***Vertex Support in Trine Subgraph.*** *Given a graph G, the vertex support in the trine subgraph of a vertex $v \in V(G)$, denoted by ts(v), is defined as $ts(v) = sup_{T_{t(v)}}(v)$.*

Algorithm Overview. When edges in G are removed, some triangles that contain these edges inevitably disappear. Thus, we enumerate these triangles, maintain $ts(\cdot)$ based on Def. 13, and push vertices into a queue Q if $ts(\cdot) < t(\cdot)$ (Lines 1-8). These vertices await subsequent modifications to $t(\cdot)$. Next, for each vertex $v \in Q$, we determine its new triness t_{new} based on $\triangle_v^k = sup_{G[\{v\} \cup \{x|t(x) \geq k\}]}(v)$ and maintain the new $ts(v)$ (Lines 11, 18 and 19). Additionally, due to the changing of $t(v)$, $ts(\cdot)$ of a vertex $u \in N_G(v) \cap \bigcup_{t_{new} < k \leq t(v)} \Phi_k^*$ that forms a triangle with v may also decrease, which may lead to an update for $t(u)$. The algorithm adds u to Q if $t(u)$ needs an update, i.e., $ts(u) < t(u)$ (Lines 12-16).
Analysis. Alg. 4 runs in $O(||CHANGE||_2 \cdot \max_{v \in CHANGE} ||v||_2)$ time, which validates that the trine maintenance problem is bounded for edge removals. The proof is given in Appx. A.3.

6 Experimental Evaluation

Datasets. 10 datasets are used (see Tab. 2), in which Yelp is from [1] Wiki and Facebook are from Konect[18] and the others are from SNAP [19]. We use SN, CN, and WG to denote social network, comment network, and web graph, respectively. We remove self-loops and multiple edges. For multiple edges in temporal graphs, we only reserve the earliest timestamp. We use k_{max}^{core} to represent the maximal k for k-core, and k_{max}^{truss}, k_{max}^{trine} correspondingly.

Table 2: Statistics of Datasets

Dataset	n	m	type	temporal	k_{max}^{core}	k_{max}^{truss}	k_{max}^{trine}
Superuser(SU)	192K	715K	CN	Yes	61	35	2228
Facebook(FA)	64K	817K	SN	Yes	52	36	1770
Youtube(YO)	1.1M	3.0M	SN	No	51	19	1104
Google(GO)	875K	4.3M	WG	No	44	44	1890
Wiki(WI)	2.8M	8.1M	CN	Yes	210	85	22010
Yelp(YE)	2.0M	9.5M	SN	No	188	103	18214
Pokec(PO)	1.6M	22M	SN	No	47	29	1112
Stackoverflow(ST)	2.6M	28M	CN	Yes	198	79	14756
Orkut(OR)	3.1M	117M	SN	No	253	78	14234
Friendster(FR)	65.6M	1.8G	SN	No	304	129	16510

Algorithms. We evaluate k-core decomposition [5], k-truss decomposition [41], k-trine computation (*Comp. k-trine*, Alg. 1), trine decomposition (*TrineD*, Alg. 2), and trine maintenance (*BatchInsertion* by Alg. 3, *BatchInsertion+* by Alg. 5, and *BatchRemoval* by Alg. 4). For fairness, we use the same triangle computation method in all the algorithms.
Executions. The algorithms are implemented using C++ and compiled with GCC (version 11.4.0) under O3 optimization. The computer has an Intel Xeon 2.1GHz CPU and 512G of RAM.

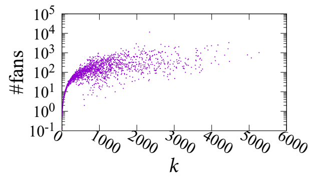
6.1 Effectiveness Results

Exp-1: Statistical Results. Fig. 2 shows the effectiveness of triness in modeling user activities in the Yelp network. Each point in the figure shows the average number of fans (resp. reviews) of the users with the same degree, coreness, trussness, and triness, respectively. It was validated that the correlation of coreness with user activity level is the best, compared with degree and centrality metrics [45]. Our result shows that triness provides a finer granularity than coreness and trussness. The disturbance of triness is due to the over-fine granularity and the sensitivity to small data samples in each value. Therefore, we recommend using a slightly rougher granularity on modeling/estimating user engagement.

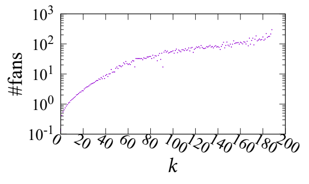
We also tested the Pearson correlation coefficient between different Metrics and node activity values (#Fans or #Reviews). The result in Tab. 3 shows that a) triness is more correlated with node activity value than degree; b) If we divide the whole range of all the metrics to k_{max}^{truss}, i.e., using the same granularity for a fair comparison, triness is more correlated with #Fans than coreness, indicating its potential in user activity analysis. Therefore, We can design different strategies according to the triness values s.t. the overall user engagement can be improved.

Beside, we report the average degree (d_{avg}) of the $(k-1)$-core, k-truss, and $(k-1)(k-2)$-trine in Tab. 4, respectively, where $k = k_{max}^{truss}$ (the parameter is decided by Prop. 1 and 2). The bold values are the best. We can observe that: a) for all datasets, d_{avg} of trine is no smaller than d_{avg} of truss; and b) for datasets other than Facebook, Google and Friendster, d_{avg} of trine is the highest.
Exp-2: Case Study on DBLP. We evaluate the k-trine of the DBLP collaboration network, compared with k-truss. We extract the network of all authors where an edge connects two authors if they collaborate on at least one KDD paper with no larger than 6 authors.

