Efficient Triangle-Connected Truss Community Search In Dynamic Graphs

Tianyang Xu
School of Computer Science
Wuhan University, China
tianyangxu@whu.edu.cn

Zhao Lu
School of Computer Science
Wuhan University, China
luzhao@whu.edu.cn

Yuanyuan Zhu
School of Computer Science
Wuhan University, China
yyzhu@whu.edu.cn

ABSTRACT

Community search studies the retrieval of certain community structures containing query vertices, which has received lots of attention recently. k-truss is a fundamental community structure where each edge is contained in at least $k-2$ triangles. Triangle-connected k-truss community (k-TTC) is a widely-used variant of k-truss, which is a maximal k-truss where edges can reach each other via a series of edge-adjacent triangles. Although existing works have provided indexes and query algorithms for k-TTC search, the cohesiveness of a k-TTC (diameter upper bound) has not been theoretically analyzed and the triangle connectivity has not been efficiently captured. Thus, we revisit the k-TTC search problem in dynamic graphs, aiming to achieve a deeper understanding of k-TTC. First, we prove that the diameter of a k-TTC with n vertices is bounded by $\lfloor \frac{2n}{k+1} \rfloor$. Then, we encapsulate triangle connectivity with two novel concepts, partial class and truss-precedence, based on which we build our compact index, EquiTree, to support the efficient k-TTC search. We also provide efficient index construction and maintenance algorithms for the dynamic change of graphs. Compared with the state-of-the-art methods, our extensive experiments show that EquiTree can boost search efficiency up to two orders of magnitude at a small cost of index construction and maintenance.

PVLDB Reference Format:
Tianyang Xu, Zhao Lu, and Yuanyuan Zhu. Efficient Triangle-Connected Truss Community Search In Dynamic Graphs. PVLDB, 16(3): 519 - 531, 2022.
doi:10.14778/3570690.3570701

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ttcs-vldb/TTCS-VLDB/.

1 INTRODUCTION

Graphs model relationships among entities in many real-world applications where communities naturally exist [15, 35]. Existing studies on communities mainly fall into two categories: *community detection* to find all the communities in the graph, which has been studied for decades [14, 30, 41]; *community search* to retrieve the communities containing query vertices, which has attracted increasing attention recently [13, 19, 39].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 3 ISSN 2150-8097.
doi:10.14778/3570690.3570701

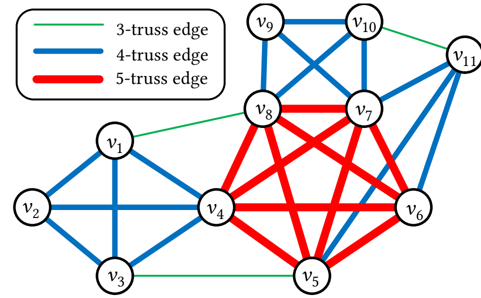


Figure 1: The example graph.

Many community models have been proposed to meet diverse search requirements, among which k-core [27] and k-truss [40] are two of the most widely used models. *k-core* is a subgraph where every vertex has at least k neighbors inside [37], while *k-truss* is a subgraph where every edge is contained in at least $k-2$ triangles inside [8]. Many researches adopted k-truss model since triangles indicate strong relationships [17] and are basic building blocks of complex networks [2], such as geo-social group discovery [5, 6, 29], network reinforcement [4, 38, 46], and other tasks [45, 51].

However, not every pair of edges in a k-truss is strongly connected [19]. Cut-vertex may still exist in a k-truss. For example, v_4 is a cut-vertex in the 4-truss (formed by bold blue edges and bolder red edges) in Fig. 1. Then, *triangle connectivity* is used to strengthen k-truss [19], which defines the *triangle-connected k-truss* model [13]. A k-truss is triangle-connected if its edges can reach each other via a chain of edge-adjacent triangles (i.e., two consecutive triangles share a common edge). For example, let H denote the subgraph consisting of the bold blue edges and bolder red edges in Fig. 1, which is a 4-truss as every edge is contained in at least 2 triangles. But H is not a triangle-connected 4-truss as (v_2, v_3) cannot reach (v_4, v_5) via a chain of edge-adjacent triangles in H. Triangle-connected k-truss can model overlapped communities [9, 10], explore finer granularity [19], contains fewer free-riders, and has no cut-vertex [50], making it a community model preferred by [1, 32, 43, 49, 50]. A *Triangle-connected k-Truss Community (k-TTC)* is a maximal triangle-connected k-truss [1, 13, 19], and the k-TTC search problem is: given a query vertex v_q and a trussness k, retrieve all k-TTCs containing v_q.

TCP-Index [19] and EquiTruss [1] are the state-of-the-art for searching k -TTC, and both take $O(m)$ space to support the online search in dynamic graphs, where m is the number of edges. TCP-Index [19] builds a series of maximal spanning trees whose edge weights represent the pre-computed trussness. However, graph edges may repeatedly appear in TCP-Index, and finding k-TTCs needs to access both TCP-Index and the original graph, which

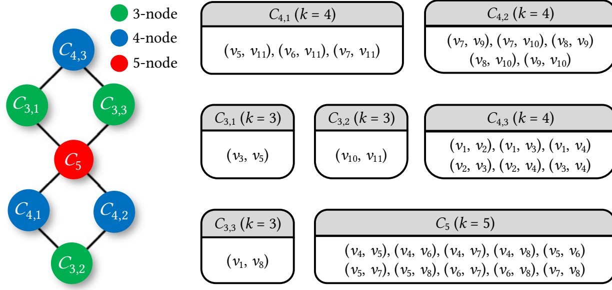


Figure 2: The example equivalence classes and EquiTruss.

makes TCP-Index large and inefficient. EquiTruss [1] constructs a summary graph based on k-truss equivalence classes to keep trussness and triangle connectivity. Thus, it can conduct k-TTC search without accessing the original graph. Unfortunately, the summary graph may be even much larger than the original one due to the small granularity of k-truss equivalence classes (details are reported in Section 6), incurring expensive computational costs.

Thus, in this paper, we revisit the k-TTC Search problem in dynamic graphs [1, 13, 19], and make the following contributions. First, we prove the diameter upper bound of a k-TTC, $\lfloor \frac{2n}{k+1} \rfloor$ (n is the number of vertices in k-TTC), which is no larger than the diameter upper bound of k-truss and theoretically confirms the intuition that triangle connectivity can strengthen the cohesiveness of communities. Next, we propose a novel concept *k-partial class* $\mathcal{P}$ and discover the *truss-precedence* relation $\prec$ on $\mathcal{P}$, which can well describe the triangle connectivity of k-TTC. Then, we derive an efficient index EquiTree inspired by the Hasse diagram of $(\mathcal{P}, \prec)$. Compared with the state-of-the-art indexes, our EquiTree index needs less space and can support the k-TTC query more efficiently. Finally, we propose an efficient index construction algorithm and the maintenance algorithms for the dynamic change of graphs based on the truss-precedence properties of *k-partial classes*.

We organize the rest of the paper as follows. Section 2 gives the preliminaries. Section 3 proves the diameter upper bound of the triangle-connected k-truss. Section 4 presents the truss-precedence and the EquiTree index. Section 5 describes the maintenance of EquiTree. Experimental studies are reported in Section 6. Sections 7 and 8 review the related work and conclude the paper.

2 PRELIMINARY

We denote a simple undirected graph by $G = (V, E)$, where V and E are the vertex set and edge set, respectively. Given a graph G, we use $V(G)$ and $E(G)$ to denote its vertex set and edge set, and use $n = |V(G)|$ and $m = |E(G)|$ to denote its vertex number and edge number. The neighbors of vertex v in G are defined as $N(v, G) = \{u \in V(G) | (u, v) \in E(G)\}$ and their degrees are denoted as $deg(v, G) = |N(v, G)|$. The distance (length of the shortest path) between nodes u and v in G is denoted as $dist_G(u, v)$. A *triangle* $\triangle^G_{uvw}$ is a 3-length cycle defined as the edge set $\{(u, v), (v, w), (w, u)\}$. The support of an edge $e_{uv} = (u, v)$ in G is the number of triangles containing e_{uv}, defined as $sup(e_{uv}, G) = |\{\triangle^G_{uvw} | w \in V(G)\}|$. When the context is clear, we simplify $N(v, G)$, $deg(v, G)$, $dist_G(u, v)$, $\triangle^G_{uvw}$, $sup(e, G)$ as $N(v)$, $deg(v)$, $dist(u, v)$, $\triangle_{uvw}$, $sup(e)$, respectively.

Definition 1. (**k-Truss** [13]) *A k-truss in G is a subgraph H, such that $\forall e \in E(H)$, $sup(e, H) \geq k - 2$.*

The above definition indicates that k-truss is a subgraph H edge-induced by all the edges with support at least $k - 2$ in H. The trussness of a subgraph $H \subseteq G$ is the minimum support of all the edges in H plus 2, defined as $\tau(H) = \min_{e \in E(H)}(sup(e, H) + 2)$. The trussness of edge $e \in E(G)$ is the maximum trussness of subgraphs containing e, i.e., $\tau(e) = \max_{e \in E(H) \wedge H \subseteq G} \tau(H)$. A k-truss H is maximal if there is no H' s.t. $\tau(H') \geq k$ and $H \subset H'$ [9].

A triangle $\triangle$ is a *k-triangle* if $\min_{e \in \triangle} \tau(e) \geq k$. Two triangles $\triangle_s$ and $\triangle_t$ are *edge-adjacent* if they share a common edge, i.e., $|\triangle_s \cap \triangle_t| = 1$. $\triangle_s$ and $\triangle_t$ are k-triangle-connected, denoted as $\triangle_s \overset{k}{\longleftrightarrow} \triangle_t$, if there exists a sequence of k-triangles $\triangle_1 = \triangle_s, \ldots, \triangle_n = \triangle_t$ ($n \geq 2$) such that for $1 \leq i < n$, (1) $|\triangle_i \cap \triangle_{i+1}| = 1$ and (2) $\tau(\triangle_i \cap \triangle_{i+1}) = k$. Two edge e_1 and e_2 are *k-triangle connected*, denoted as $e_1 \overset{k}{\leftrightarrow} e_2$, iff (1) e_1 and e_2 belong to the same k-triangle, or (2) $e_1 \in \triangle_s, e_2 \in \triangle_t$ s.t. $\triangle_s \overset{k}{\longleftrightarrow} \triangle_t$. We can relax $\triangle_s \overset{k}{\longleftrightarrow} \triangle_t$ and $e_1 \overset{k}{\leftrightarrow} e_2$ to $\triangle_s \leftrightarrow \triangle_t$ and $e_1 \leftrightarrow e_2$ if we remove the constraint of k trussness on the triangles and their adjacent edges.

Definition 2. (**Triangle-connected k-Truss Community (k-TTC)** [13]) *A subgraph H is a triangle-connected k-truss community if it satisfies (1) $\tau(H) \geq k$; (2) $\forall e, e' \in E(H), e \leftrightarrow e'$; (3) no other H' exists s.t. $H \subset H'$ and H' satisfies (1) and (2).*

Example 1. *As shown in Fig. 1, the support of (v_2, v_3) is 2 as it is only contained in $\triangle_{v_1 v_2 v_3}$ and $\triangle_{v_2 v_3 v_4}$. The subgraph in bold blue and bolder red edges is a 4-truss but not 4-TTC as (v_2, v_3) cannot reach (v_4, v_5) via a series of edge-adjacent 4-triangles. The subgraph edge-induced by $\{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4), (v_3, v_4)\}$ is a 4-TTC. Similarly, the subgraph in bolder red edges is a 5-truss and also a 5-TTC.*

A k-class Φ_k in graph G [40] is the set of edges with trussness k s.t. $\Phi_k = \{e \mid e \in E(G) \wedge \tau(e) = k\}$, and it can be further divided into k-truss equivalence classes [1]. A *k-truss equivalence class* consists of all edges that are k-triangle connected and have the same trussness k. The set of all equivalence classes forms a mutually exclusive and collectively exhaustive partition of $E(G)$. EquiTruss [1] uses these equivalence classes as super-nodes and links any two super-nodes that are k-triangle connected (k is the minimum trussness of these two super-nodes) to construct a summary graph. Thus, a maximal connected component consisting of super-nodes with trussness at least k in the summary graph represents a k-TTC in the original graph. Then, for a query vertex v_q and an integer k, EquiTruss can retrieve k-TTCs containing v_q by first finding super-nodes containing v_q and then returning the maximal connected components containing these super-nodes.

