Truss-based Community Search: a Truss-equivalence Based Indexing Approach

Esra Akbas Peixiang Zhao
Department of Computer Science
Florida State University
Tallahassee, Florida 32306
akbas@cs.fsu.edu zhao@cs.fsu.edu

ABSTRACT

We consider the community search problem defined upon a large graph G: given a query vertex q in G, to find as output all the densely connected subgraphs of G, each of which contains the query v. As an *online*, *query-dependent* variant of the well-known community detection problem, community search enables *personalized* community discovery that has found widely varying applications in real-world, large-scale graphs. In this paper, we study the community search problem in the *truss-based model* aimed at discovering all dense and cohesive k-truss communities to which the query vertex q belongs. We introduce a novel equivalence relation, *k-truss equivalence*, to model the intrinsic density and cohesiveness of edges in k-truss communities. Consequently, all the edges of G can be partitioned to a series of k-truss equivalence classes that constitute a space-efficient, truss-preserving index structure, EquiTruss. Community search can be henceforth addressed directly upon EquiTruss *without* repeated, time-demanding accesses to the original graph, G, which proves to be theoretically optimal. In addition, EquiTruss can be efficiently updated in a dynamic fashion when G evolves with edge insertion and deletion. Experimental studies in real-world, large-scale graphs validate the efficiency and effectiveness of EquiTruss, which has achieved at least an order of magnitude speedup in community search over the state-of-the-art method, TCP-Index.

1. INTRODUCTION

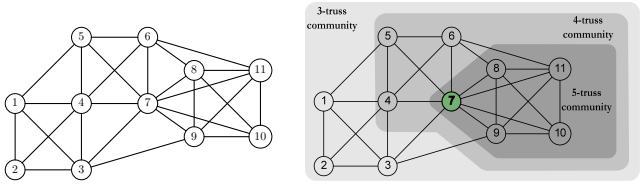
Modern science and technology have witnessed in the past decade a proliferation of complex data that can be naturally modeled and interpreted as *graphs*. In real-world networked applications, the underlying graphs oftentimes exhibit fundamental community structures supporting widely varying interconnected processes. *Community detection* has thus become one of the most well-studied problems in graph management and analytics, the objective of which is to identify densely-knitted subgraphs revealing latent and critical community structures of graphs [35, 20]. Existing community detection methods focus primarily on discovering communities in an a-priori, top-down manner with only reference to the input graph. As a result, *all* communities have to be exhaustively identified, thus incurring expensive time/space cost and a huge amount of fruitless computation, if only a fraction of them are of special interest to end-users.

In many real-world occasions, people are more interested in the communities pertaining to a given vertex. For example, in a social network, a user is typically more curious about the communities she participates in rather than all the communities of the entire graph [29]. This *query-dependent* variant of community detection is usually referred to as the *community search* problem, the objective of which is to identify dense subgraphs containing the query vertex [8, 13, 11, 7, 29]. Community search admits an *online*, *bottom-up* process for community detection desirable especially in real-world, large graphs. Furthermore, it opens the door to *personalized* community discovery, and has thus found a wide range of applications in expert recommendation and team formation [36], personal context discovery [4], social contagion modeling [32], and gene/protein regulation [29].

In this paper, we consider community search based on the *truss* model [5]. Given a graph G, a k-truss ($k \geq 2$) is the largest subgraph of G with each constituent edge contained in at least $(k-2)$ triangles. There have been numerous dense-subgraph notions proposed thus far towards modelling real-world community structures, including clique or quasi-clique [6, 31], k-core [25, 1], k-truss [11, 33], and nucleus [26], to name a few. Our choice of k-truss as the underlying community model is influenced by the following important facts: (1) as opposed to primitive vertices/edges, the higher-order graph motif, *triangle*, is exploited as building blocks to quantify the strong and stable relationships in communities [2]. As a result, the high-density and cohesiveness of real-world communities can be encoded in k-truss with strong theoretical guarantees (more details will be elaborated in Section 2.2); (2) k-truss enables a comprehensive modelling of multiple, overlapping communities in G. By tuning the parameter k, we can derive a collection of k-truss communities that form an inclusive, dense-graph hierarchy representing the *cores* of G at varied levels of granularity [26]; (3) discovering all k-trusses from G is polynomially tractable [33], while most existing dense-graph models render the community search problem NP-hard [13, 6]. As a consequence, k-truss has been extensively employed for community search in real-world networked applications [12, 14, 13, 11, 33].