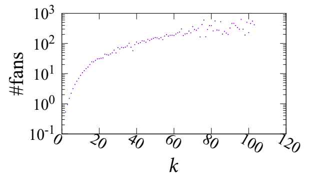
We find there are 12 authors in the 20-trine who are excluded by the 6-truss (note that 6-truss is always a subgraph of 20-trine). Fig. 3 depicts the 12 authors by the red nodes, and their neighbors in



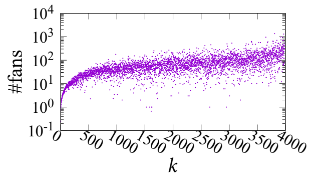
(a) Degree



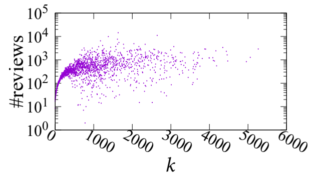
(b) Coreness



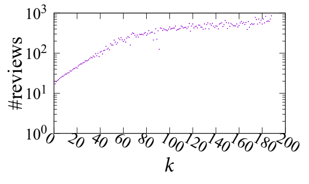
(c) Trussness



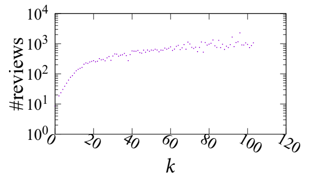
(d) Triness



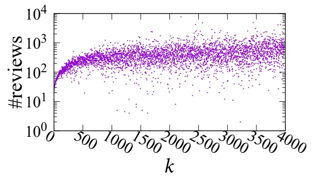
(e) Degree



(f) Coreness



(g) Trussness



(h) Triness

Figure 2: #Fans and #Review in Yelp with Different Metrics

Table 3: Result of Pearson Correlation Coefficient ρ

Metric	Different granularities		Same granularity	
	#Fans	#Reviews	#Fans	#Reviews
Degree	0.454	0.358	0.695	0.576
Coreness	0.921	0.958	0.910	0.975
Trussness	0.872	0.865	0.872	0.865
Triness	0.551	0.382	0.925	0.884

20-trine by the yellow nodes. We find that the 12 authors are well connected with the yellow nodes but excluded by the 6-truss. A similar result is observed in the SIGMOD subgraph in Fig. 4.

For both KDD and SIGMOD cases, we find the average degree (resp. the average number of publications in the corresponding conference) of the authors in the 20-trine is higher than that observed in the 6-truss: 6.75 vs 6.72 in KDD, and 7.26 vs 7.10 in SIGMOD (resp. 2.65 vs 2.64 in KDD, and 3.91 vs 3.80 in SIGMOD). These indicate that $(k-1)(k-2)$-trine may further contain some well-engaged nodes with more weak ties compared with k-truss.

Table 4: Average Degrees of Different Subgraphs ($k = k_{max}^{truss}$)

Dataset	$(k-1)$-core	k-truss	$(k-1)(k-2)$-trine
SU	89.55	61.85	**105.50**
FA	**81.65**	50.62	73.30
YO	56.67	63.04	**82.65**
GO	**48.79**	46.71	46.71
WI	276.27	208.48	**350.49**
YE	284.61	149.47	**334.22**
PO	62.76	30.75	**64.84**
ST	237.96	143.24	**354.91**
OR	240.63	110.71	**426.95**
FR	**333.79**	130.94	138.96

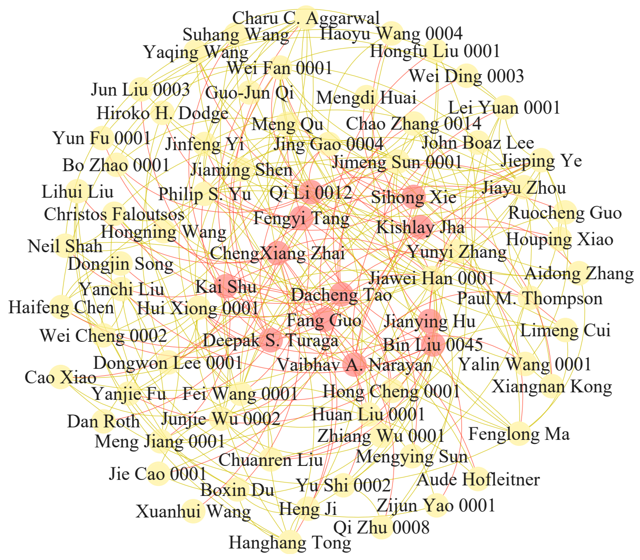


Figure 3: A Case Study on KDD Subgraph of DBLP (induced by the red nodes in 20-trine\6-truss and their neighbors)

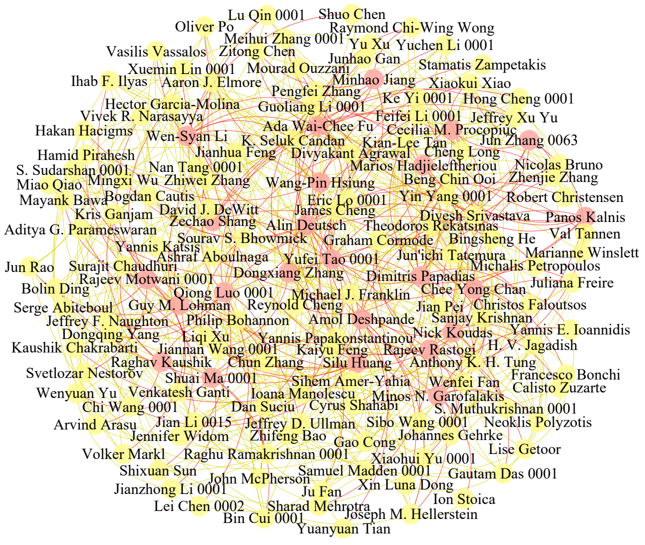


Figure 4: A Case Study on SIGMOD Subgraph of DBLP

Table 5: Decomposition Time (in second)

Dataset	*Core*	*Truss*	*Trine*	*Trine$_{with\ index}$*
SU	0.072	1.723	1.190	1.429
FA	0.048	1.933	0.546	0.850
YO	0.724	10.605	4.707	5.819
GO	1.289	11.891	5.035	7.041
WI	1.236	281.70	136.40	148.47
YE	1.964	41.415	16.520	23.626
PO	6.466	153.21	27.519	38.221
ST	6.987	1020.0	124.18	147.79
OR	38.098	2394.5	333.95	427.43
FR	1177.2	27306.1	14070.7	16088.8