Example 2. *Fig. 2 shows the EquiTruss for Fig. 1 where each super-node represents an equivalence class as shown in the boxes on the right. Edges in $C_{3,1}$ are 3-triangle connected, edges in $C_{4,1}$ are 4-triangle connected, and $C_{4,3}$ links to $C_{3,3}$ since (v_1, v_8) is 3-triangle connected with edges in $C_{4,3}$. To retrieve the 4-TTCs containing v_9, we start from $C_{4,2}$ to get the maximal connected component induced by super-nodes $\{C_{4,2}, C_5, C_{4,1}\}$ with trussness at least 4, and then return all the edges contained in these super-nodes as the query result.*

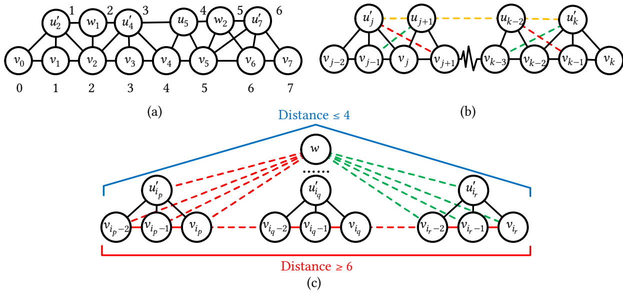


Figure 3: Example for the proof of diameter upper bound.

3 THE DIAMETER UPPER BOUND

For a k-truss H with n vertices, its diameter d is bounded by $\lfloor \frac{2n-2}{k} \rfloor$ [19], which can be easily derived from the following Lemma in [8].

LEMMA 1. *If G_d is a k-truss with diameter d, then $|V(G_d)| \geq \frac{d+1}{2}k$ if d is odd; otherwise $|V(G_d)| \geq \frac{d}{2}k + 1$ [8].*

To study the diameter upper bound of a triangle-connected k-truss, the additional condition, triangle-connectivity, needs to be exploited. We first introduce some notations. Let r be the longest shortest path in a triangle-connected graph with diameter d, G_d, and $V_r = \{v_0, v_1, \ldots, v_d\}$ be the vertices in r. Then we can divide $V(G_d) \setminus V_r$ into two parts: U, the *unique triangle-makers* of r, i.e., $U = \{u \mid \exists \triangle_{uv_iv_{i+1}} \wedge (\nexists \triangle_{u'v_iv_{i+1}} \wedge u \neq u'), v_i \in V_r\}$; W, other vertices, which is $V(G_d) \setminus V_r \setminus U$. Specifically, we denote u as u_i if $\triangle_{uv_{i-1}v_i}$ exists and $\triangle_{uv_iv_{i+1}}$ does not exist. If both $\triangle_{uv_{i-2}v_{i-1}}$ and $\triangle_{uv_{i-1}v_i}$ exist, we denote u as u'_i, indicating that u'_i *covers* (i.e. forms triangle with) two consecutive edges in r. Note that a vertex u cannot cover three edges in r as this will contradict the diameter d. We denote the set of u' as U'. Then, $\forall u'_i \in U'$, $d_{u'_i} = i - 1$ since $d_{u'_i} \leq d_{v_{i-2}} + 1 = i - 1$ and $d_{v_i} \leq d_{u'_i} + 1$ (d_x is a shorthand for $dist(v_0, x)$).

EXAMPLE 3. *In Fig. 3(a), $V_r = \{v_0, \ldots, v_7\}$ and $U = \{u'_2, u'_4, u_5, u'_7\}$. Then $W = \{w_1, w_2\}$. Note that w_2 does not belong to U because $\triangle_{u'_7v_5v_6}$ and $\triangle_{w_2v_5v_6}$ both exist, violating the definition of U. $u'_2 \in U'$ because both $\triangle_{u'_2v_1v_2}$ and $\triangle_{u'_2v_0v_1}$ exist. The same holds for u'_4 and u'_7. The d_x value is labelled beside each x, and $d_{u'_2} = 1$, $d_{u'_4} = 3$, $d_{u'_7} = 6$.*

Next, we discuss the relation between $|W|$ and $|U'|$, which paves the way for Lemma 3.

LEMMA 2. *For a triangle-connected graph G_d with diameter d, if $|U'| \geq 2$, then $|W| \geq 1$.*

PROOF. Assume that $|W| = 0$. We denote any two consecutive vertices in U' as u'_j, u'_k ($j < k$). When $k - j = 1$, both $\triangle_{u'_jv_{j-1}v_j}$ and $\triangle_{u'_kv_{j-1}v_j}$ exist, and u'_j or u'_k must belong to W, which is contradictory. When $k - j > 1$, there must be $u_{j+1}, \ldots, u_{k-2}$ that each of them only covers one edge in r, as shown in Fig. 3(b). For each $j + 1 \leq l \leq k - 2$, since (v_{l-1}, u_l) (green dashed line)/(u_l, v_{l+1}) (red dashed line) does not exist according to the definitions of U/U', (u_l, u_{l+1}) (yellow dashed line) must exist to keep triangle connectivity. As we have derived that $d_{u'_j} = j - 1, d_{u_{j+1}} = j, \ldots, d_{u_{k-2}} = k - 3$, we have $d_{u'_k} \leq d_{u_{k-2}} + 1 = k - 2$, which contradicts $d_{u'_k} = k - 1$. Thus, $|W| \geq 1$. ◻

LEMMA 3. *If G_d is a triangle-connected graph with diameter d ($d > 1$), then $|V(G_d)| \geq 2d$.*

PROOF. We prove this by contradiction. Assume that $|V(G_d)| < 2d$. Then from the definitions, we have $|U| = |V(G_d)| - |W| - |V_r|$, $|U'| = d - |U|$, which implies $|U'| > |W| + 1$. First, according to Lemma 2, there exists at least one w for each consecutive pair u'_{i_j} and $u'_{i_{j+1}}$ so that edges covered by u'_{i_j} and $u'_{i_{j+1}}$ are triangle connected. Next we prove that there is no w alone that can make the edges covered by any three u' (i.e., $u'_{i_p}, u'_{i_q}, u'_{i_r}$ where $p < q < r$) triangle connected. Assume that such a w exists, as shown in Fig. 3(c). Then any subset D' ($|D'| \geq 2$) of the red dashed lines connected to $\{u'_{i_p}, v_{i_p}, v_{i_p-1}, v_{i_p-2}\}$ except$\{(w, v_{i_p-2}), (w, v_{i_p})\}$ can be possible connections, and we always have $dist(w, v_{i_p-2}) \leq 2$. Similarly, we have $dist(w, v_{i_r}) \leq 2$, and thus $dist(v_{i_p-2}, v_{i_r}) \leq 4$. However, from the proof of lemma 2, we have $dist(v_{i_p-2}, v_{i_r}) = i_r - (i_p - 2) = i_r - i_q + i_q - i_p + 2 \geq 2 + 2 + 2 = 6$, which leads to a contradiction. Thus, each consecutive pair u'_{i_j} and $u'_{i_{j+1}}$ needs at least a unique w, and we have $|W| \geq |U'| - 1$ which contradicts $|U'| > |W| + 1$. Thus, $V(G_d) \geq 2d$. ◻

THEOREM 1. *If d is the diameter of a triangle-connected k-truss T_k with n vertices, then $d \leq \lfloor \frac{2n}{k+1} \rfloor$.*

PROOF. We construct a minimum triangle-connected k-truss T_k in two steps. First, we build a triangle-connected graph G_d with diameter path r of length d. Second, we add vertices and edges to G_d to increase the supports of the edges in r by $k - 3$ since they already have at least one support in Step 1. Let m denote the number of vertices of a $(k-1)$-truss with diameter d. Besides $d + 1$ vertices in the diameter path r, we can add $m - (d+1)$ vertices in Step 2 so that each edge in r has support at least $k - 2$. According to Lemma 1, we have:

$$m \geq \begin{cases} (d+1)(k-1)/2, & d \text{ is odd} \\ d(k-1)/2 + 1, & d \text{ is even} \end{cases}. \quad (1)$$

When $d > 1$, we sum up the minimum vertices added in Step 1 ($2d$ according to Lemma 3) and Step 2. Then we get $n \geq |V(G_d)| \geq 2d + m - (d+1)$. When d is odd, $n \geq 2d + (d+1)(k-1)/2 - (d+1) \geq \frac{(d+1)(k+1)}{2} - 2$, which implies $d \leq \frac{2n}{k+1} - \frac{k-3}{k+1}$, and since $k \geq 3$, we have $d \leq \lfloor \frac{2n}{k+1} \rfloor$; when d is even, $n \geq \frac{d(k+1)}{2}$, which implies $d \leq \frac{2n}{k+1}$, and thus $d \leq \lfloor \frac{2n}{k+1} \rfloor$. When $d = 1$, Theorem 1 also holds as T_k is a clique. ◻

Theorem 1 shows that a triangle connected k-truss has a tighter (or at least equal) diameter upper bound than k-truss when $k = 3$ with $n \geq 4$ and $k \geq 4$, which confirms the stronger cohesiveness of k-TTC benefited from the triangle connectivity.

4 THE EQUITREE INDEX

In this section, we first introduce the truss-precedence property of k-TTC, then describe the structure of EquiTree built upon it, and finally give the index construction and query algorithms.

4.1 Truss-Precedence

We build our index based on a novel concept, k-partial class, which is a partition of a k-class at a higher level than k-truss equivalence classes. Recall that a k-truss equivalence class C consists of all the edges with trussness k s.t. $\forall e, e' \in C$, $e \overset{k}{\leftrightarrow} e'$. A k-partial class P will be defined based on a relaxed condition, i.e., $\forall e, e' \in P$, $e \overset{\geq k}{\leftrightarrow} e'$. We say $e \overset{\geq k}{\leftrightarrow} e'$ iff there exists a sequence of k-triangles $\triangle_1, \ldots, \triangle_n$ such that (1) $|\triangle_i \cap \triangle_{i+1}| = 1$ for $1 \leq i < n$ and (2) $\tau(\triangle_i \cap \triangle_{i+1}) \geq k$. Note that different from k-truss equivalence class, condition (2) is relaxed to $\geq k$ instead of k. Thus k-partial class is a coarser-grained partition of k-class that can help build a more compact index to support efficient queries.

Definition 3. **(k-Partial Class)** *A k-partial class P of a k-TTC H_k is a subset of $E(H_k)$ s.t. $\forall e \in P, \tau(e) = k$ and $\forall e, e' \in P$, $e \overset{\geq k}{\leftrightarrow} e'$.*

Definition 4. **(Truss-Precedence $\prec$)** *Given two partial classes P and P', we say P truss-precedes P', denoted as $P \prec P'$, iff $\forall e \in P$, $e' \in P'$, $\tau(e) < \tau(e')$ and $e \overset{\geq \tau(e)}{\longleftrightarrow} e'$.*

Each edge $e \in E(G)$ with $\tau(e) \geq 3$ is in a unique $\tau(e)$-partial class P since e is contained in only one $\tau(e)$-TTC. A k-partial class may contain edges from multiple k-truss equivalence classes, showing a higher level of abstraction.