EXAMPLE 1. *Figure 1(a) presents a toy graph G. Given a query vertex v_7, the k-truss communities ($k = 3, 4, 5$) con-*

Proceedings of the VLDB Endowment, Vol. 10, No. 11



(a) Graph G **(b)** k-truss communities for v_7

Figure 1: Truss-based communities for v_7 in G

taining v_7 are illustrated in Figure 1(b). For instance, when $k = 5$, each edge in the 5-truss community is contained in at least 3 different triangles. By tuning the values of k, we generate a series of dense and cohesive community structures pertaining to v_7. □

Searching k-truss communities based simply on the definition becomes immediately prohibitive as it incurs a lot of random exploration and wasteful edge accesses in the graph. The state-of-the-art solution, TCP-Index [11], indexes the pre-computed k-trusses in a series of maximum spanning trees (MSTs) for community search. Unfortunately, each edge of G might be maintained in multiple MSTs, making TCP-Index redundant and excessively large, typically several times larger than G. On the other hand, k-truss communities have to be reconstructed *online* from MSTs during community search, thus involving costly, repeated accesses to G and making community search extremely inefficient, especially in real-world, massive graphs (The analysis on the limitations of TCP-Index will be detailed in Section 2.2).

In this paper, we propose a novel graph indexing solution to the truss-based community search problem in real-world, large-scale graphs. Our main idea is to introduce a new concept of *k-truss equivalence* among edges of a graph: given two edges e and e' of G, they are k-truss equivalent if and only if they belong to the same k-truss, and are further connected by a series of triangles in a strong sense (modeled by the notion of *k-triangle connectivity* in Definition 9). Intuitively, if e belongs to a k-truss community *w.r.t.* a query vertex v, so does e'. We prove that k-truss equivalence is an *equivalence relation*, such that all the edges of G can be partitioned to at most one *equivalence class* based on k-truss equivalence. We further design a truss-equivalence based index, EquiTruss, which is a summarized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consisting of a *super-node* set, $\mathcal{V}$, and a *super-edge* set, $\mathcal{E}$; each super-node $\nu \in \mathcal{V}$ represents an equivalence class of edges based on k-truss equivalence, and there exists a super-edge $(\nu, \mu) \in \mathcal{E}$ if the edges partitioned to the super-node ν and μ ($\nu, \mu \in \mathcal{V}$), respectively, are connected via triangles in G. We prove that community search can be carried out directly upon EquiTruss *without* repeated accesses to the original graph G, and its time complexity is solely determined by the actual size of output, which is theoretically optimal. In addition, EquiTruss is amenable to efficient, dynamic update when the underlying graph G evolves in terms of edge insertion and deletion. We examine, both theoretically and experimentally, the efficiency and effectiveness of EquiTruss, which has achieved at least an order of magnitude speedup for community search, in comparison to the state-of-the-art method, TCP-Index. Furthermore, EquiTruss provides simple yet powerful community search functionalities in large-scale graphs, and thus can be effectively employed in the studies of real-world networked data. We summarize the contributions of EquiTruss as follows,

- We introduce a novel notion, k-truss equivalence, to capture the intrinsic relationship of edges in truss-based communities. Based on this new concept, we can partition any graph G into a series of truss-preserving equivalence classes for community search (Section 3);
- We design and develop a truss-equivalence based index, EquiTruss, that is space-efficient, cost-effective, and amenable to dynamic changes in the graph G. More importantly, community search can be performed directly upon EquiTruss without costly revisits to G, which is theoretical optimal (Section 4);
- We carry out extensive experimental studies in real-world, large-scale graphs, and compare EquiTruss with the state-of-the-art solution, TCP-Index. Experimental results demonstrate that EquiTruss is smaller in size, faster to be constructed and maintained, and admits at least an order of magnitude speedup for community search in large graphs (Section 5);

The remainder of the paper is organized as follows. In Section 2, we formulate the truss-based community search problem and discuss the related work. In particular, we examine the state-of-the-art solution, TCP-Index, and analyze its weaknesses. In Section 3, we introduce a novel notion, k-truss equivalence, for truss-based community modeling and search. We present the truss-equivalence based indexing solution, EquiTruss, in Section 4. Experimental studied and key findings are reported in Section 5, followed by concluding remarks in Section 6.

2. BACKGROUND

In this section, we discuss the preliminary concepts for the community search problem in graphs. A primer of terminologies and notations in this paper is briefed in Table 1. We also elaborate on related work with a special focus on the state-of-the-art solution, TCP-Index [11], and discuss its limitations for community search.