6.2 Efficiency Results

Exp-3: Time Cost of Decompositions. Tab. 5 reports the time cost of core, truss, and trine decompositions on all the datasets. The time cost of core decomposition is linear which is certainly smaller than truss and trine decomposition. Note that our target is not to outperform k-core in efficiency and the k-core runtime is reported as a reference. In the table, trine decomposition is faster than truss decomposition, e.g., it is 7.2× faster on orkut. This is because k-trine is a vertex-oriented model s.t. triangle enumeration on some edges can be pruned. We also evaluate the trine decomposition with indexing, i.e., T-order, $rem(\cdot)$, and $ts(\cdot)$ for the input of trine maintenance. The cost of initializing these functions is minor.

Exp-4: Time Cost of Computation and Maintenance. We evaluate the impact of ΔG on the efficiency of maintenance algorithms. Given an integer b as the number of edges to insert/remove, and a dataset G, ΔG is formed by the b edges in G with the largest timestamps if G has timestamps, or randomly b edges if G has no timestamps. We first evaluate BatchRemoval by deleting ΔG from G, and then evaluate BatchInsertion/BatchInsertion+ by inserting ΔG back to recover G, which produces one plot for each algorithm in Fig. 5. For comparison, we also report the runtime of TrineD and Comp. k-trine where $k_{max} = k^{trine}_{max}$ as given in Tab. 2.

Fig. 5 shows that a) the k-trine computation algorithm is faster than the decomposition (TrineD) as k-trine can be computed on the $\lfloor\sqrt{k+0.25}+0.5\rfloor$-core; b) BatchInsertion+ runs in less time than BatchInsertion by 2 − 4×, which validates the effectiveness of the pruning technique in Sec. 5.3; and c) The maintenance algorithms are efficient and scalable, e.g., they run in much less time than TrineD when $|\Delta G| \leq 5000$, the speedup ratio is 1 − 3 orders of magnitude for $|\Delta G| = 500$, and processing 5000 edges only costs 10 - 20x time of processing 500 edges.

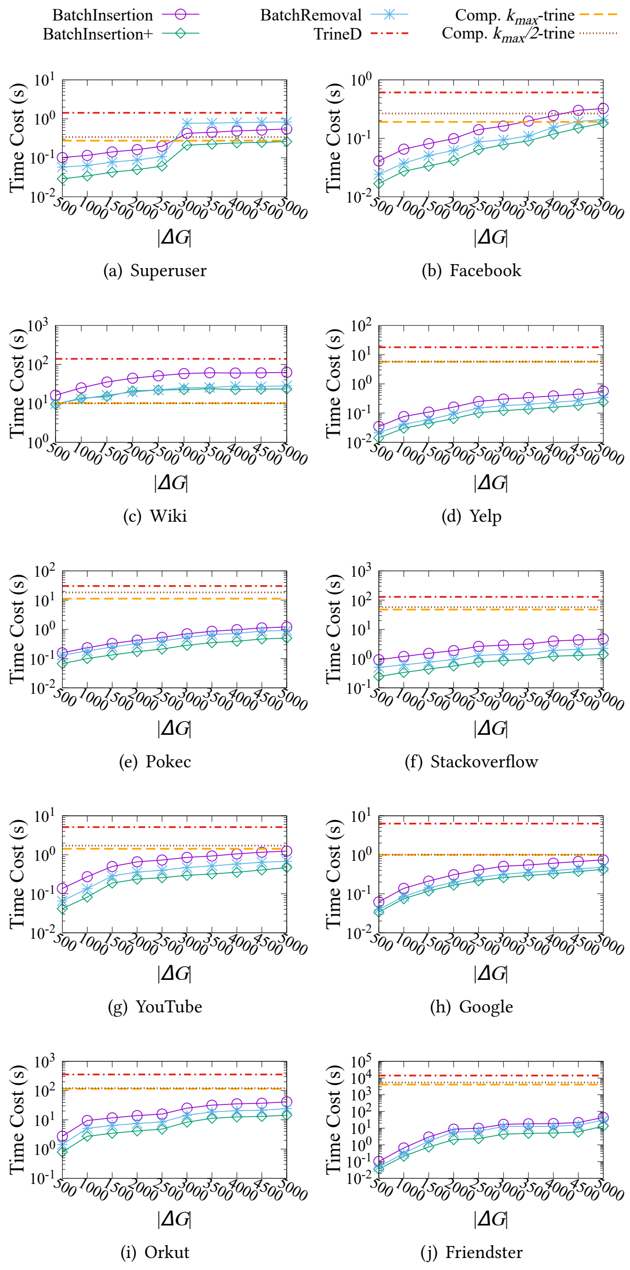


Figure 5: Maintenance Time with Different $|\Delta G|$

7 Conclusion

This paper introduces and studies the k-trine cohesive subgraph and efficient algorithms for computing k-trine on static and dynamic graphs. We analyze the properties of k-trine to optimize the algorithms. Extensive experiments on large real-world networks validate the effectiveness of k-trine and the efficiency of our algorithms. For future studies, it is interesting to study parallel algorithms, and the hierarchical structure formed by different k-trines.

Acknowledgments

This work is partially supported by the National Natural Science Foundation of China (U2436208, 62372129, 62302294), the Guangdong S&T Programme under Grant 2024B0101010002, and the Guangdong Basic and Applied Basic Research Foundation (2024A1515011501, 2023A1515012603). We thank the anonymous reviewers for their insightful suggestions.

References

[1] 2022. Yelp Open Dataset. https://www.yelp.com/dataset.

[2] E. A. Akkoyunlu. 1973. The Enumeration of Maximal Cliques of Large Graphs. *SIAM J. Comput.* 2, 1 (1973), 1–6.