Example 4. *In Fig. 2, there are one 3-partial class $P_3 = C_{3,1} \cup C_{3,2} \cup C_{3,3}$, two 4-partial classes $P_{4,1} = C_{4,1} \cup C_{4,2}$, $P_{4,2} = C_{4,3}$, and one 5-partial class $P_5 = C_5$. $P_3 \prec P_{4,1}$ since $(v_7, v_{11}) \leftrightarrow (v_7, v_{10})$.*

Theorem 2. *Let $\mathcal{P}$ denote the set of nonempty k-partial classes. Then truss-precedence is a strict partial order relation upon $\mathcal{P}$.*

Proof. Given a k-partial class P, based on Definitions 3 and 4, since there is no $e_1, e_2 \in P$ s.t. $\tau(e_1) < \tau(e_2)$, $P \nprec P$ (irreflexivity). Given two partial classes P_1, P_2 s.t. $P_1 \prec P_2$, based on Definition 4, there is no $e_1 \in P_1, e_2 \in P_2$ s.t. $\tau(e_1) > \tau(e_2)$. Thus, $P_2 \nprec P_1$ (antisymmetry). Given partial classes P_1, P_2, P_3, s.t. $P_1 \prec P_2, P_2 \prec P_3$, $\forall e_1 \in P_1, e_2 \in P_2, e_3 \in P_3$, we have $\tau(e_1) < \tau(e_2), \tau(e_2) < \tau(e_3)$, $e_1 \overset{\geq \tau(e_1)}{\longleftrightarrow} e_2$, and $e_2 \overset{\geq \tau(e_2)}{\longleftrightarrow} e_3$, implying $\tau(e_1) < \tau(e_3)$ and $e_1 \overset{\geq \tau(e_1)}{\longleftrightarrow} e_3$. Thus, $P_1 \prec P_3$ (transitivity). □

Lemma 4. *The Hasse diagram of poset $(\mathcal{P}, \prec)$ is a forest.*

Proof. Suppose that there exists a node representing a partial class P whose indegree is more than 2. Then, there exist $P_1 \prec P$ and $P_2 \prec P$ with $\tau(P_1) = \tau(P_2)$ and $\nexists P' \neq P_1$ s.t. $P_1 \prec P' \prec P_3$ or $P_2 \prec P' \prec P_3$. Thus we have $P_1 \cup P_2 \cup P_3 \subseteq E(H)$ (H is a $\tau(P_1)$-TTC), and based on Definition 3, we have $P_1 = P_2$, which leads to a contradiction. □

The above definitions of k-partial class and the truss-precedence relation give a novel formal hierarchy abstraction of k-TTC, which can well capture the triangle connectivity and nesting property of k-TTC. Such a high-level concept would help us design a more compact index to support efficient query/maintenance.

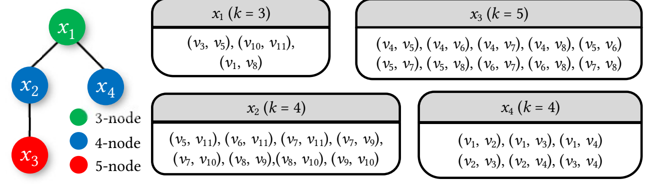


Figure 4: The example EquiTree.

4.2 The Structure of EquiTree

We now extend the Hasse diagram of $(\mathcal{P}, \prec)$ to a compact index, *EquiTree*. According to Lemma 4, we define our index as a tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the tree node set and $\mathcal{E}$ is the tree edge set. Each tree node $x \in \mathcal{V}$ has two attributes, $x.k$ and $x.E$, where $x.E$ is a $x.k$-partial class. We add edge (x_1, x_2) to $\mathcal{E}$ if $x_1.E \prec x_2.E$ and there exists no partial class P s.t. $x_1.E \prec P \prec x_2.E$. In this way, we construct the EquiTree index that captures the nesting property of k-partial classes. Let $\mathcal{T}_x$ denote the subtree rooted at x. It can be proved that each $\mathcal{T}_x$ represents a k-TTC, and vice versa. We use a map from graph edges to tree nodes to enable the index for the search task.

Example 5. *Fig. 4 shows the EquiTree constructed for the graph in Fig. 1. $x_1.E = P_3 = C_{3,1} \cup C_{3,2} \cup C_{3,3} = \{(v_1, v_8), (v_3, v_5), (v_{10}, v_{11})\}$ is a 3-partial class with trussness 3. Meanwhile, $\mathcal{T}_{x_1}$ consists of all the edges in the 3-TTC, and the two 4-TTCs nested in it are represented by the two subtrees $\mathcal{T}_{x_2}$ and $\mathcal{T}_{x_4}$. The 5-TTC nested in $\mathcal{T}_{x_2}$ is $\mathcal{T}_{x_3}$.*

Lemma 5. *If graph G comprises l k-cliques where any two k-cliques are not triangle-connected, then $|V(G)| \geq \frac{kl}{2}$.*

Proof. In graph G, every k-clique shares at most 1 vertex with another k-clique to avoid triangle connectivity. When $k \leq l$, every k-clique can share at most k vertices with other k k-cliques. Thus we can construct a graph G' where each vertex represents a k-clique and each edge denotes a shared vertex between two k-cliques. According to Handshaking Lemma, we have $|E(G')| \leq \frac{kl}{2}$, indicating that the maximum number of unique shared vertices is $\frac{kl}{2}$. Then we have $|V(G)| \geq kl - \frac{kl}{2} = \frac{kl}{2}$. When $k > l$, similarly we have $|V(G)| \geq kl - \frac{l(l-1)}{2} > \frac{kl}{2}$. □

Theorem 3. *Given graph G and its EquiTree $\mathcal{T}$, we have $N < 2n(\ln k_{\max} - \frac{3}{2} + \gamma)$, where $N = |\mathcal{V}|$, $n = |V(G)|$, $k_{\max}$ is the maximum trussness in G, and γ is the Euler–Mascheroni constant.*

Proof. Let $\mathcal{L}_k$ denote the set of tree nodes with trussness k. Then $|\mathcal{V}| = \sum_{3 \leq k \leq k_{\max}} |\mathcal{L}_k|$. Next, we get the upper bound of $|\mathcal{L}_k|$ by forcing the k-TTC represented by subtree $\mathcal{T}_x$ (where $x \in \mathcal{L}_k$) to be as small as possible. When every $x \in \mathcal{L}_k$ represents a k-clique that is not triangle-connected to another k-clique $y \in \mathcal{L}_k$, then $|\mathcal{L}_k|$ reaches it maximum value. Let $V_k = \{v | v \in V(H_k) \wedge \tau(H_k) = k \wedge H_k \subseteq G\}$ denote vertices in k-TTCs with trussness k. According to Lemma 5, $|V_k| \geq \frac{k|\mathcal{L}_k|}{2}$. Then, $\sum_{3 \leq k \leq k_{\max}} |\mathcal{L}_k| \leq \sum_{3 \leq k \leq k_{\max}} \frac{2|V_k|}{k} < 2|V| \sum_{3 \leq k \leq k_{\max}} \frac{1}{k} \approx 2|V|(\ln k_{\max} - \frac{3}{2} + \gamma)$, where γ is the Euler–Mascheroni constant according to the partial sums of the harmonic series [3]. □

Besides, EquiTree has the following favorable properties: (1) every $e \in E$ with $\tau(e) \geq 3$ must be contained in a tree node $x \in \mathcal{V}$; (2) no edge $e \in E$ can be contained in two tree nodes; (3) every $x.E$ is nonempty. Thus, EquiTree needs $O(m)$ space to store all the graph edges, which is the same as EquiTruss and TCP-Index. However, EquiTree has much fewer nodes ($O(n \ln k_{\max})$) than EquiTruss and TCP-Index, and thus can significantly boost the query efficiency.

4.3 Index Construction Algorithm

A k-partial class may consist of multiple k-truss equivalence classes, and we can compute the k-partial class with the following lemma.

Lemma 6. *Two k-truss equivalence classes C and C' ($C' \neq C$) belong to the same partial class P iff there exists another k'-truss equivalence class C'' ($k' > k$) s.t. $C'' \overset{k}{\leftrightarrow} C$, $C'' \overset{k}{\leftrightarrow} C'$.*

Proof. "$\Rightarrow$". We prove this by contradiction. If C and C' belong to the same partial class P, they will occur in the same k-TTC H_k. Assume that C'' does not exist, then $\forall e \in E(H), \tau(e) = k$. According to Definition 3, $P = E(H_k)$. Since all the edges in P are triangle-connected, P is also a k-truss equivalence class, leading to $P = C = C'$, which contradicts $C \neq C'$. Thus, C'' exists. "$\Leftarrow$". C, C', and C'' must occur in the same k-TTC since all the edges have trussness at least k and are k-triangle-connected. According to Definition 3, C and C' belong to the same k-partial class P. □

EquiTree construction is equivalent to the Hasse diagram construction, which involves (1) finding nonempty partial classes, and (2) detecting truss-precedence relations. We give an efficient method to perform (1) and (2) simultaneously in Algorithm 1, which builds EquiTree from leaves to the root. We use AUF [12], a variant of the union-find forest, to enable the incremental detection of triangle connectivity. We keep the triangle connectivity in the union-find forests and record the subtree root x' as the anchor of x if there exists no node y s.t. $y.E \prec x'.E \prec x.E$. To prevent AUF from returning premature anchors, we put new connections in a buffer B (line 12) and update AUF after x is created (line 20).

Algorithm 1 first computes the edge trussness and k-classes Φ_k by truss decomposition [40], and then initializes a list $e.list$ for each edge e to record the triangle-adjacent tree nodes (line 1). Then, we enumerate k-truss equivalence classes from $k_{\max}$ (the largest $\tau(e)$) to 3 by BFS (where Q is the queue) and create a temporary tree node x accordingly (lines 2-20). In the beginning, B and Q are initialized to be empty. Then for each edge $e \in \Phi_k$, we create a temporary node x (line 6), connect x to its children and do merging (lines 10–13), and then process the edges triangle-adjacent to e (lines 14–17).

The child x' of x is detected by checking the tree nodes in $e.list$ ($e \in x'.E$) since ProcessEdge (lines 22-27) has recorded the triangle-adjacent tree nodes y into $e.list$. Thus, we have $x.E \prec y.E$ (line 10). For each such x and y, we find the root of the subtree containing y, x' by AUF and connect x and x' (lines 11-12). In addition, if two nodes x_1 and x_2 have the same child x', indicating that they belong to the same k-partial class, we say a conflict happens and merge them into one single node by MergeNodes (line 13), details of which can be found in Algorithm 8 in the Appendix.