2.1 Preliminaries

We consider in this paper an undirected, connected, simple graph $G = (V_G, E_G)$, where V_G is a set of vertices, and $E_G \subseteq V_G \times V_G$ is a set of edges. Given a vertex $v \in V_G$, we denote the set of neighboring vertices of v as $N_G(v)$, where $N_G(v) = \{u | u \in V_G : (u, v) \in E_G\}$, and the degree of v is $d(v) = |N_G(v)|$. We use d_{max} to denote the maximum degree of vertices in G. A triangle Δ_{uvw} is a cycle of length three comprising three distinct vertices $u, v, w \in V_G$. Based on triangles, we define the following key concepts,

DEFINITION 1 (**Edge Support**). *The support of an edge $e = (u, v) \in E_G$, denoted as $sup_G(e)$, is the number of triangles with e as a constituent edge, i.e., $sup_G(e) = |\{\Delta_{uvw} : w \in V_G\}|$.* □

DEFINITION 2 (**Subgraph Trussness**). *Given a subgraph $G'(V', E') \subseteq G$, the trussness of G', denoted as $\tau(G')$, is the largest integer k ($k \geq 2$), such that $\tau(G') = \text{argmax}_k \{sup_{G'}(e) \geq k - 2 : \forall e \in E'\}$.* □

Consider a subgraph $G' \subseteq G$. If $\tau(G') = k$ and there exists no super-graph G'' of G' ($G' \subseteq G'' \subseteq G$) such that $\tau(G'') = \tau(G') = k$, G' is referred to as a **maximal k-truss**, or k-truss for short. Given a fixed value of k, there

Table 1: A primer of terminologies and notations

Notation	Decription
$G = (V_G, E_G)$	A undirected, simple graph G
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	A summarized graph $\mathcal{G}$ in EquiTruss
u, v, w, x, y, z	Vertices in V_G of G
ν, μ, ψ	Super-nodes in $\mathcal{V}$ of $\mathcal{G}$
$N_G(v)$	The set of neighboring vertices of $v \in V_G$
Δ_{uvw}	A triangle formed by vertices u, v, w
$sup(e)$	The support of e, $e \in E_G$
$\tau(G')$, $\tau(e)$	The trussness of a graph G', or an edge e
$\Delta_s \leftrightarrow \Delta_t$, $e_1 \leftrightarrow e_2$	Δ_s and Δ_t (e_1 and e_2) are triangle connected
$\Delta_s \overset{k}{\leftrightarrow} \Delta_t$, $e_1 \overset{k}{\leftrightarrow} e_2$	Δ_s and Δ_t (e_1 and e_2) are k-triangle connected
$e_1 \overset{k}{=} e_2$	Edges e_1 and e_2 are k-truss equivalent

exists only one k-truss in G due to its maximality, while a k-truss is not necessarily a connected graph. Therefore, the classic definition of k-truss is not suitable to directly model real-world communities that are both densely and cohesively connected. To tackle this issue, the *triangle-connectivity* constraint is further imposed upon k-truss:

Definition 3 (**Triangle Adjacency**). *Given two triangles Δ_1 and Δ_2 in G, they are adjacent if Δ_1 and Δ_2 share a common edge; that is, $\Delta_1 \cap \Delta_2 \neq \emptyset$.* □

Definition 4 (**Triangle Connectivity**). *Consider two triangles Δ_s and Δ_t in G. They are connected, denoted as $\Delta_s \leftrightarrow \Delta_t$, if there exist a sequence of n triangles $\Delta_1, \ldots, \Delta_n$ in G ($n \geq 2$), such that $\Delta_s = \Delta_1$, $\Delta_t = \Delta_n$, and for $1 \leq i < n$, $\Delta_i \cap \Delta_{i+1} \neq \emptyset$.* □

Analogously, for any two edges $e, e' \in E_G$, they are *triangle-connected*, denoted as $e \leftrightarrow e'$, if and only if (1) e and e' belong to the same triangle, *i.e.*, $e, e' \in \Delta$, or (2) $e \in \Delta_s$, $e' \in \Delta_t$, *s.t.* $\Delta_s \leftrightarrow \Delta_t$. To this end, the truss-based community can be defined as follows,

Definition 5 (**k-truss Community**). *Given a graph G and an integer $k \geq 3$[1], a subgraph $G' \subseteq G$ is a k-truss community if it satisfies the following conditions:* (1) *G' is a k-truss;* (2) *$\forall e, e' \in E_{G'}$, $e \leftrightarrow e'$.* □