[3] Wen Bai, Yadi Chen, and Di Wu. 2020. Efficient temporal core maintenance of massive graphs. *Inf. Sci.* 513 (2020), 324–340.

[4] A.L Barabási, H Jeong, Z Néda, E Ravasz, A Schubert, and T Vicsek. 2002. Evolution of the social network of scientific collaborations. *Physica A: Statistical Mechanics and its Applications* 311, 3 (2002), 590–614.

[5] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[6] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two Simplified Algorithms for Maintaining Order in a List. In *Algorithms - ESA 2002*, Vol. 2461. 152–164.

[7] Ed Bullmore and Olaf Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature reviews neuroscience* 10, 3 (2009), 186–198.

[8] Lijun Chang and Lu Qin. 2019. Cohesive Subgraph Computation Over Large Sparse Graphs. In *ICDE*. 2068–2071.

[9] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[10] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National security agency technical report* 16 (2008), 3–1.

[11] Maximilien Danisch, Oana Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*. 589–598.

[12] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2022. *Cohesive Subgraph Search Over Large Heterogeneous Information Networks.*

[13] Yixiang Fang, Kaiqiang Yu, Reynold Cheng, Laks V. S. Lakshmanan, and Xuemin Lin. 2019. Efficient Algorithms for Densest Subgraph Discovery. *Proc. VLDB Endow.* 12, 11 (2019), 1719–1732.

[14] Eric Gilbert and Karrie Karahalios. 2009. Predicting tie strength with social media. In *SIGCHI*. 211–220.

[15] Priya Govindan, Chenghong Wang, Chumeng Xu, Hongyu Duan, and Sucheta Soundarajan. 2017. The k-peak Decomposition: Mapping the Global Structure of Graphs. In *WWW*. 1441–1450.

[16] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[17] Jon M Kleinberg, Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew S Tomkins. 1999. The web as a graph: Measurements, models, and methods. In *COCOON'99*. 1–17.

[18] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW '13*. 1343–1350.

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

[20] Rong-Hua Li, Jeffrey Xu Yu, and Rui Mao. 2014. Efficient Core Maintenance in Large Dynamic Graphs. *TKDE* 26, 10 (2014), 2453–2465.

[21] Longlong Lin, Pingpeng Yuan, Rong-Hua Li, Chun-Xue Zhu, Hongchao Qin, Hai Jin, and Tao Jia. 2024. QTCS: Efficient Query-Centered Temporal Community Search. *Proc. VLDB Endow.* 17, 6 (2024), 1187–1199.

[22] Boge Liu, Long Yuan, Xuemin Lin, Lu Qin, Wenjie Zhang, and Jingren Zhou. 2019. Efficient (α, β)-core Computation: an Index-based Approach. In *The World Wide Web Conference*. 1130–1141.

[23] Quanquan C. Liu, Jessica Shi, Shangdi Yu, Laxman Dhulipala, and Julian Shun. 2022. Parallel Batch-Dynamic Algorithms for k-Core Decomposition and Related Graph Problems. In *SPAA*. ACM, 191–204.

[24] Omar Lizardo. 2024. Theorizing the concept of social tie using frames. *Soc. Networks* 78 (2024), 138–149.

[25] Zhao Lu, Yuanyuan Zhu, Ming Zhong, and Jeffrey Xu Yu. 2022. On Time-optimal (k, p)-core Community Search in Dynamic Graphs. In *ICDE*. 1396–1407.

[26] Fragkiskos D. Malliaros, Christos Giatsidis, Apostolos N. Papadopoulos, and Michalis Vazirgiannis. 2020. The core decomposition of networks: theory, algorithms and applications. *VLDB J.* 29, 1 (2020), 61–92.

[27] Fragkiskos D. Malliaros and Michalis Vazirgiannis. 2013. To stay or not to stay: modeling engagement dynamics in social graphs. In *CIKM*. 469–478.

[28] David W. Matula and Leland L. Beck. 1983. Smallest-Last ing and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[29] Atsushi Miyauchi, Tianyi Chen, Konstantinos Sotiropoulos, and Charalampos E. Tsourakakis. 2023. Densest Diverse Subgraphs: How to Plan a Successful Cocktail Party with Diversity. In *KDD*. 1710–1721.

[30] Flaviano Morone, Gino Del Ferraro, and Hernán A Makse. 2019. The k-core as a predictor of structural collapse in mutualistic ecosystems. *Nat. Phys.* 1 (2019), 95.

[31] Lu Qin, Rong-Hua Li, Lijun Chang, and Chengqi Zhang. 2015. Locally Densest Subgraph Discovery. In *KDD*. 965–974.

[32] Garry Robins and Philippa Pattison. 2001. Random graph models for temporal processes in social networks. *J Math Sociol* 25, 1 (2001), 5–41.

[33] Ahmet Erdem Sariyüce, Bugra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V. Çatalyürek. 2016. Incremental k-core decomposition: algorithms and evaluation. *VLDB J.* 25, 3 (2016), 425–447.

[34] Ahmet Erdem Saríyüce, Buğra Gedik, Gabriela Jacques-Silva, Kun-Lung Wu, and Ümit V Çatalyürek. 2013. Streaming algorithms for k-core decomposition. *Proc. VLDB Endow.* 6, 6 (2013), 433–444.

[35] Stephen B Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269–287.

[36] Yingxia Shao, Lei Chen, and Bin Cui. 2014. Efficient cohesive subgraphs detection in parallel. In *SIGMOD*. 613–624.

[37] Kijung Shin, Tina Eliassi-Rad, and Christos Faloutsos. 2016. CoreScope: Graph Mining Using k-Core Analysis - Patterns, Anomalies and Algorithms. In *ICDM*. 469–478.

[38] Siyi Teng, Jiadong Xie, Fan Zhang, Can Lu, Juntao Fang, and Kai Wang. 2024. Optimizing Network Resilience via Vertex Anchoring. In *WWW*. 606–617.