Example 6. *Fig. 5 illustrates the construction process for the graph in Fig. 1. After truss decomposition, we examine k from 5 to 3. When $k = 5$, there is one truss equivalence class labeled C_5, and we create a node for it (Fig. 5(a)). When $k = 4$, there are three equivalence classes $C_{4,1}$, $C_{4,2}$, and $C_{4,3}$. Since $C_{4,1}$ and $C_{4,2}$ are triangle-connected with C_5, we connect them to C_5 (Fig. 5(b)), which causes a conflict and invokes the MergeNodes procedure (Fig. 5(c)). When $k = 3$, there are three equivalence classes $C_{3,1}$, $C_{3,2}$, and $C_{3,3}$. Clearly, $C_{3,2}$ is connected with $C_{4,2}$. Since both $C_{3,1}$ and $C_{3,3}$ are connected to C_5 and $C_{4,3}$, we connect them to $C_{4,3}$ and the parent of C_5 based on AUF (Fig. 5(d)). Then we handle the conflicts by MergeNodes (Fig. 5(e)).*

Algorithm 1: Leaf-to-Root EquiTree Construction

Input : A graph $G = (V, E)$
Output: EquiTree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$
1 compute $\tau(e)$ and $\Phi_k = \{e | \tau(e) = k\}$; $e.list \leftarrow \emptyset$;
2 **for** $k \leftarrow k_{\max}$ **to** 3 **do**
3 $\quad B \leftarrow \varnothing$; $Q \leftarrow \varnothing$;
4 $\quad$**while** $\exists e \in \Phi_k$ **do**
5 $\quad\quad$mark e as processed and push e to Q;
6 $\quad\quad$add a new tree node x to $\mathcal{V}$;
7 $\quad\quad$**while** $Q \neq \varnothing$ **do**
8 $\quad\quad\quad e(u, v) \leftarrow Q$.dequeue();
9 $\quad\quad\quad x.E \leftarrow x.E \cup \{e\}$;
10 $\quad\quad\quad$**foreach** $y \in e.list$ **do**
11 $\quad\quad\quad\quad x' \leftarrow$ AUF.Find(y);
12 $\quad\quad\quad\quad$add (x, x') to $\mathcal{E}$ and B;
13 $\quad\quad\quad\quad P \leftarrow$ parents of x'; MergeNodes(P);
14 $\quad\quad\quad$**foreach** $w \in N(u) \cup N(v)$ **do**
15 $\quad\quad\quad\quad \tau_{min} \leftarrow \min\{\tau((u, w)), \tau((v, w)), \tau((u, v))\}$;
16 $\quad\quad\quad\quad$ProcessEdge($(u, w), x$) if $\tau(u, w) = \tau_{min}$;
17 $\quad\quad\quad\quad$ProcessEdge($(v, w), x$) if $\tau(v, w) = \tau_{min}$;
18 $\quad\quad$remove e from Φ_k and E;
19 $\quad$**foreach** $(x, y) \in B$ **do**
20 $\quad\quad$AUF.Union(x, y);
21 **return** $\mathcal{T}(\mathcal{V}, \mathcal{E})$;
22 **Procedure** ProcessEdge (e, y)
23 $\quad$**if** *e is not processed* **then**
24 $\quad\quad$**if** $\tau(e) = k$ **then**
25 $\quad\quad\quad$mark e as processed and push e to Q;
26 $\quad\quad$**else**
27 $\quad\quad\quad e.list \leftarrow e.list \cup y$;

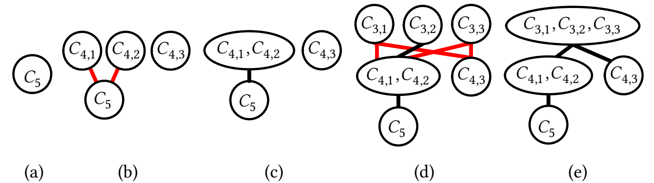


Figure 5: The example for construction algorithm.

The time complexity of Equitree construction is $O(m^{1.5})$. The initialization by truss decomposition takes $O(m^{1.5})$ time [40]. The AUF operations need $O(m \cdot \alpha(m))$ time since there are $O(m)$ nodes and each AUF operation needs $O(\alpha(m))$ time ($\alpha(m)$ is the inverse Ackermann function). For each edge $e \in E$, we consider each triangle

Algorithm 2: EquiTree Query

Input : EquiTree $\mathcal{T}$, query vertex v_q, trussness k
Output : k-truss communities $\mathcal{A}$

1 $X \leftarrow \{x \mid x \in \mathcal{V} \wedge x.E$ contains $v_q\}$;
2 **foreach** $x \in X$ **do**
3 | **while** *the parent of x exists with trussness* $\geq k$ **do**
4 | | $x \leftarrow$ the parent of x;
5 | $\mathcal{A} \leftarrow \mathcal{A} \cup \bigcup_{y \in \mathcal{T}_x} y.E$;
6 **return** $\mathcal{A}$;

Algorithm 3: The Maintenance Framework

Input : EquiTree $\mathcal{T}$, the modified edge $e^* = (u, v)$

1 $G' \leftarrow$ new graph after modifying G;
2 $\Phi'_k \leftarrow \{e \mid \tau(e, G') = k \wedge \tau(e, G) \neq \tau(e, G')\}$;
3 $\Psi_k \leftarrow \{x \mid x.E \cap \Phi'_k \neq \emptyset\}$;
4 $\Phi \leftarrow \{\Phi'_k | \Phi'_k \neq \emptyset\}$; $\Psi \leftarrow \{\Psi_k | \Psi_k \neq \emptyset\}$;
5 $Y \leftarrow$ NewNodes$(\mathcal{T}, \Phi, \Psi)$;
6 Restructure $(\mathcal{T}, e^*, Y, \Psi)$;

containing e, which is examined only once since e will be removed from the graph. Therefore, finding truss equivalence classes and connecting tree nodes is equivalent to counting all the triangles in $O(m^{1.5})$ time. Thus, the overall complexity is $O(m^{1.5})$.

4.4 Query Algorithm

Algorithm 2 conducts community search on EquiTree. First, we find all the tree nodes X containing v_q (line 1). Then for each $x \in X$, we trace up to find the the last ancestor of x, x' with $\tau(x') \geq k$ (lines 3–4). Finally, we add the edges in $\mathcal{T}'_x$ to the final results $\mathcal{A}$ (line 5).

Example 7. *Consider the example graph in Figure (1). For query vertex v_4 and $k = 4$, we first find x_3 and x_4 containing v_4. As $x_3.k = 5$, we trace up to x_2 with trussness 4. Then, we obtain two 4-TTCs represented by $\mathcal{T}_{x_2}$ and $\mathcal{T}_{x_4}$. For query vertex v_9 and $k = 4$, we first find x_2 with trussness 4 and return the 4-TTC represented by $\mathcal{T}_{x_2}$ directly, which is more efficient than EquiTruss as shown in Example 2.*

The query algorithm is time-optimal with complexity $O(|E(\mathcal{A})|)$ ($\mathcal{A}$ is k-truss communities containing v_q), since finding roots of subtrees containing v_q is dominated by returning edges in $\mathcal{A}$.

5 INDEX MAINTENANCE

Real-world graphs are constantly changing, and updating the index accordingly is critical for online search [13]. As vertex insertion/deletion can be reduced to edge insertion/deletion [1, 19], we focus on the latter. Maintaining EquiTree is quite challenging as it requires a series of triangle-connectivity checking on the area affected by edge insertion/deletion. We first give the general maintenance framework for both insertion and deletion, then dive into the details for each operation.

Algorithm 3 gives the maintenance framework. First, we identify the *k-affected edges* $\Phi'_k = \{e | \tau(e, G') = k \wedge \tau(e, G) \neq \tau(e, G')\}$ in a way similar to [19, 47] (line 2)[1], find the *k-affected tree nodes* Ψ_k containing edges in Φ'_k (line 3), and compute the union of nonempty Φ'_k and Ψ_k, respectively (line 4). Next, we generate tree nodes Y to hold the affected edges by NewNode (Algorithm 9 in Appendix) (line 5), and restructure EquiTree to absorb Y by two different procedures tailored for insertion and deletion (line 6).

We first analyze the time complexity of lines 1–5 in Algorithm 3. Lines 1 and 2 need $O(||E_{\Phi'}||_1 |E_{\Phi'}|)$ time for edge insertion and $O(R \log R)$ time ($R = O(||\mathsf{AFF}||_1^2 |\mathsf{AFF}|)$) for edge deletion [47], where $E_{\Phi'} = \bigcup_{2 \leq k \leq k_{\max}} \Phi'_k$ is the set of edges with changed trussness, AFF is the minimum difference between the truss decomposition orders of the original and updated graphs (AFF is a superset of $E_{\Phi'}$, but its practical size is very close to $E_{\Phi'}$), $||E_{\Phi'}||_1$ and $||\mathsf{AFF}||_1$ are sizes of the 1-hop neighbors of $E_{\Phi'}$ and AFF respectively. For convenience, we denote the above complexity for insertion and deletion as T_+ and T_-, respectively. Since lines 3-5 can be done in $O(|E_{\Phi'}|)$ time which is dominated by T_+ and T_-, the overall time complexity of Algorithm 3 except line 6 is T_+ for insertion and T_- for deletion.

5.1 Edge Insertion

We restructure EquiTree for edge insertion (line 6 in Algorithm 3) in two steps, as shown in Algorithm 4. First, we create new tree nodes Y for edges with increased trussness and examine the tree nodes triangle-connected to any $y \in Y$ (lines 1-13). Second, we handle the triangle connections between nodes not in Y (lines 14-20).

In both steps, SerialMerge (lines 21-29) deals with tree node merging triggered by triangle connections. For two partial classes $P, P' \in E(H_{\tau(e)})$, if $\tau(e) \leq \tau(e')$, we denote it as $P \leq P'$. If a new node x' is triangle-connected to tree nodes x_1 and x_2 with $\tau(x') \geq \tau(x_1) = \tau(x_2)$, we need to merge x_1 and x_2 since $x_1 \leq x'$ and $x_2 \leq x'$. Besides, we also need to merge the two paths from x_1 and x_2 to their latest common ancestor x'', i.e., merge any two nodes on paths $x'' \rightsquigarrow x_1$ and $x'' \rightsquigarrow x_2$ if they have the same trussness. We call the set of nodes that might trigger a series of merging as a seed node set, denoted by S. SerialMerge then collects the tree nodes on the path from nodes in S to their latest common ancestor and merges any of these nodes with the same k value.

In step 1, we first remove the empty nodes in Ψ_k (line 3-5). Next, for each $y \in Y$, we find the nodes triangle-connected to y, X' (line 7), and collect seed nodes S that may trigger merging. For each $x' \in X'$, if $\tau(x') \leq \tau(y)$, then x' and its ancestors all precede y. Thus, we add (x', y) to $\mathcal{E}$ and add y to S (line 9–10). Likewise, if $\tau(x') > \tau(y)$, we add (y, x') to $\mathcal{E}$ and add x' to S (line 11-12). Finally, we perform SerialMerge on S according to Lemma 6 (line 13).

In step 2, we denote the triangle containing e^* as $\triangle'$. Then for the other two edges $e_1, e_2 \in \triangle'$, we find their corresponding tree nodes x_1 and x_2 (line 18). If $x_1, x_2 \notin Y$, we add (x_1, x_2) to $\mathcal{E}$ since they are triangle-connected with $\triangle'$, and then add x_2 to S' (line 19). Finally, we perform SerialMerge on S' (line 20).

Example 8. *After inserting (v_8, v_{11}) into Fig. 1, the affected edges $\Phi'_5 = \{(v_5, v_{11}), (v_6, v_{11}), (v_7, v_{11}), (v_8, v_{11})\}$ change trussness to 5, and $\Phi'_4 = \{(v_{10}, v_{11})\}$ change trussness to 4. The affected nodes are $\Psi_5 = \{x_2\}$ and $\Psi_4 = \{x_1\}$ (Fig. 6(a)). We create new nodes y_1 with $y_1.E = \Phi'_4$ and y_2 with $y_2.E = \Phi'_5$ (Fig. 6(b)). In step 1, we start from y_2 and connect y_2 to x_2, x_3, and y_1 since they are triangle-connected (Fig. 6(c)). Next, we perform SerialMerge on $S = \{x_2, x_3, y_1, y_2\}$ to merge y_2, x_3 into x'_3, and x_2, y_1 into x'_2 (Fig. 6(d)). Then, we check the*

[1] For a graph G, we set $\tau(e, G) = 0$ if $e \notin E(G)$.