Example 2. *Given the toy graph G as shown in Figure 1(a), we consider a subgraph G' induced by the set of vertices $\{v_7, v_8, v_9, v_{10}, v_{11}\}$. We note that for each edge e in G', we have $sup_{G'}(e) = 3$ meaning that e is involved in at least three different triangles. As a result, the subgraph trussness of G', $\tau(G')$, is 5, and G' is actually a 5-truss community because for any pair of edges in G', they are triangle connected. For example, $(v_7, v_8) \leftrightarrow (v_{10}, v_{11})$, because $(v_7, v_8) \in \Delta_{7,8,11}$, $(v_{10}, v_{11}) \in \Delta_{8,10,11}$, and $\Delta_{7,8,11} \cap \Delta_{8,10,11} = \{(v_8, v_{11})\}$.* □

It has been well recognized that triangle is a fundamental, higher-order graph motif representing a strong and stable relationship in graphs [2, 16], and the community modelling based on triangles, rather than primitive vertices/edges, results in more accurate communities in real-world graphs [2, 11]. In Definition 5, the high-density and cohesiveness of communities are guaranteed, respectively: condition (1) ensures the community is a densely-connected subgraph modeled by k-truss, and condition (2) ensures all edges within a community are cohesively connected via strong and stable triangle motifs. Furthermore, this truss-based community definition allows a vertex to participate in multiple communities [11]. We finally define the truss-based community search problem as follows,

[1]In the classic k-truss definition, $k = 2$ indicates a degraded case where a (sub)graph has no triangles involved. Such a graph is neither densely nor cohesively connected, and thus is omitted in our discussion.

Definition 6 (**k-truss community search**). *Given a graph $G(V_G, E_G)$, a query vertex $q \in V_G$, and an integer $k \geq 3$, find all k-truss communities containing q.* □

2.2 Related Work

Community search. As a query-dependent variant of the well-known community detection problem, community search is to find cohesive and densely connected subgraphs involving a given query vertex (or a set of query vertices) in a graph. Community search was first explored when communities were modeled as k-core (each vertex in a k-core has its degree no less than k) with distance and size constraints, rendering the problem NP-hard [29]. Online search of overlapping communities for a query vertex in terms of α-adjacency γ-quasi-k-cliques was further proposed [7], while the resultant communities may not be cohesively connected, compared with truss-based communities [11]. Influential community search, which is not query-dependent, aims to discover top-r most influential k-core communities [21]. Local community search [34, 1, 7] is to identify k-cores that contain query vertices and also maximize/minimize some goodness metric of communities, such as density, modularity, and graph conductance. However, many "free-rider" vertices irrelevant to the query vertex are inevitably returned for some goodness metrics. Furthermore, only a single community is identified, which fails to account for the real-world cases where a query vertex may participate in different communities. Community search was also examined in attributed graphs [9, 12] and spatial graphs [8].

Truss. Dense, cohesive subgraphs are critical components revealing potential community structures of real-world, massive graphs. There has been a rich literature in modelling and quantification of dense and cohesive graphs, including clique or quasi-clique [6, 31], k-core [25, 1, 15], and nucleus [26, 27]. All the aforementioned models except k-core suffer from the computational intractability problem [28], whereas k-core may result in incohesive subgraphs [37, 5]. Truss is defined on the higher-order graph motif, triangle, and enjoys numerous advantages in community modelling and computation [5]: a k-truss is a $(k-1)$-core but not vice versa [5]; a k-truss is $(k-1)$-edge-connected: any deletion of fewer than $(k-1)$ edges will not disconnect a k-truss; a k-truss with n vertices has its diameter no more than $\lfloor \frac{2n-2}{k} \rfloor$; that is, a k-truss is diameter-bounded [13]. All these properties are critical indicators of good communities [28]. In addition, k-truss based communities exhibit an inclusive hierarchy representing *cores* of a graph at different levels of granularity; that is, a k-truss is contained in a $(k-1)$-truss for $k \geq 3$. When augmented with the triangle connectivity constraint, k-truss can account for the more practical case where a vertex may belong to multiple k-truss communities, which is consistent with sociological studies [11, 10]. In [5], the authors designed polynomial algorithms to find k-trusses from a graph and extended the methods to MapReduce. An improved algorithm was further proposed based on triangle enumeration in $O(|E_G|^{3/2})$ time [33]. A similar I/O-efficient algorithm for k-truss computation was designed and facilitated by graph database technologies [37]. A parallel method, PETA, can detect local k-trusses within a few iterations, and it has the same complexity as corresponding serial algorithms [28].