[39] Anxin Tian, Alexander Zhou, Yue Wang, and Lei Chen. 2023. Maximal D-Truss Search in Dynamic Directed Graphs. *Proc. VLDB Endow.* 16, 9 (2023), 2199–2211.

[40] Charalampos E. Tsourakakis. 2015. The K-clique Densest Subgraph Problem. In *WWW*. 1122–1132.

[41] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proc. VLDB Endow.* 5, 9 (2012), 812–823.

[42] Na Wang, Dongxiao Yu, Hai Jin, Chen Qian, Xia Xie, and Qiang-Sheng Hua. 2017. Parallel Algorithm for Core Maintenance in Dynamic Graphs. In *ICDCS*. 2366–2371.

[43] Stanley Wasserman and Katherine Faust. 1994. *Social network analysis: Methods and applications*. Vol. 8.

[44] Fan Zhang, Haicheng Guo, Dian Ouyang, Shiyu Yang, Xuemin Lin, and Zhihong Tian. 2024. Size-Constrained Community Search on Large Networks: An Effective and Efficient Solution. *TKDE* 36, 1 (2024), 356–371.

[45] Fan Zhang, Qingyuan Linghu, Jiadong Xie, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2023. Quantifying Node Importance over Network Structural Stability. In *KDD*. 3217–3228.

[46] Fan Zhang, Long Yuan, Ying Zhang, Lu Qin, Xuemin Lin, and Alexander Zhou. 2018. Discovering Strong Communities with User Engagement and Tie Strength. In *DASFAA*. 425–441.

[47] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2017. When Engagement Meets Similarity: Efficient (k, r)-Core Computation on Social Networks. *PVLDB* 10, 10 (2017), 998–1009.

[48] Ying Zhang, Lu Qin, Fan Zhang, and Wenjie Zhang. 2019. Hierarchical Decomposition of Big Graphs. In *ICDE*. 2064–2067.

[49] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *SIGMOD*. 1024–1041.

[50] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *ICDE*. 337–348.

A Theoretical Proofs

A.1 The Properties of k-Trine

Proof of Prop. 1. Given a k-trine subgraph T_k, according to Lemma 1 and Obs. 3, we have $\forall v \in T_k, deg_{T_k}(v) \times (deg_{T_k}(v) - 1) \geq t(v) \geq k$, i.e., $\forall v \in T_k, deg_{T_k}(v) \geq \sqrt{k + 0.25} + 0.5$. Thus, k-trine of G is a subgraph of the $(\sqrt{k + 0.25} + 0.5)$-core of G.

Proof of Prop. 2. Given a k-truss S, we have i) S is also a $(k-1)$-core [10], i.e., $\forall v \in V(S), deg_S(v) \geq k - 1$, and ii) $\forall e \in S, sup_S(e) \geq k - 2$. Thus, $\forall v \in V(S), sup_S(v) \geq (k-2) \times (k-1)$, i.e., each k-truss is a subgraph of the $(k-2) \times (k-1)$-trine of G.

Proof of Prop. 3. Let T_k be the k-trine, and $v_0 \in V(T_k)$ be an endpoint of one of the longest short paths in T_k. We can divide $V(T_k)$ into $d + 1$ parts (i.e., $V_0, ..., V_d$) where V_i contains all vertices whose distance from v_0 is i. Based on above definitions, we have $\forall v \in V_i, N_{T_k}(v) \subseteq V_{i-1} \cup V_i \cup V_{i+1} \backslash \{v\}$.

Next, we use induction to prove the theorem holds for any $d \geq 1$.

Since, for $d = 1, 2, 3$, the above property holds obviously, we prove if the property holds at $d = x \geq 1$, it also holds at $d = x+3 \geq 4$.

Assume that the property holds at $d = x$ but does not hold at $d = x + 3$, i.e, one can not construct a T_k'', which diameter is x and satisfies $|T_k''| = \sum_{i=0}^{x} |V_i''| < x + 1 + \lceil \frac{x+1}{3} \rceil \cdot \lfloor \sqrt{k + 0.25} - 0.5 \rfloor$ but can construct a T_k', which diameter is $x + 3$ and satisfies $|T_k'| = \sum_{i=0}^{x+3} |V_i'| < x + 4 + \lceil \frac{x+4}{3} \rceil \cdot \lfloor \sqrt{k + 0.25} - 0.5 \rfloor$.

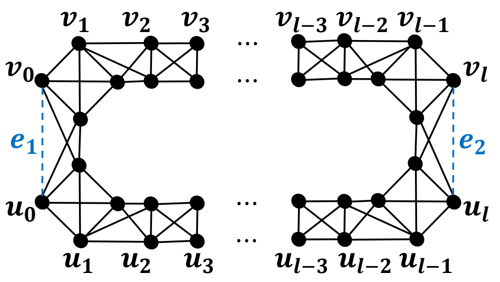


Figure 6: A Graph for Proving Unboundedness

First, it is obviously that if we can construct a T'_k that satisfies the above conditions, T'_k can further satisfy $|V'_0| = |V'_{x+3}| = |V'_2| = |V'_{x+1}| = 1$ and $|V'_1| = |V'_{x+2}| = \lceil\sqrt{k+0.25}+0.5\rceil$, i.e., $|V'_0| + |V'_1| + |V'_2| = |V'_{x+1}| + |V'_{x+2}| + |V'_{x+3}| = \lceil\sqrt{k+0.25}-0.5\rceil + 3$. And then, let $(V'_{i-1}, V'_i, V'_{i+1}) = \arg\min_{(V'_{i-1},V'_i,V'_{i+1})} |V'_{i-1}| + |V'_i| + |V'_{i+1}|$.