Algorithm 4: Restructure-Insertion

Input : EquiTree $\mathcal{T}$, inserted edge e^*, new nodes Y

1 **foreach** $k \leftarrow k_{\max}$ *to* 2 **do**
2   skip if $\Phi_k = \varnothing$;
3   **foreach** $x \in \Psi_k$ **do**
4   **if** $x.E = \varnothing$ **then**
5   remove tree node x;
6    $y \leftarrow$ the tree node in Y with trussness k;
7    $X' \leftarrow$ tree nodes triangle-connected with y;
8   **foreach** $x' \in X'$ **do**
9   **if** $\tau(x') \leq \tau(y)$ **then**
10        $\mathcal{E} \leftarrow \mathcal{E} \cup \{(x', y)\}$; $S \leftarrow S \cup \{y\}$;
11   **if** $\tau(x') > \tau(y)$ **then**
12        $\mathcal{E} \leftarrow \mathcal{E} \cup \{(y, x')\}$; $S \leftarrow S \cup \{x'\}$;
13   SerialMerge(S);
14 $T' \leftarrow$ all triangles with e^* as an edge;
15 **foreach** $\triangle' \in T'$ **do**
16    $e_1, e_2 \leftarrow$ edges of $\triangle'$ except e^* (w.l.o.g. $\tau(e_1) \leq \tau(e_2)$);
17   **if** $e_1, e_2 \notin Y$ **then**
18      $x_1, x_2 \leftarrow$ tree nodes containing e_1 and e_2 respectively;
19      $\mathcal{E} \leftarrow \mathcal{E} \cup \{(x_1, x_2)\}$; $S' \leftarrow S' \cup \{x_2\}$;
20 SerialMerge(S');
21 **Procedure** *SerialMerge*(S)
22   **while** $S \neq \varnothing$ **do**
23      $S' \leftarrow$ nodes in S with the largest trussness;
24   S.pop(S');
25   add parents of S' to S if they are not added before;
26      $x \leftarrow$ MergeNodes(S');
27   add (x, x') and remove other children of x if x' exists;
28      $x' \leftarrow x$;

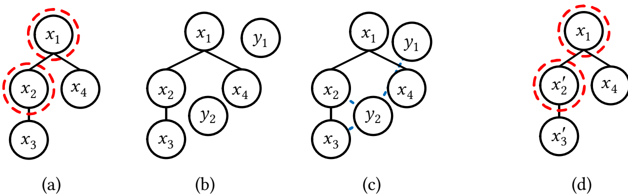


Figure 6: Example for edge insertion.

triangle connectivity of y_1 (now merged into x_2'), which is triangle-connected to x_3', and find that (x_2', x_3') is already connected. Next, we find nodes containing the other two edges in the triangle newly formed by e^*, which are x_2' and x_3'. Since they are handled in step 1, we do not need to perform step 2.

The overall time complexity of index maintenance for edge insertion is $O(|E_{\Phi'}|deg_{\max} + k_{\max}N \log N + T_+)$, where T_+ is the time complexity of lines 1-5 in Algorithm 3 as discussed before, and $O(|E_{\Phi'}|deg_{\max}+k_{\max}N \log N)$ ($deg_{\max}$ is the maximum degree in G) is the time complexity of Algorithm 4 derived as follows. First of all, SerialMerge takes $O(N \log N)$ time since lines 23-28 take $O(\log |S|)$ time if we maintain S by a heap and $|S| \leq N$ since a tree node will be added to S at most once. In lines 2-7, checking the affected nodes takes $O(|\Psi|)$ time, and finding nodes triangle-connected to y takes $O(\sum_{(u,v) \in y.E} \min\{deg(u), deg(v)\})$ according to [40]. Since lines 8-12 and lines 14-20 take $O(|X'|)$ and $O(|sup(e^*)|)$ time, respectively dominated by other parts, $|\Psi| \leq |E_{\Phi'}|$, and $\sum_{(u,v) \in E_{\Phi'}} \min\{deg(u), deg(v)\} \leq |E_{\Phi'}| \cdot deg_{\max}$, the overall time complexity of Algorithm 4 is $O(|E_{\Phi'}|deg_{\max} + k_{\max}N \log N)$. The practical index maintenance time is significantly less than construction since both $|E_{\Phi'}|$ and N are much smaller than m.

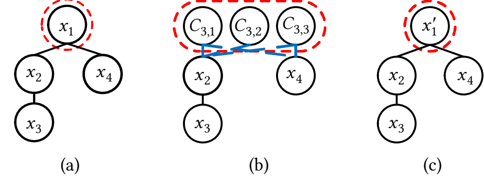


Figure 7: Example for NodeSplit.

5.2 Edge Deletion

We also restructure EquiTree for edge deletion in two steps: first, we handle the k-affected tree nodes for each k, and then we deal with other nodes that may split.

In both steps, SplitNode will split edges in a tree node into k-truss equivalence classes [1] and reorganize them into k-partial classes. Specifically, we first split $x.E$ into k-truss equivalence classes and create a temporary node x' for each class; then we connect x' to each child of x, x'', if $x' \prec x''$. If multiple nodes connect to the same x'', they should be merged according to Lemma 6.

Example 9. *After deleting (v_8, v_{11}) in Example 8, we split x_1 in Fig. 7(a). First, we detect its k-truss equivalence classes and split x_1 into three nodes $C_{3,1}$, $C_{3,2}$, and $C_{3,3}$ (Fig. 7(b)). Since $C_{3,1}$ and $C_{3,3}$ are triangle-connected to x_3 and x_4, we have $C_{3,1}, C_{3,3} \prec x_3$ and $C_{3,1}, C_{3,3} \prec x_4$. Since x_2 is the child of x_1 and the parent of x_3, we have $C_{3,1}, C_{3,3} \prec x_2$, and can connect $C_{3,1}, C_{3,3}$ to x_2 and x_4. Since $C_{3,2} \prec x_2$, we connect $C_{3,2}$ to x_2. As $C_{3,1}, C_{3,2}, C_{3,3}$ all connect to x_2, we merge them into x_1' (Fig. 7(c)).*

Algorithm 5 shows the maintenance of EquiTree for edge deletion. In the first step, we deal with the affected tree nodes. First, we delete any tree nodes with trussness 2 (line 1). Next, for each k, we denote the node in Y with trussness k by y, the node in Ψ_k by x^*, and the parent of x^* by x_p^* (lines 4-6). We then insert y between x^* and x_p^*, and merge two nodes if there is a conflict (lines 7-9). Then, if $x^*.E = \varnothing$, we delete x^* and add its children to x_p^* (line 11). Finally, if possible, we perform SplitNode on x^* (line 12). In the second step, we check other nodes that may split. We use x' to denote the parent of the last split tree node, and perform SplitNode on x' and then on its parents recursively until the node cannot split (lines 13-16).

Example 10. *We delete (v_8, v_{11}) after its insertion (Eg. 8). Likewise, $\Phi_4' = \{(v_5, v_{11}), (v_6, v_{11}), (v_7, v_{11}), (v_8, v_{11})\}$ and $\Phi_3' = \{(v_{10}, v_{11})\}$. The affected nodes are $\Psi_4 = \{x_3\}$ and $\Psi_3 = \{x_2\}$ (Fig. 8(a)). We create y_3 with $y_3.E = \Phi_4'$ and y_2 with $y_2.E = \Phi_3'$ (Fig. 8(b)). During Restructure-Deletion, x_2 and x_3 are not deleted since $x_2.E \neq \varnothing$ and $x_3.E \neq \varnothing$. First, we insert y_3 between x_2 and x_3 (Fig. 8(c)) and merge*

Algorithm 5: Restructure-Deletion

Input : EquiTree $\mathcal{T}$, deleted edge e^*, new nodes Y

1 delete the tree nodes in Y with trussness 2;
2 **foreach** $k \leftarrow k_{\max}$ *to* 3 **do**
3   skip if $\Phi_k = \emptyset$;
4    $y \leftarrow$ the tree node in Y with trussness k;
5    $x^* \leftarrow$ the tree node in Ψ_k;
6    $x_p^* \leftarrow$ the parent of x^*;
7   delete (x_p^*, x^*) if x_p^* exists and add (y, x^*) if y exists;
8   **if** *both* x_p^* *and* y *exist* **then**
9   add edge (x_p^*, y); Merge($\{y, x_p^*\}$) if $\tau(y) = \tau(x_p^*)$;
10   **if** $x^*.E = \emptyset$ **then**
11   delete x^* and add its children to x_p^* if x_p^* exists;
12   SplitNode(x^*) if x^* is not deleted;
13 $x' \leftarrow$ the parent node of the last tree node split;
14 **while** x' *exists and can be split* **do**
15   SplitNode(x');
16    $x' \leftarrow$ the parent of x';
17 **Procedure** *SplitNode(x)*
18    $X \leftarrow$ split x into multiple nodes according to k-truss equivalence classes in $x.E$;
19   **foreach** $x' \in X$ **do**
20   **foreach** x'' *that* $x'' \xleftrightarrow{\tau(x')} x'$ *and* $\tau(x'') > \tau(x')$ **do**
21        $y \leftarrow$ a child of x that $x'' \in \mathcal{T}_y$; add edge (x', x'');
22   if x'' already has a parent, merge x' and x'';

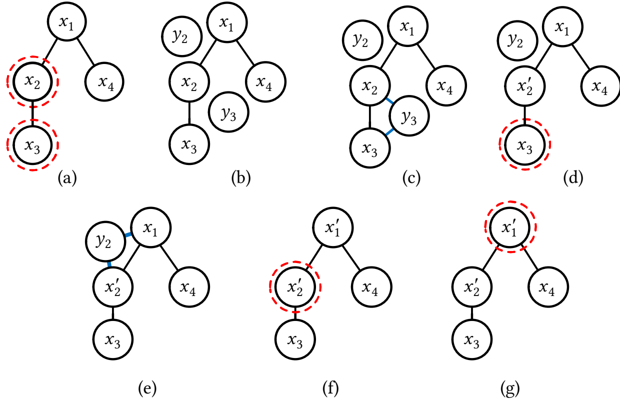


Figure 8: Example of edge deletion.

x_2 and y_3 into x_2' as $\tau(x_2) = \tau(y_3) = 4$ (Fig. 8(d)). We then try SplitNode on x_3, but $x_3.E$ contains only one 5-truss equivalence class, leading to no changes. Next, we insert y_2 between x_1 and x_2' (Fig. 8(e)) and merge y_2 and x_1 into x_1' since $\tau(y_2) = \tau(x_1) = 3$ (Fig. 8(f)). As x_2' only contains one 4-truss equivalence class, no changes occur during SplitNode. Out of the loop, since x_2' is the last node split, we start splitting from x_1'. Although there are three 3-truss equivalence classes, they all precede x_2'. Therefore, they are merged again into x_1', and thus no changes happen (Fig. 8(g)).

The overall time complexity of index maintenance for edge deletion is $O(m^{1.5})$. First, lines 2-11 can be done in $O(k_{\max})$. Next,

Algorithm 6: Restructure-Insertion Batched

Input : EquiTree $\mathcal{T}$, inserted edge e^*, new nodes Y

1 **foreach** $k \leftarrow k_{\max}$ *to* 2 **do**
2   lines 2-5 in Algorithm 4;
3    $Y' \leftarrow$ the nodes in Y with trussness k;
4   **foreach** $y \in Y'$ **do**
5   lines 7-12 in Algorithm 4;
6 lines 14-19 in Algorithm 4; BatchMerge(S);
7 **Procedure** *BatchMerge*(S)
8   partition S into $S_1', \ldots, S_l'$ based on connectivity;
9   **foreach** $S_i' (1 \le i \le l)$ **do**
10   SerialMerge(S_i');

we analyze the cost of splitting nodes. First of all, the time complexity of SplitNode is $O(\sum_{(u,v)\in x.E} \min\{deg(u), deg(v)\} + k_{\max} \cdot |x.E|)$ since checking the triangle connectivity in tree node x takes $O(\sum_{(u,v)\in x.E} \min\{deg(u), \deg(v)\})$ [40] (line 18) and lines 19-22 take $O(|X| \cdot k_{\max}) \le O(|x.E| \cdot k_{\max})$ time. Since each SplitNode deals with edges with specific trussness, each edge will be split at most once. Thus lines 12-16 take $O(\sum_{(u,v)\in E(G)} \min\{deg(u), deg(v)\} + mk_{\max}) = O(\sum_{v\in V(G)} deg(v) \cdot |nb_{\ge}(v)| + mk_{\max})$ to split nodes where $nb_{\ge}(v) = \{v | v \in N(v), deg(v) \ge deg(u)\}$. Since $|nb_{\ge}(v)| \le 2\sqrt{m}$ [40], $O(\sum_{v\in V(G)} deg(v) \cdot |nb_{\ge}(v)|) = O(m^{1.5})$. Since $k_{\max} < \sqrt{m}$ and T_- is dominated by $O(m^{1.5})$, the overall time complexity is $O(m^{1.5})$. In practice, the maintenance time is significantly less as usually only a small portion of tree nodes are split.