Due to its advantages in community modeling and computation, truss has been extensively applied for community

search. In [13], community search was reformulated as an NP-hard problem that identifies a closest k-truss community with the minimum diameter and the largest k containing *a set of* query vertices. A greedy algorithm with compact indexes was proposed for approximate solutions. Truss has also been extended for community search in probabilities graphs [14] and attributed graphs [12].

Graph Summarization. As the scale and complexity of graphs increase, graph summarization techniques have been explored toward simplifying massive graphs into succinct and quality-preserving *summaries* for significant reduction in graph storage and computation cost [17]. GraSS [19] summarizes graphs by greedily grouping vertices into a probabilistic adjacency matrix, upon which neighborhood queries can be efficiently approximated. In [24], graphs are summarized to super-nodes and super-edges with guaranteed reconstruction error. The compressed summary is used to approximate queries including adjacency, degree, eigenvector centrality, and subgraph counting. In [22], the authors devised greedy and randomized algorithms to compress graphs with bounded minimum discription length (MDL) errors. VOD [17] is a vocabulary-based graph summarization method aimed at minimizing the information-theoretic encoding cost of graphs. SNAP [30] groups vertices based on attributes, and iteratively splits groups until reaching the maximum attribute- and relationship-compatible grouping.

Graph summarization is problem-driven, and typically optimized toward application-dependent objectives. However, there exist no prior graph summarization methods for the community search problem, as addressed in this paper. To our knowledge, no existing research has explored the intrinsic relationships of edges within k-truss communities, which lead to the summarized, community-preserving index, EquiTruss, as proposed in this paper.

TCP-Index. The state-of-the-art solution to truss-based community search is TCP-Index (Triangle Connectivity Preserved Index) [11], which maintains trussness values and triangle-adjacency information of the pre-computed k-trusses into a group of tree-structured indexes. Specifically, for each vertex $x \in V_G$, we consider the vertex-centric *ego-net* G_x, where $V_{G_x} = N_G(x)$ and $E_{G_x} = \{(y,z)|(y,z) \in E_G, y,z \in N_G(x)\}$. The edge $(y,z) \in E_{G_x}$ is further assigned a weight w indicating that a triangle Δ_{xyz} arises in a k-truss community ($w \geq k$). Given the weighted graph G_x, a maximum spanning tree (MST), $\mathcal{T}_x$, is identified, and all $\mathcal{T}_x$'s ($\forall x \in V_G$) constitute TCP-Index of G. We note that, for any two vertices connected through a series of edges with weights no less than k in $\mathcal{T}_x$, they belong to the same k-truss community. Namely, the community structures are *losslessly compressed* in TCP-Index. However, during community search, a series of costly, *decompression-like* operations have to be undertaken *online* in both TCP-Index and the original graph G to fully reconstruct edges of resultant k-truss communities. For instance, if any edge $(u,v) \in \mathcal{T}_x$ is in a k-truss community, we have to examine both $\mathcal{T}_u$ and $\mathcal{T}_v$ in TCP-Index, and revisit G to find missing edges of the community, thus inevitably incurring expensive computational cost.

The limitations of TCP-Index are summarized as follows: (1) given any k-truss community of G, its constituent edges have to be examined and maintained redundantly in different MSTs, thus rendering the construction of TCP-Index extremely time-consuming and the resultant index excessively large; (2) during community search, a costly truss-reconstruction process has to been undertaken by repeated accesses to both TCP-Index and G, thus making community search inefficient; (3) when G evolves, the dynamic maintenance of TCP-Index becomes complicated and time-consuming. For instance, when a new edge is inserted to G or an existing edge is removed from G, a significant fraction of MSTs in TCP-Index need to be updated accordingly, which is time-consuming. As a result, TCP-Index may fail in supporting online, efficient community search especially in real-world, massive graphs.

3. TRUSS EQUIVALENCE

To systematically address the limitations of TCP-Index and enable efficient community search, we propose a new notion, *k-truss equivalence*, to characterize a fundamental equivalence relation for edges that are *strongly* connected in a k-truss community. As a consequence, a truss-equivalence based index, EquiTruss, can be developed that is theoretically optimal for community search. To start with, we consider a preprocessing step to decompose an input graph G into k-trusses ($k \geq 2$). Given an edge $e \in E_G$, we first define edge trussness of e as follows,

DEFINITION 7 (**Edge Trussness**). *The trussness of an edge $e \in E_G$, denoted as $\tau(e)$, is the maximum subgraph trussness of a subgraph $G^* \subseteq G$ that involves e as a constituent edge, i.e., $\tau(e) = \max_{G^* \subseteq G}\{\tau(G^*) : e \in E_{G^*}\}$.* □