i) If $i-1 = 0 \vee i+1 = x+3$, since $|V'_0| + |V'_1| + |V'_2| = |V'_{x+1}| + |V'_{x+2}| + |V'_{x+3}| = \lceil\sqrt{k+0.25}-0.5\rceil + 3$, $\forall i \in [1, x+2]$, $|V'_{i-1}| + |V'_i| + |V'_{i+1}| \geq \lceil\sqrt{k+0.25}-0.5\rceil + 3$. Thus for $(x+4) \mod 3 = 0$, $n \geq \frac{x+4}{3} \cdot (|V'_0| + |V'_1| + |V'_2|) = x + 4 + \lceil\frac{x+4}{3}\rceil \cdot \lceil\sqrt{k+0.25}-0.5\rceil \geq x + 4 + \lceil\frac{x+4}{3}\rceil \cdot \lfloor\sqrt{k+0.25}-0.5\rfloor$. It is easy to show that the above conclusion also holds for $(x+4) \mod 3 \neq 0$. Therefore, we cannot construct T'_k that satisfies the assumption in this case.

ii) Then we continue our discussion with the minimal tried, $(V'_{i-1}, V'_i, V'_{i+1})$, satisfies $i-1 \neq 0 \wedge i+1 \neq x+3$. Due to its minimality, we have $V'_{i+1} \leq V'_{i-2}$ and $V'_{i-1} \leq V'_{i+2}$. Further, by deleting V'_{i-1}, V'_i and V'_{i+1} from T'_k and connect each vertex in V'_{i-2} and V'_{i+2}, we can construct a graph of diameter x which is still a k-trine, call it T''_k. In addition, since $\forall v \in V'_i$ can only obtain $sup_{T'_k}(v)$ by forming triangles with vertices in $V'_{i-1} \cup V'_i \cup V'_{i+1} \backslash \{v\}$, we have $|V'_{i-1} \cup V'_i \cup V'_{i+1}| \geq \lfloor\sqrt{k+0.25}-0.5\rfloor + 3$. This resulted in $|T''_k| = |T'_k| - |V'_{i-1} \cup V'_i \cup V'_{i+1}| < x + 1 + \lceil\frac{x+1}{3}\rceil \cdot \lfloor\sqrt{k+0.25}-0.5\rfloor$ which contradicts the assumption that the property holds when $d = x$. Thus, We have proved that T'_k can be constructed if and only if T''_k can be constructed, i.e., if the property holds at $d = x \geq 1$, it also has at $d = x + 3 \geq 4$.

Therefore, the property holds for any possible values of k and d.

A.2 Unboundedness of Edge Insertion

Fig. 6 conforms to the description in Sec. 5.1, where e_1 and e_2 correspond to ΔG_1 and ΔG_2 respectively. First, the triness of all vertices in G is 8. Moreover, when inserting e_1 into G, none of the vertices' triness changes. For the insertion of e_2, the situation is similar to e_1. However, when we insert both e_1 and e_2, trinesses of all vertices in G become 12.

Assume that there exists a bounded locally persistent insertion algorithm $\mathcal{A}$ for trine maintenance, i.e., there exists a polynomial function f and a const c such that for any G and ΔG, $\mathcal{A}$ runs in $O(f||CHANGE||_c))$ time. Let $trace(G, \Delta G)$ denote the sequence of vertices $\mathcal{A}$ visits when ΔG is inserted. Under our assumption, we have $O(trace(G, e_1)) = O(trace(G, e_2)) = O(1)$. Since whether e_1 has been inserted makes the result of the insertion of e_2 different, there exists a vertex x in $trace(G, e_1)$, $trace(G + e_1, e_2)$ and $trace(G, e_2)$ such that $\mathcal{A}$ can not recognizes whether e_1 is inserted until it traverses x and gets the message. However, since $O(trace(G, e_1)) = O(1)$, x must be the $c - hop$ neighbor of u_0 or v_0 and the length of the path between x and u_l (or v_l) is $O(l)$ which means $O(trace(G, e_2)) = O(l)$ and contradicts the first point. In summary, the trine maintenance problem is unbounded for edge insertions under the model of locally persistent algorithms.

A.3 Complexity Analysis

TrineD Algorithm. To find the vertex with the smallest $sup_{G^*}(\cdot)$ in G^*, we can use a bin sort to sort the vertices in $O(m)$ time. As the core decomposition algorithm in [5], we maintain the bin-sort order once sup_{G^*} changes. For Lines 2-3, we employ the SOTA triangle listing algorithm with time complexity of $O(|E| \cdot \alpha(G))$, where $\alpha(G)$ is the arboricity of G, and $|E| \cdot \alpha(G)$ has a bound of $O(|E|^{1.5})$ [9]. Resembling the truss decomposition algorithm in [41], a hash table is constructed to store $E(G)$ for $sup_{G^*}(\cdot)$ maintenance. Thus, the time complexity of Lines 5 - 12 is $O(|E|^{1.5})$.

In summary, the time complexity of Alg. 2 is $O(|E|^{1.5})$.

BatchInsertion Algorithm. We follow the structure of maintaining $\preceq$ in [6], making the operations of modifying $\preceq$ and comparing the order of vertices both $O(1)$. Thus Time complexity of Alg. 3 is mainly dominated by maintaining vertices where $ext(\cdot) \neq 0$, and the triangle enumeration operations during maintenance.

Lemma 2. *If v^* causes the algorithm to branch into Lines 24 - 27, there must exist a vertex $u \in N_{G'}(v^*)$, s.t. $u \in \mathcal{P} \vee (u, v^*) \in E^+$.*

First, Lines 1 - 5 enumerate triangles by traversing 1-hop neighbors of E^+, such that the time complexity is $O(||E^+||_1)$. Next, Lines 12 - 16 and 19 - 20 enumerate triangles containing $v \in \mathcal{P}_{his}$, such that the time complexity is $O(||\mathcal{P}_{his}||_2)$. Then, as for Lines 26 - 27, we enumerate $u \in \mathcal{P} \cap N_{G'}(v)$ first since lemma 2 holds, and then enumerate only the triangles containing (u, v), making the time complexity $O(||\mathcal{P}_{his}||_2)$. As for maintaining vertices where $ext(\cdot) \neq 0$, since lemma 2 holds, all the elements that once hold $ext(\cdot) \neq 0$ belong to $||\mathcal{P}_{his}||_1 \cup ||E^+||_1$. The time complexity is $O((||\mathcal{P}_{his}||_1 \cup ||E^+||_1) log(||\mathcal{P}_{his}||_1 \cup ||E^+||_1))$. Thus Alg. 3 runs in $O((||\mathcal{P}_{his}||_1 \cup ||E^+||_1) log(||\mathcal{P}_{his}||_1 \cup ||E^+||_1) + ||\mathcal{P}_{his}||_2)$ time.