5.3 Batch Maintenance

In real-world applications, a series of edge insertions or deletions may emerge in a short time window. Thus, we propose a batch maintenance algorithm, which follows the same initial steps (lines 1-5) in Algorithm 3 but with a different restructure process.

5.3.1 Batch Edge Insertion. Algorithm 6 shows the maintenance for batched edge insertion, which differs from Algorithm 4 in the following two aspects. First, instead of only one new node y created for each k, a set of such new tree nodes Y needs to be processed in batch maintenance (lines 3-5). Second, SerialMerge in Algorithm 4 cannot directly deal with batch insertion because it requires every $e \in S$ to be in the same connected component of G, which is true for single insertion but not necessarily true for batch insertion. Therefore, we propose BatchMerge (lines 7-10) to partition S into subsets $S_1', \ldots, S_k'$ based on their connectivity (line 8) and then apply SerialMerge to each subset (lines 9-10).

Following the analysis of Algorithm 4, the maintenance time complexity for batched edge insertion is $O(|E_{\Phi'}| deg_{\max} + l \cdot N \log N + T_+)$ since sorting $|Y|$ needs $O(|Y| \log |Y|)$ time ($|Y| < N$) and the number of connected components in S is l (line 8).

5.3.2 Batch Edge Deletion. Redundant SplitNode operations will occur if we apply Algorithm 5 to each edge individually because deleting multiple edges may cause the same tree node x to be split repeatedly. If we collect all the nodes that need to be split first and then split them in a batch, we can avoid such redundant splits. Therefore, we propose a new procedure, BatchSplit, to split a node

Algorithm 7: Restructure-Deletion Batched

Input : EquiTree $\mathcal{T}$, deleted edge e^*, new nodes Y

1 **foreach** $x^* \in \Psi$ **do**
2   **if** $x^*.E = \emptyset$ **then**
3     $x_p^* \leftarrow$ the parent of x^*;
4   delete x^* and add its children to x_p^* ;
5     $S' \leftarrow S' \cup \{x_p^*\}$;
6   **else**
7     $S' \leftarrow S' \cup \{x^*\}$;
8 lines 1–5 in Algorithm 6;
9 BatchMerge(S); BatchSplit(S');
10 **Procedure** *BatchSplit(S)*
11   **while** $S \neq \emptyset$ **do**
12     $S' \leftarrow$ nodes in S with the same largest trussness;
13   S.pop(S');
14   **foreach** $x \in S'$ **do**
15   SplitNode(x);
16   add the parent of x to S if x can be split and this parent has not been added before;

set S, as shown in lines 11–16 of Algorithm 7. It begins with nodes in S with the largest k value, denoted as S' (line 3). For each node $x \in S'$, if it can be split, we add its parent to S (line 16). We repeat such processes until S becomes empty.

Equipped with BatchSplit, we developed a new maintenance method for batched edge deletion, as shown in Algorithm 7, whose main difference from Algorithm 6 is that we collect the node to be split first and delete the empty tree nodes before splitting (lines 1-7). Specifically, for each affected tree node x^*, if it is empty, we delete it and add its children to its parent x_p^* (if exists) (lines 2-4). Moreover, we also add x_p^* to S' as the deletion of x^* may break connections in x_p^* (line 5). For other nonempty affected nodes x^*, since their connection may be broken as well, they are also added into S' (line 7). Next, similar to Algorithm 6, for each $y \in Y'$, we find the tree nodes triangle-connected to y, X', connect y to nodes in X', and collect seed nodes S (line 8). Finally, we run BatchMerge on S and run BatchSplit on S' (line 9).

The time complexity for batched edge deletion is $O(m^{1.5})$. First, as SplitNode costs $O(\sum_{(u,v)\in x.E} \min\{deg(u), deg(v)\} + |x.E|k_{\max})$ and no node is added twice (line 16), BatchSplit takes $O(\sum_{(u,v)\in E(G)} \min\{deg(u), deg(v) + mk_{\max}\} = O(m^{1.5})$ as proved in single edge deletion. Next, we examine the complexity of other parts, which are also bounded by $O(m^{1.5})$ as analyzed in the maintenance algorithm for single-edge deletion. Thus the overall complexity is $O(m^{1.5})$.

6 EXPERIMENT

We conducted extensive experimental studies to evaluate the effectiveness and efficiency of our algorithms.

6.1 Setting Up

Datasets. We consider six real-world networks publicly available in SNAP and Network Repository[2]. Table 1 reports their statistics,

[2]https://snap.stanford.edu/data and https://www.networkrepository.com

Table 1: Graph statistics, along with the maximum vertex degree $d_{\max}$ and the maximum edge trussness $k_{\max}$.

Dataset	Vertices	Edges	$d_{\max}$	$k_{\max}$
Facebook (FB) [42]	4,039	88,234	77	97
Catster (CS) [36]	149,700	5,449,275	80,636	207
DBLP (DB) [42]	317,080	1,049,866	342	114
LiveJournal (LJ) [42]	3,997,962	34,681,189	14,815	352
Orkut (OK) [42]	3,072,441	117,185,083	33,313	78
Weibo (WB) [36]	58,655,850	261,321,071	278,491	80

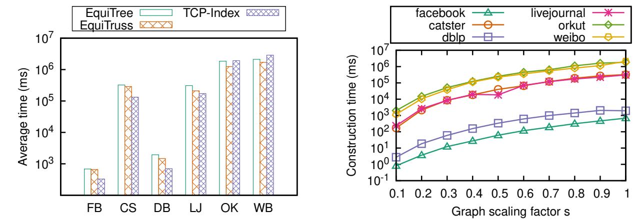


Figure 9: Comparison of construction time.

Figure 10: Construction scalability of EquiTree.

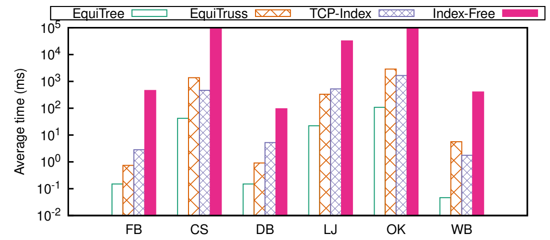


Figure 11: Comparison of average query time.

where $d_{\max}$ denotes the maximum vertex degree and $k_{\max}$ denotes the maximum edge trussness.

Algorithms. We compare *EquiTree* with the state-of-the-art methods, *TCP-Index* [19] and *EquiTruss* [1]. To evaluate the query efficiency, we compare *EquiTree*, *EquiTruss*, *TCP-Index*, and a naive baseline *Index-Free* that starts from the query vertex v_q and traverses the triangle-connected edges with pre-computed trussness no less than k to get the k-TTC containing v_q. To evaluate the maintenance efficiency, we compare *EquiTree* with the maintenance algorithms of *EquiTruss* [1] and a baseline algorithm *EquiTree-Reconstruct* that constructs EquiTree from scratch. We obtained the executable file of TCP-Index from the authors and implemented other algorithms in C++. All the experiments were conducted on a Linux server with two Intel 4.0GHz 8-core CPUs and 96GB memory.

6.2 Index Compactness

First, we evaluate the compactness of *TCP-Index* $\mathcal{C}$, *EquiTruss* $\mathcal{G}$ and *EquiTree* $\mathcal{T}$. Table 2 compares the index size, where $\mathcal{V}/\mathcal{E}$ denotes the nodes/edges number of the index (its ratio to the vertex/edge number of the original graph is given in parenthesis) and S denotes the index size in megabytes. *TCP-Index* and *EquiTruss* have multiple times more vertices than the original graphs while *EquiTree* has

Table 2: Comparison of the sizes of TCP-Index C, EquiTruss $\mathcal{G}$, and EquiTree $\mathcal{T}$ (K = 10^3, M = 10^6).

Dataset	$\lvert\mathcal{V}(\mathcal{T})\rvert$	$\lvert\mathcal{V}(\mathcal{G})\rvert$	$\lvert\mathcal{V}(C)\rvert$	$\lvert\mathcal{E}(\mathcal{T})\rvert$	$\lvert\mathcal{E}(\mathcal{G})\rvert$	$\lvert\mathcal{E}(C)\rvert$	$\lvert S(\mathcal{T})\rvert$	$\lvert S(\mathcal{G})\rvert$	$\lvert S(C)\rvert$
Facebook	393 (9.7%)	5K (147.7%)	176K (4367.2%)	377 (0.4%)	33K (37.8%)	172K (195.4%)	0.82	1.16	8.74
Catster	579 (0.4%)	1M (692.5%)	10M (7174.6%)	221 (0.0%)	8M (164.1%)	10M (194.5%)	39.74	181.2	526
DBLP	72K (22.8%)	126K (40.0%)	869K (274.3%)	21K (2.0%)	105K (10.0%)	549K (52.3%)	13.02	14.42	101
LiveJournal	294K (7.4%)	4M (119.2%)	68M (1715.3%)	141K (0.4%)	13M (38.6%)	65M (188.5%)	401	635	3081
Orkut	287K (9.3%)	17M (560.7%)	231M (7526.7%)	105K (0.1%)	76M (65.4%)	228M (194.8%)	1456	2872	10553
Weibo	96K (0.2%)	23M (40.9%)	449M (767.1%)	3K (0.0%)	66M (25.5%)	389M (148.9%)	1061	2382	15507

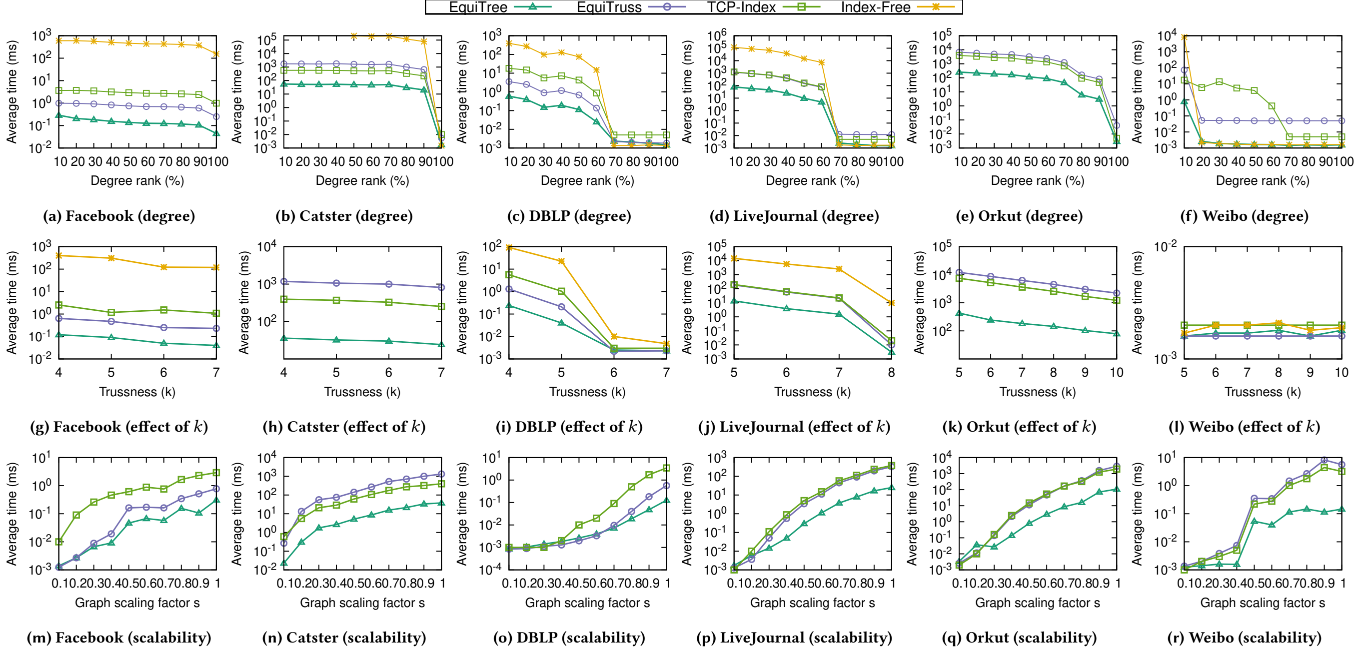


Figure 12: Comparison of query efficiency with varying parameters.