It is important to note that a (maximal) k-truss of G consists of all the edges with edge trussness no less than k. We thus can apply a truss decomposition algorithm [33], as detailed in Algorithm 1, to compute edge trussness and discover all k-trusses from G. The algorithm starts with an initialization step to compute edge supports in $O(|E|^{1.5})$ time using existing triangle enumeration methods [23, 18] (Line 1). After the initialization, for k starting from 2, we iteratively select the edge $e^*(u,v)$ with the lowest support (Line 5), assign the edge trussness k to e^*, and remove it from G (Line 11). Meanwhile, we decrement the support of all the other edges forming triangles with e^*, and reorder them based on their new edge support (Lines 7-10). This process continues until all the edges with edge support no greater than $(k-2)$ are removed from G (Line 4). If there are still edges left in G, we increment k by one to process the edges with edge trussness $(k+1)$ (Lines $12-14$). The time complexity of Algorithm 1 is $O(|E_G|^{1.5})$ and its space complexity is $O(|V_G| + |E_G|)$ [33].

EXAMPLE 3. *We apply Algorithm 1 in the graph G (Figure 1(a)) to compute edge trussness for all edges of G, and the results are presented in Figure 2. Edges with different edge trussness values are illustrated in different colors.* □

We further define a stronger triangle-connectivity constraint: *k-triangle connectivity*, as follows,

DEFINITION 8 (**k-triangle**). *Given a triangle $\Delta_{uvw} \subseteq G$, if edge trussnesses of all the three constituent edges are no less than k, i.e., $\min\{\tau(u,v), \tau(v,w), \tau(u,w)\} \geq k$, Δ_{uvw} is denoted as a k-triangle.* □

DEFINITION 9 (**k-triangle connectivity**). *Given two k-triangles Δ_s and Δ_t in G, they are k-triangle connected, denoted as $\Delta_s \overset{k}{\leftrightarrow} \Delta_t$, if there exists a sequence of $n \geq 2$ k-triangles $\Delta_1, \ldots, \Delta_n$ s.t. $\Delta_s = \Delta_1$, $\Delta_t = \Delta_n$, and for $1 \leq i < n$, $\Delta_i \cap \Delta_{i+1} = \{e|e \in E_G\}$ and $\tau(e) = k$.* □

Algorithm 1: Truss Decomposition

Input: A graph $G(V_G, E_G)$
Output: The edge trussness $\tau(e)$ for each $e \in E_G$

```
Computer sup(e) for each edge e ∈ E_G;
Sort all edges in the non-decreasing order of their support;
k ← 2;
while ∃e ∈ E_G, sup(e) ≤ (k − 2) do
    e*(u, v) ← arg min_{e∈E_G} sup(e);
    assume w.l.o.g. d(u) ≤ d(v);
    foreach w ∈ N(u) and (v, w) ∈ E_G do
        sup(u, w) ← sup(u, w) − 1;
        sup(v, w) ← sup(v, w) − 1;
        Reorder (u, w) and (v, w) w.r.t. new edge support;
    τ(e*) ← k, remove e* from E_G;
if ∃e ∈ E_G then
    k ← k + 1;
    goto Step 4;
return {τ(e)|e ∈ E_G};
```

EXAMPLE 4. *Consider the graph G as shown in Figure 2, and two 4-triangles $\Delta_{4,5,7}$ and $\Delta_{6,8,11}$. They are 4-triangle connected as there are two 4-triangles $\Delta_{5,6,7}$ and $\Delta_{6,7,8}$, such that $\Delta_{4,5,7} \cap \Delta_{5,6,7} = \{(5,7)\}$, $\Delta_{5,6,7} \cap \Delta_{6,7,8} = \{(6,7)\}$, $\Delta_{6,7,8} \cap \Delta_{6,8,11} = \{(6,8)\}$, and edges trussness values of all these edges are 4. However, the two 3-triangles $\Delta_{1,4,5}$ and $\Delta_{3,4,7}$ are not 3-triangle connected.* □

Intuitively, if $\Delta_s \overset{k}{\leftrightarrow} \Delta_t$, the two k-triangles, Δ_s and Δ_t, are connected by a series of k-triangles with a chain of *join edges* (common edges shared by consecutive triangles) that have the edge trussness k. Analogously, we say two edges $e, e' \in E_G$ are *k-triangle connected*, denoted as $e \overset{k}{\leftrightarrow} e'$, if and only if (1) e and e' belong to the same k-triangle, or (2) $e \in \Delta_s$, $e' \in \Delta_t$, *s.t.* $\Delta_s \overset{k}{\leftrightarrow} \Delta_t$. To this end, we define a new relation, *k-truss equivalence*, upon E_G, as follows,