BatchRemoval Algorithm. Firstly, Lines 2 - 8 in Alg. 4 runs in $O(||E^-||_1)$ since the dominating cost is enumerasing triangles for edges in E^-. And $||E^-||_1 \subseteq ||CHANGE||_1 \subseteq ||CHANGE||_2$. Secondly, each loop iteration in Lines 10 - 19 costs only $O(||v||_2)$ where v is the vertex at the top of Q now. Thirdly, Only vertices in $CHANGE$ can be pushed into Q and each one can be pushed into Q at most $O(t(v)) \leq O(||v||_2)$ times. By combining the above, we have Alg. 4 runs in $O(||CHANGE||_2 \cdot \max_{v \in CHANGE} ||v||_2)$.

B Algorithm Examples and Pesudo-code

B.1 An Example of BatchInsertion Algorithm

Example 1. *We illustrate the process of our maintenance algorithm by the running example in Fig. 7. Let G be the graph in Fig. 7 excluding edge (v_3, v_4) and (v_4, v_5), i.e. edges indicated by the dotted line. The initial $\preceq$ is $\{v_4, v_5, v_6, v_7, v_0, v_1, v_2, v_3\}$, while $\Phi_0 = \{v_4\}$, $\Phi_2 = \{v_5, v_6, v_7\}$ and $\Phi_6 = \{v_0, v_1, v_2, v_3\}$. In the initialization phase (Line 1*

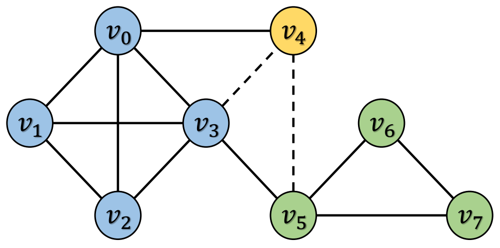


Figure 7: An Example for Trine Maintenance

- 5), Algorithm 3 enumerates all edges in $E^+ = \{(v_3, v_4), (v_4, v_5)\}$ and calculates $ext(\cdot)$ and only $ext(v_4) = 4 \neq 0$. Then, the algorithm enumerates each k and v^* (Line 7).

First, $k = 0$. When $v^* = v_4$, the algorithm moves v_4 from $\Phi_0(G)$ to $\mathcal{P}$ and calculate $ubr(v_4) = 4$ (Lines 18 - 23) since $rem(v_4) + ext(v_4) = 4 > t(v_4) = 0$. Meanwhile, by enumerating $\triangle(v_0, v_3, v_4)$ and $\triangle(v_3, v_4, v_5)$ the algorithm modifies $ext(v_0)$ to 2, $ext(v_3)$ to 4 and $ext(v_5)$ to 2 (Line 19 - 20).

Next, $k = 2$. When $v^* = v_5$, the algorithm moves v_5 from $\Phi_2(G)$ to $\mathcal{P}$ and calculate $ubr(v_5) = 4$ since $rem(v_5) + ext(v_5) = 4 > t(v_5) = 2$. Meanwhile, by enumerating $\triangle(v_5, v_6, v_7)$ the algorithm modifies $ext(v_6)$ and $ext(v_7)$ to 2. When $v^* = v_6$, the algorithm keeps v_6 in $\Phi_2(G)$ since $ext(v_6) = 2 \leq k = 2$ (Line 24 - 27). Meanwhile, by enumerating $\triangle(v_5, v_6, v_7)$ the algorithm decreases $ubr(v_5)$ to 2 and $ext(v_7)$ to 0 (Line 26 - 27).

Then, the algorithm executes Line 29. Since $ubr(v_5) = 2 \leq k = 2$, the algorithm repositions v_5 into the tail of $\Phi_2(G)$, and the algorithm enumerates $\triangle(v_3, v_4, v_5)$ thus modifying $ext(v_3)$ to 2 and $ubr(v_4)$ to 2. Since the above process makes $ubr(v_4) = 2 \leq k = 2$, the algorithm repositions v_4 into the tail of $\Phi_2(G)$, and the algorithm enumerates $\triangle(v_4, v_0, v_3)$ thus modifying $ext(v_0)$ and $ext(v_3)$ to 0.

Throughout the execution of the algorithm, only $t(v_4)$ is modified to 2 while v_7, v_0, v_1, v_2 and v_3 are skipped by the v^* since their $ext(\cdot)$ become zero before they are pointed to by v^* (Line 29).

B.2 An Example of BatchInsertion+ Algorithm

Example 2. *We also use Fig. 7 as the running example. The algorithm first enumerates $E^+ = \{(v_3, v_4), (v_4, v_5)\}$ and calculates $ext(v_4) = 4 \neq 0$ (Lines 1 - 5). And then it enumerates each k (Line 6).*

First, $k = 0$. When $v^* = v_4$, Alg. 5 moves v_4 from Φ_0 to $\mathcal{P}$ and calculate $ubr(v_4) = 4$ (Lines 23, 27 - 28). But unlike Alg. 3, Alg. 5 does not enumerate any $\triangle$, since $\forall u \in N(v_4), t(u) > k$.