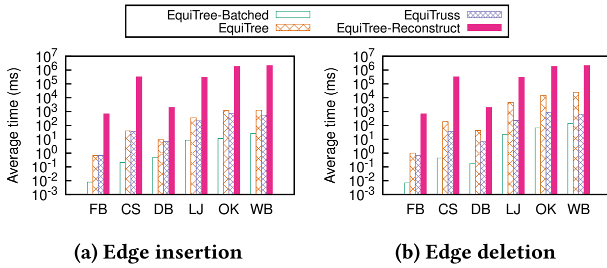


Figure 13: Comparison of average maintenance efficiency.

significantly fewer nodes (mostly less than 10%) and edges (less than 2%). We also report the storage size in megabytes. EquiTree has the smallest size, but the differences between the other two indexes are not as large as those of nodes/edge numbers since all the indexes need to store graph edges with a trussness of at least 3.

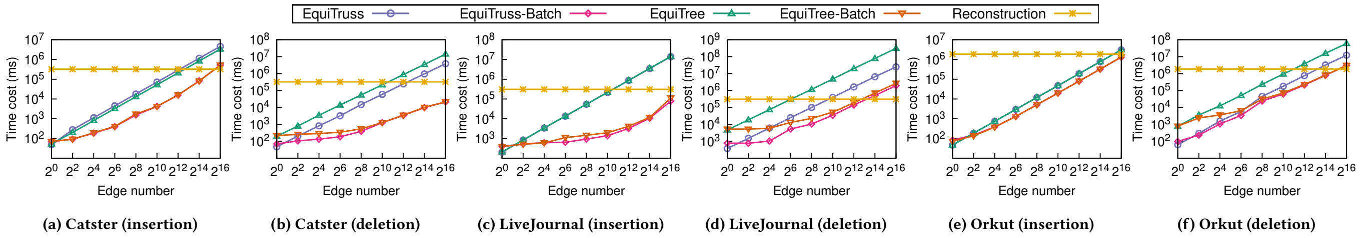
6.3 Index Construction Efficiency

Fig. 9 shows the construction time of *TCP-Index*, *EquiTruss*, and *EquiTree*, which are close. We also test the scalability of *EquiTree*. Let s denote the graph scaling factor. For each s from 0.1 to 1.0, we randomly select $s|V(G)|$ vertices to obtain the induced subgraphs, on which we construct *EquiTree*. The result in Fig. 10 shows that the construction algorithm is scalable.

6.4 Query Efficiency

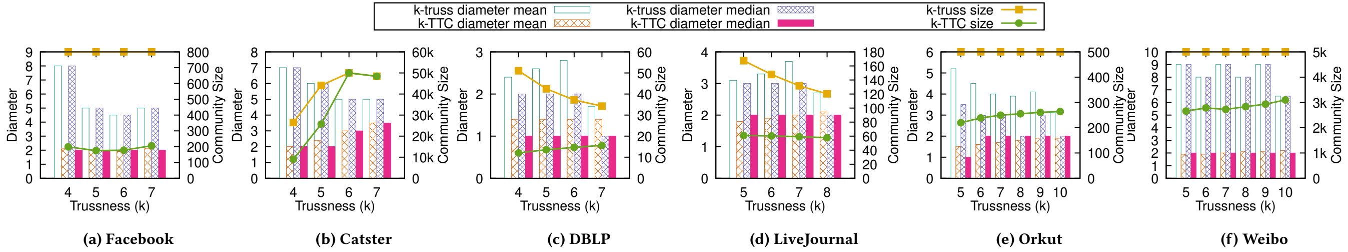
First, evaluate the general query efficiency of the compared algorithms. From each graph G, we randomly select 1000 query vertices, and set the default truss value k as 4 in Facebook and Catster, 5 in DBLP, 6 in LiveJournal, 10 in Orkut and Weibo. The results are reported in Fig. 11. *EquiTree* significantly outperforms *EquiTruss* and *TCP-Index*, where the speedup is up to two orders of magnitude in Weibo. *Index-Free* performs the worst since it incurs exhaustive BFS explorations and costly triangle-connectivity evaluations.

Then, we examine the effect of degrees of query vertices. We denote the degree rank of a vertex as 10% if its degree is in the top 10%, and 20% if its degree falls in the top [10%, 20%], and other degree ranks 30%, . . . , 100% are defined similarly. For each rank, we randomly select 1,000 vertices. The results are reported in Fig. 12 (a)–(f). Searching k-TTCs containing high-degree vertices usually takes



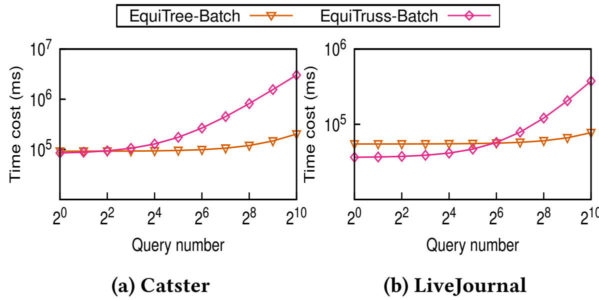
(a) Catster (insertion) (b) Catster (deletion) (c) LiveJournal (insertion) (d) LiveJournal (deletion) (e) Orkut (insertion) (f) Orkut (deletion)

Figure 14: Comparison of maintenance total time costs with different edge numbers.



(a) Facebook (b) Catster (c) DBLP (d) LiveJournal (e) Orkut (f) Weibo

Figure 15: Comparison of median, mean diameters and average sizes of k-TTCs and k-truss communities with different k values.



(a) Catster **(b) LiveJournal**

Figure 16: Total time of querying and batch maintenance.

more time because they tend to participate in more and larger communities. When the degree percentile is above 70%, the query time drops drastically because the number of communities decreases. In Weibo, the query time drops significantly after 20%, showing that only the 10% most active users form tight communities.

Next, we examine the effect of k. We randomly choose 1,000 query vertices and run community searches with varying k. Fig. 12 (g)–(l) show that *EquiTree* performs the best for most k, especially on large graphs. Also, the query time decreases when k increases as the communities become smaller. The average query time for Weibo is quite small, as a random vertex in such a social network may not participate in any k-TTC.

Finally, we test the scalability of the query algorithms. For each scaling factor s from 0.1 to 1.0, we randomly select $s|V(G)|$ vertices from the datasets and obtain the induced subgraphs. Next, we randomly choose 1,000 query vertices[3]. The results in Fig. 12 (m)–(r) show that our query algorithm has the best scalability.

[3]If the subgraph has less than 1,000 vertices for each subgraph, all of them are selected.

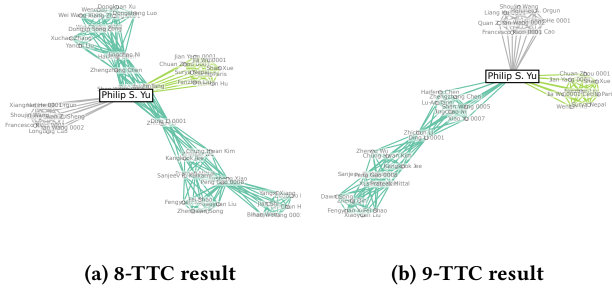
6.5 Maintenance Efficiency

First, we compare the average maintenance time of *EquiTree* and *EquiTruss* in Fig. 13. For each graph, we randomly delete 1,000 edges, and then add them back [4]. For *EquiTruss* and *EquiTree*, we maintain the index at each edge update. For *EquiTree-Batched*, we maintain the index by inserting/deleting *all the edges* in one batch. We also plot *EquiTree-Reconstruct*, the baseline algorithm that constructs *EquiTree* from scratch. *EquiTree* outperforms the baseline *EquiTree-Reconstruct* by several orders of magnitude. Moreover, *EquiTree-Batched* outperforms *EquiTree* by more than one order of magnitude. For edge insertion, which commonly occurs in real-world applications, *EquiTree* and *EquiTruss* have similar performances. For edge deletion, which rarely occurs in real-world applications, *EquiTree* needs a little more time, which is still acceptable considering its performance in online community search.

Then, we evaluate how the maintenance time changes with the number of updated edges on three medium-sized graphs. The number of edges ranges from 2^0 to 2^{16}, and the maintenance is done in one batch for batched algorithms. As shown in Fig. 14, the batched algorithms generally outperform the non-batched ones except for an extremely small number of updated edges (usually less than 4). Moreover, the time costs of non-batched algorithms increase more rapidly when the edge number increases, while the increases of batched algorithms are much slower. Overall, batched algorithms are more efficient under a certain scale. For large-scale updates, reconstruction may be a better option.

Although the maintenance of EquiTree-Batch is slightly slower than EquiTruss-Batch on some datasets in Fig. 14, from the viewpoint of the system, maintenance will be done when there are no or only a few queries in practice. For the rare case that the index is maintained when the queries come, we further evaluate the total

[4]Only edges with trussness larger than 2.



(a) 8-TTC result **(b) 9-TTC result**

Figure 17: Community search result of Philip S. Yu on DBLP

time cost of querying and batch maintenance. Fig.16 shows the total time cost of querying and batch maintenance of 1,000 edge insertions/deletions on Catster and LiveJournal as the maintenance of EquiTree-Batch is slightly slower than that of EquiTruss-Batch on these two datasets in Fig. 14. The results show that on both datasets, EquiTree-Batch surpasses EquiTruss-Batch when the query number is larger than 50. Therefore, we recommend EquiTree-Batch for most cases where queries are more frequent than maintenance.

6.6 Effectiveness Analysis

6.6.1 Statistics on Diameters. Diameter is an important metric to evaluate community quality. For each graph, we find all k-truss and k-TTCs with varying k and then calculate their mean and median diameters. To show the effectiveness of triangle connectivity, if a k-truss is identical to a k-TTC, we remove it from the evaluation. Fig. 15 shows that k-TTCs has smaller mean and median diameters on all the datasets, especially on Facebook and Weibo, which demonstrates that k-TTC generates tighter communities and confirms our analysis on the diameter upper bound of k-TTC (Section 3).

6.6.2 Community size. For each graph, we find all k-truss and k-TTCs with varying k, and then report their vertex number in Fig. 15. The sizes of k-TTCs are significantly smaller in most cases, which is more friendly for users to explore.

6.6.3 A Case Study. Previous studies have statistically shown the effectiveness of k-TTC [1, 19]. For self-completeness, we give a case study that searches k-TTCs for Philip S. Yu in the DBLP graph[5]. We connect an edge when two authors have cooperated at least three times to reduce the free-rider effect, and then perform the search with truss values 8 and 9, respectively. The results are reported in Fig. 17. When $k = 8$, the search result contains three communities: two are from Macquarie University and one is from NEC Laboratories America. When $k = 9$, although Philip S. Yu is still in three communities, the two from Macquarie University are not affected, while the one from NEC Laboratories becomes significantly smaller. Therefore, by tuning k, with the help of EquiTree, we can achieve personalized community search in large-scale graph analysis.

7 RELATED WORKS

Community detection aims to retrieve all communities in the entire network, which is well-studied in the literature [14] [24]. Some decomposition methods were also proposed to detect specified community structures, such as core [7] [25] and truss [8] [40]. *Community search* aims to find communities containing a given set of query vertices, which is attracting increasing interest. [13, 20] comprehensively review recent studies of community search based on models such as quasi-clique [28], k-core [37], k-truss [22] [1], k-ECC [18], and (k, p)-core [33, 44]. Utilizing richer information, attributed community search [12, 21] and spatial community search [6, 26] have also been studied. Recently, machine learning also has been applied to search flexible community structures [16, 23].