DEFINITION 10 (**k-truss equivalence**). *Given any two edges $e_1, e_2 \in E_G$, they are k-truss equivalent ($k \geq 3$), denoted as $e_1 \overset{k}{=} e_2$, if and only if* (1) $\tau(e_1) = \tau(e_2) = k$, *and* (2) $e \overset{k}{\leftrightarrow} e'$. □

THEOREM 1. *k-truss equivalence is an equivalence relation upon E_G.* □

PROOF. k-truss equivalence is a binary relation defined upon E_G, and we prove the following key properties of an equivalence relation for k-truss equivalence:

Reflexivity. Consider an edge $e_0 \in E_G$, *s.t.* $\tau(e_0) = k$. Based on Definition 7, there exists at least one subgraph $G^*(V^*, E^*) \subseteq G$ such that $e_0 \in E^*$, and $\forall e \in E^*, \tau(e) \geq k$. Since $k \geq 3$, there exists at least one k-triangle $\Delta \subseteq G^*$ such that $e_0 \in \Delta$. Namely, $e_0 \overset{k}{=} e_0$;

Symmetry. Consider two edges $e_1, e_2 \in E_G$, $e_1 \overset{k}{=} e_2$. That is, $\tau(e_1) = \tau(e_2) = k$, and either of the following cases holds: (1) e_1 and e_2 are in the same k-triangle; (2) there exist two k-triangles Δ_1 and Δ_2, such that $e_1 \in \Delta_1$, $e_2 \in \Delta_2$, and $\Delta_1 \overset{k}{\leftrightarrow} \Delta_2$. For case (1), as e_2 is located in the same k-triangle as e_1, so $e_2 \overset{k}{=} e_1$. For case (2), note that k-triangle connectivity is symmetric, so $\Delta_2 \overset{k}{\leftrightarrow} \Delta_1$. Namely, $e_2 \overset{k}{=} e_1$.

Transitivity. Consider three edges $e_1, e_2, e_3 \in E_G$, *s.t.* $e_1 \overset{k}{=} e_2$ and $e_2 \overset{k}{=} e_3$. Namely $\tau(e_1) = \tau(e_2) = \tau(e_3) = k$, and either of the following cases holds: (1) there exist two k-triangles Δ_1 and Δ_2, such that $e_1, e_2 \in \Delta_1$ and $e_2, e_3 \in \Delta_2$. If $\Delta_1 = \Delta_2$, e_1 and e_3 are located in the same k-triangle, so $e_1 \overset{k}{=} e_3$. Otherwise, $\Delta_1 \cap \Delta_2 = \{e_2\}$ and $\tau(e_2) = k$, so $\Delta_1 \overset{k}{\leftrightarrow} \Delta_2$. Therefore, $e_1 \overset{k}{=} e_3$; (2) there exist $m(\geq 2)$ k-triangles $\Delta_{l_1}, \ldots, \Delta_{l_m}$ in G, *s.t.* $e_1 \in \Delta_{l_1}$, $e_2 \in \Delta_{l_m}$, and all the edges joining these m consecutive k-triangles are with the same edge trussness, k. Meanwhile, there exist $n(\geq 2)$ k-triangles $\Delta_{t_1}, \ldots, \Delta_{t_n}$ in G *s.t.* $e_2 \in \Delta_{t_1}$, $e_3 \in \Delta_{t_n}$ and all the edges joining these n k-triangles are with the same edge trussness, k. If $\Delta_{l_m} = \Delta_{t_1}$, we know that $\Delta_{l_1} \overset{k}{\leftrightarrow} \Delta_{t_n}$ through a series of $(m+n-1)$ adjacent k-triangles $\Delta_{l_1}, \ldots, \Delta_{l_m}, \ldots, \Delta_{t_n}$. Otherwise, we know that $\Delta_{l_m} \cap \Delta_{t_1} = \{e_2\}$ and $\tau(e_2) = k$, so $\Delta_{l_1} \overset{k}{\leftrightarrow} \Delta_{t_n}$ through a series of $(m+n)$ adjacent k-triangles $\Delta_{l_1}, \ldots, \Delta_{l_m}, \Delta_{t_1}, \ldots, \Delta_{t_n}$. Therefore, $e_1 \overset{k}{=} e_3$. □