Next, $k = 2$. The first step of Alg. 5 is to raise the lower bound of $t(v_4)$ to 2 and compute the impact of v_4 on Φ_2 (Line 7 - 9). At this time, by enumerating $\triangle(v_3, v_4, v_5)$ the algorithm modifies $ext(v_5)$ to 2. The algorithm ignores v_3 since $t(v_3) = 6 > k$. And then $v^* = v_5$, and the algorithm moves v_5 from $\Phi_2(G)$ to $\mathcal{P}$ and calculate $ubr(v_5) = 4$. Meanwhile, by enumerating $\triangle(v_5, v_6, v_7)$ the algorithm modifies $ext(v_6)$ $ext(v_7)$ to 2 (Line 24 - 26). When $v^* = v_6$, the algorithm keeps v_6 in $\Phi_2(G)$ and calculate $rem(v_6) = 0 + 2$ since $rem(v_6) + ext(v_6) = 2 \leq k$ (Line 29 - 30). Meanwhile, by enumerating $\triangle(v_6, v_7, v_5)$ the algorithm modifies $ubr(v_5)$ to 2 and $ext(v_7)$ to 0 (Line 31 - 32).

Then, the algorithm executes Line 33. Since $ubr(v_5) = 2 \leq k$, the algorithm repositions v_5 into $\Phi_2(G)$ and before v_7, and by enumerating $\triangle(v_3, v_4, v_5)$ the algorithm modifies $ubr(v_4)$ to 2. Since the above process makes $ubr(v_4) = 2 \leq k$, the algorithm repositions v_4 into $\Phi_2(G)$ and before v_7, and the algorithm enumerates none of $\triangle$ since all the neighbors are either not in Φ_2 (v_0 and v_3) or before v_4 in (v_5).

Throughout the execution of the Alg. 5, v^* enumerates the same vertices as Alg. 3, i.e., v_4,v_5 and v_6. However, in traversing these vertices, Alg. 5 enumerates only $\triangle(v_3, v_4, v_5)$ and $\triangle(v_5, v_6, v_7)$ (twice each), while Alg. 3 additionally enumerates $\triangle(v_0, v_3, v_4)$ (also twice). It is evident that Delaying Impact reduces the number of triangles enumerated by the algorithm.

Algorithm 5: BatchInsertion+

```
Input: $G$, $E^+$, T-order $\preceq$, $rem(\cdot) : \forall v \in \preceq, rem(v) = sup_{G[\preceq_v]}(v)$
Output: the triness of each vertex after inserting all new edges
1  $\mathcal{P} \leftarrow \emptyset, ubr(\cdot) = \{0\}, ext(\cdot) = \{0\}$;
2  for each $e = (u, v) \in E^+$ do
3      for each $\triangle(u, v, w) \in G \cup E^+$ do
4          $x \leftarrow$ the vertex in $\{u, v, w\}$ with $x \preceq u \wedge x \preceq v \wedge x \preceq w$;
5          $ext(x) \leftarrow ext(x) + 2$;
6  for each $k \in \mathbb{N}$ from small to large, s.t. $\mathcal{P} \neq \emptyset \vee \Phi_k(G) \neq \emptyset$ do
       /* Delay Impact */
7      for $v \in \Phi_k$ shares triangles with vertices in $\mathcal{P}$ do
8          for $\triangle(u, v, w) \in \mathcal{P} \cup \bigcup_{i \geq k} \Phi_i$ with $u \in \mathcal{P} \vee w \in \mathcal{P}$ do
9              $ext(v) \leftarrow ext(v) + 2$;
10     $v^* \leftarrow$ the first vertex in $\Phi_k(G)$;
       /* Reposition Some Vertices in $\mathcal{P}$ to $\preceq^*$ */
11     for $v \in \mathcal{P} \wedge ubr(v) \leq k$ do
12         move $v$ from $\mathcal{P}$ to $\Phi_k(G)$ and before $v^*$;
13         $t(v) \leftarrow k$; $rem(v) \leftarrow ubr(v)$; $ubr(v) \leftarrow 0$;
14         for $\triangle(u, v, w) \in \preceq_v \cup \mathcal{P}$ with
             $u \in (\preceq_v \cap \Phi_k) \cup \mathcal{P} \vee w \in (\preceq_v \cap \Phi_k) \cup \mathcal{P}$ do
15             $ubr(u) \leftarrow ubr(u) - 2$ if $u \in \mathcal{P}$;
16             $ubr(w) \leftarrow ubr(w) - 2$ if $w \in \mathcal{P}$;
17             $ext(u) \leftarrow ext(u) - 2$ if $u \in \Phi_k$;
18             $ext(w) \leftarrow ext(w) - 2$ if $w \in \Phi_k$;
19     while $v^* \neq nil$ do
20         $v^*_{next} \leftarrow$ the vertex next to $v^*$ in $\Phi_k(G)$;
           /* Check $v^*$ for the Pending Set */
21         if $ext(v^*) = 0$ then
22             no process;
23         else if $ext(v^*) + rem(v^*) > k$ then
               /* case-1: move $v^*$ from $\preceq^*$ to $\mathcal{P}$ */
24             for $\triangle(u, v^*, w) \in \preceq_{v^*}$ with $u \in \Phi_k \vee w \in \Phi_k$ do
25                 $ext(u) \leftarrow ext(u) - 2$ if $u \in \Phi_k$;
26                 $ext(w) \leftarrow ext(w) - 2$ if $w \in \Phi_k$;
27             $ubr(v^*) \leftarrow ext(v^*) + rem(v^*)$; $ext(v^*) \leftarrow 0$;
28             move $v^*$ from $\Phi_k(G)$ to $\mathcal{P}$;
29         else if $ext(v^*) + rem(v^*) \leq k$ then
               /* case-2: $v^*$ stays its position in $\preceq^*$ */
30             $rem(v^*) \leftarrow ext(v^*) + rem(v^*)$; $ext(v^*) \leftarrow 0$;
31             for $\triangle(u, v^*, w) \in \preceq^*_v \cup \mathcal{P}$ with $u \in \mathcal{P} \vee w \in \mathcal{P}$ do
32                 Execute Lins 15 - 18 to update $ubr(\cdot)$ and $ext(\cdot)$;
33             Replace $v^*$ with $v^*_{next}$ and do Lines 20-25;
34         $v^* \leftarrow v^*_{next}$;
Return: $t(\cdot)$
```