Maintenance of community search aims to keep the index updated under graph changing. [47, 48] study the coreness/trussness maintenance in dynamic graphs, and [34] study such maintenance under distributed environment. [31] studies the maintenance of the hierarchy of connected k-cores against edge insertion/deletion. CL-Tree [11] maintains an index to efficiently support attributed community search in dynamic graphs. [1, 19] study the maintenance of indexes that support efficient k-TTC search.

8 CONCLUSION

In this paper, we study k-TTC search in dynamic graphs. We first derive a smaller diameter upper bound for k-TTC and then develop two novel concepts to capture the triangle connectivity among edges at a high level. We design the compact EquiTree index paired with efficient construction and maintenance algorithms. Compared with the state-of-the-art, our EquiTree is more compact and can boost the query performance of k-TTC query up to two orders of magnitude on real-world graphs at a small additional construction and maintenance cost.

9 APPENDIX

Algorithm 8: MergeNodes

Input : Nodes to be merged S and EquiTree $\mathcal{T}(\mathcal{V}, \mathcal{E})$

1. $V_+ \leftarrow \{x \mid (a, x) \in \mathcal{E}, a \in S\}$; $E' \leftarrow \bigcup_{a \in S} a.E$;
2. delete S from tree;
3. create a new tree node c with $c.E \leftarrow E'$ and add c to tree;
4. add edge (c, v) **foreach** $v \in V_+$;

Algorithm 9: NewNode

Input : EquiTree $\mathcal{T}$, affected edges Φ, affected nodes Ψ

Output: New nodes Y

1. **for** $k \leftarrow k_{\max}$ *to* 2 **do**
2. $\quad x.E \leftarrow x.E \setminus \Phi'_k$ **foreach** $x \in \Psi_k$;
3. $\quad$ create new tree node y s.t $y.E = \Phi'_k$; $Y \leftarrow Y \cup \{y\}$;
4. **return** Y;

ACKNOWLEDGMENTS

The work was supported by grants of the National Key Research and Development Project of China (NKRDPC) 2020AAA0108505, Natural Science Foundation of China 61972291 and 62272353, and Natural Science Foundation of Hubei Province 2021CFB327. Yuanyuan Zhu is the corresponding author.

[5]https://dblp.uni-trier.de/xml

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-Based Community Search: A Truss-Equivalence Based Indexing Approach. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1298–1309.

[2] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-Order Organization of Complex Networks. *Science* 353, 6295 (2016), 163–166.

[3] R. P. Boas and J. W. Wrench. 1971. Partial Sums of the Harmonic Series. *The American Mathematical Monthly* 78, 8 (1971), 864.

[4] Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Solon P. Pissis, and Michelle Sweering. 2021. On Breaking Truss-Based Communities. In *The 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. ACM, 117–126.

[5] Lu Chen, Chengfei Liu, Rui Zhou, Jianxin Li, Xiaochun Yang, and Bin Wang. 2018. Maximum Co-Located Community Search in Large Scale Social Networks. *Proceedings of the VLDB Endowment* 11, 10 (2018), 1233–1246.

[6] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu, Jeffrey Xu Yu, and Jianxin Li. 2020. Finding Effective Geo-Social Group for Impromptu Activities with Diverse Demands. In *The 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 698–708.

[7] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Ozsu. 2011. Efficient Core Decomposition in Massive Networks. In *IEEE 27th International Conference on Data Engineering*. IEEE, 51–62.

[8] Jonathan Cohen. 2008. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report* 16 (2008).

[9] Jonathan D. Cohen. 2019. Trusses and Trapezes: Easily-Interpreted Communities in Social Networks.

[10] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online Search of Overlapping Communities. In *The 2013 International Conference on Management of Data*. ACM, 277–288.

[11] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and Efficient Attributed Community Search. *The VLDB Journal* 26, 6 (2017), 803–828.

[12] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1233–1244.

[13] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A Survey of Community Search Over Big Graphs. *The VLDB Journal* 29, 1 (2020), 353–392.

[14] Santo Fortunato. 2010. Community Detection in Graphs. *Physics Reports* 486, 3-5 (2010), 75–174.

[15] Edgardo Galan-Vasquez and Ernesto Perez-Rueda. 2021. A Landscape for Drug-Target Interactions Based on Network Analysis. *PLoS ONE* 16, 3 (2021), 1–21.

[16] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: Lightweight Interactive Community Search Via Graph Neural Network. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1006–1018.

[17] Mark S. Granovetter. 1973. The Strength of Weak Ties. *Amer. J. Sociology* 78, 6 (1973), 1360–1380.

[18] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2016. Querying Minimal Steiner Maximum-Connected Subgraphs in Large Graphs. In *The 25th ACM International on Conference on Information and Knowledge Management*. ACM, 1241–1250.

[19] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-Truss Community in Large and Dynamic Graphs. In *The 2014 ACM SIGMOD International Conference on Management of Data*. ACM, 1311–1322.

[20] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2019. *Community Search Over Big Graphs*. Vol. 14. Morgan & Claypool Publishers.

[21] Xin Huang and Laks V. S. Lakshmanan. 2017. Attribute-Driven Community Search. *Proceedings of the VLDB Endowment* 10, 9 (2017), 949–960.

[22] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *Proceedings of the VLDB Endowment* 9, 4 (2015), 276–287.

[23] Yuli Jiang, Yu Rong, Hong Cheng, Xin Huang, Kangfei Zhao, and Junzhou Huang. 2022. Query Driven-Graph Neural Networks for Community Search: From Non-Attributed, Attributed, to Interactive Attributed. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1243–1255.

[24] D. Jin, Z. Yu, P. Jiao, S. Pan, D. He, J. Wu, P. Yu, and W. Zhang. 2021. A Survey of Community Detection Approaches: From Statistical Modeling to Deep Learning. *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1.

[25] Wissam Khaouid, Marina Barsky, Venkatesh Srinivasan, and Alex Thomo. 2015. K-Core Decomposition of Large Networks on a Single PC. *Proceedings of the VLDB Endowment* 9, 1 (2015), 13–23.

[26] Junghoon Kim, Tao Guo, Kaiyu Feng, Gao Cong, Arijit Khan, and Farhana M. Choudhury. 2020. Densely Connected User Community and Location Cluster Search in Location-Based Social Networks. In *The 2020 ACM SIGMOD International Conference on Management of Data*. ACM, 2199–2209.

[27] Yi-Xiu Kong, Gui-Yuan Shi, Rui-Jie Wu, and Yi-Cheng Zhang. 2019. K-Core: Theories and Applications. *Physics Reports* 832 (2019), 1–32.

[28] Pei Lee and Laks V.S. Lakshmanan. 2016. Query-Driven Maximum Quasi-Clique Search. In *The 2016 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 522–530.

[29] Qiyan Li, Yuanyuan Zhu, Junhao Ye, and Jeffrey Xu Yu. 2021. Skyline Group Queries in Large Road-social Networks Revisited. *IEEE Transactions on Knowledge and Data Engineering* (2021), 1–1.

[30] Zhenjun Li, Yunting Lu, Wei-Peng Zhang, Rong-Hua Li, Jun Guo, Xin Huang, and Rui Mao. 2018. Discovering Hierarchical Subgraphs of K-Core-Truss. *Data Science and Engineering* 3, 2 (2018), 136–149.

[31] Zhe Lin, Fan Zhang, Xuemin Lin, Wenjie Zhang, and Zhihong Tian. 2021. Hierarchical Core Maintenance on Large Dynamic Graphs. *Proceedings of the VLDB Endowment* 14, 5 (2021), 757–770.

[32] Boge Liu, Fan Zhang, Wenjie Zhang, Xuemin Lin, and Ying Zhang. 2021. Efficient Community Search with Size Constraint. In *IEEE 37th International Conference on Data Engineering*. IEEE, 97–108.

[33] Zhao Lu, Yuanyuan Zhu, Ming Zhong, and Jeffrey Xu Yu. 2022. On Time-Optimal (k, p)-Core Community Search in Dynamic Graphs. In *IEEE 38th International Conference on Data Engineering*. IEEE, 1396–1407.

[34] Qi Luo, Dongxiao Yu, Hao Sheng, Jiguo Yu, and Xiuzhen Cheng. 2021. Distributed Algorithm for Truss Maintenance in Dynamic Graphs. In *Parallel and Distributed Computing, Applications and Technologies*. Vol. 12606. Springer International Publishing, 104–115.

[35] Joshua M. Mueller, Laura Pritschet, Tyler Santander, Caitlin M. Taylor, Scott T. Grafton, Emily Goard Jacobs, and Jean M. Carlson. 2021. Dynamic Community Detection Reveals Transient Reorganization of Functional Brain Networks Across a Female Menstrual Cycle. *Network Neuroscience* 5, 1 (2021), 125–144.

[36] Ryan Rossi and Nesreen Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29. AAAI Press, 4292—-4293.

[37] Mauro Sozio and Aristides Gionis. 2010. The Community-Search Problem and How to Plan a Successful Cocktail Party. In *The 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 939.

[38] Xin Sun, Xin Huang, Zitan Sun, and Di Jin. 2021. Budget-Constrained Truss Maximization over Large Graphs: A Component-based Approach. In *The 30th ACM International Conference on Information & Knowledge Management*. ACM, 1754–1763.

[39] Chaokun Wang and Junchao Zhu. 2019. Forbidden Nodes Aware Community Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. AAAI Press, 758–765.

[40] Jia Wang and James Cheng. 2012. Truss defineiecomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5, 9 (2012), 812–823.

[41] Jierui Xie, Stephen Kelley, and Boleslaw K. Szymanski. 2013. Overlapping Community Detection in Networks: The State-of-the-Art and Comparative Study. *Comput. Surveys* 45, 4 (2013), 1–35.

[42] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *The ACM SIGKDD Workshop on Mining Data Semantics*. ACM, 1–8.

[43] Zhibang Yang, Xiaoxue Li, Xu Zhang, Wensheng Luo, and Kenli Li. 2022. K-Truss Community Most Favorites Query Based on Top-t. *World Wide Web* 25, 2 (2022), 949–969.

[44] Chen Zhang, Fan Zhang, Wenjie Zhang, Boge Liu, Ying Zhang, Lu Qin, and Xuemin Lin. 2020. Exploring Finer Granularity within the Cores: Efficient (k,p)-Core Computation. In *IEEE 36th International Conference on Data Engineering*. IEEE, 181–192.

[45] Fan Zhang, Conggai Li, Ying Zhang, Lu Qin, and Wenjie Zhang. 2020. Finding Critical Users in Social Communities: The Collapsed Core and Truss Problems. *IEEE Transactions on Knowledge and Data Engineering* 32, 1 (2020), 78–91.

[46] Fan Zhang, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. 2018. Efficiently Reinforcing Social Networks over User Engagement and Tie Strength. In *IEEE 34th International Conference on Data Engineering*. IEEE, 557–568.

[47] Yikai Zhang and Jeffrey Xu Yu. 2019. Unboundedness and Efficiency of Truss Maintenance in Evolving Graphs. In *The 2019 International Conference on Management of Data*. ACM, 1024–1041.

[48] Yikai Zhang, Jeffrey Xu Yu, Ying Zhang, and Lu Qin. 2017. A Fast Order-Based Approach for Core Maintenance. In *IEEE 33rd International Conference on Data Engineering*. IEEE, 337–348.

[49] Zibin Zheng, Fanghua Ye, Rong-Hua Li, Guohui Ling, and Tan Jin. 2017. Finding Weighted K-Truss Communities in Large Networks. *Information Sciences* 417 (2017), 344–360.

[50] Yuanyuan Zhu, Jian He, Junhao Ye, Lu Qin, Xin Huang, and Jeffrey Xu Yu. 2020. When Structure Meets Keywords: Cohesive Attributed Community Search. In *The 29th ACM International Conference on Information & Knowledge Management*. ACM, 1913–1922.

[51] Yuanyuan Zhu, Qian Zhang, Lu Qin, Lijun Chang, and Jeffrey Xu Yu. 2022. Cohesive Subgraph Search Using Keywords in Large Networks. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 178–191.