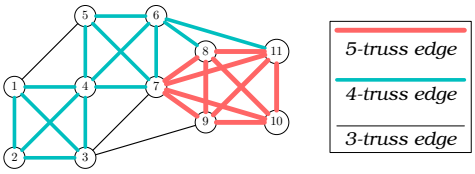


Figure 2: k-truss edges in the graph G

Given an edge $e \in E_G$, $\tau(e) = k$, the set $\mathcal{C}_e = \{e' | e' \overset{k}{=} e, e' \in E_G\}$ defines an *equivalence class* of e *w.r.t.* k-truss equivalence, and the set of all equivalence classes forms a *mutually exclusive* and *collectively exhaustive* partition of E_G. In particular, any equivalence class $\mathcal{C}_e$ consists of edges with the same edge trussness, k, that are also k-triangle connected, making $\mathcal{C}_e$ a k-truss community by definition.

4. TRUSS-EQUIVALENCE BASED INDEX

Based on k-truss equivalence, we design and develop a graph-structured index, EquiTruss (Section 4.1), which supports community search with theoretically optimal performance (Section 4.2). In addition, EquiTruss allows incremental update when G changes dynamically (Section 4.3).

4.1 Index Design and Construction

According to k-truss equivalence, all the edges of the graph G are partitioned into a series of mutually exclusive equivalence classes, each of which represents a k-truss community. We thus design a truss-equivalence based index, EquiTruss, as a summarized graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a super-node set and $\mathcal{E}$ is a super-edge set, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. A super-node $\nu \in \mathcal{V}$ represents a distinct equivalence class $\mathcal{C}_e$ where $e \in E_G$, and a super-edge $(\mu, \nu) \in \mathcal{E}$, where $\mu, \nu \in \mathcal{V}$, indicates that the two equivalence classes are triangle-connected; that is, $\exists e \in \mu$ and $\exists e' \in \nu$, *s.t.* $e \leftrightarrow e'$. It is important to recognize that EquiTruss is a community-preserving graph summary, where all k-truss communities are completely encoded in super-nodes, and the triangle connectivity across different communities is exactly maintained in super-edges, thus making all the information critical to community search readily available in EquiTruss. Furthermore, each edge e of G is maintained in exactly one super-node representing its k-truss equivalence class, $\mathcal{C}_e$. In comparison to TCP-Index where e has to be maintained redundantly in multiple MSTs, EquiTruss is significantly more succinct and space-efficient.

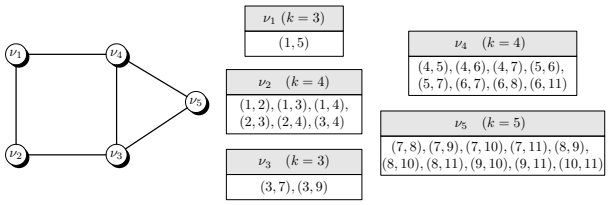


Figure 3: Truss-equivalence based index, EquiTruss

Example 5. *The truss-equivalence based index,* EquiTruss*, of the graph G (Figure 1(a)) is shown in Figure 3. It contains* 5 *super-nodes representing k-truss equivalence classes for edges in G, as tabulated in Figure 3. For example, the super-node ν_2 represents a* 4*-truss community with* 6 *edges: they are* 4*-triangle connected, and have the same edge trussness value of* 4*. Meanwhile, there are* 6 *super-edges in* EquiTruss *depicting triangle connectivity between super-nodes (k-truss communities).*

Given the graph G, we construct the truss-equivalence based index, EquiTruss, in Algorithm 2. In the initialization phase (Lines 1-7), we first call Algorithm 1 to compute edge trussness for each edge $e \in E_G$ (Line 1), then reallocate edges to different sets, Φ_k, in terms of edge trussness (Lines 5-6). Given $e \in E_G$, we maintain two auxiliary data structures: **processed** is a Boolean variable indicating whether e has been examined in index construction, and is initialized to FALSE (Line 3); **list** is a set of super-node identifies, each of which represents a previously explored super-node, μ, where $\tau(\mu) < k$, and μ is triangle-connected to the current super-node, ν ($\tau(\nu) = k$), via the edge e. The e.list is initialized as an empty set (Line 4). We then examine all the edges of G in a non-decreasing order of edge trussness from Φ_3 to $\Phi_{k_{max}}$ consecutively (Line 8). When selecting an edge $e \in \Phi_k$, we create a new super-node ν corresponding to the equivalence class $\mathcal{C}_e$ of e (Lines 10-12). Using $e = (u, v)$ as an initial seed, we traverse G (in BFS) to identify all the edges k-truss equivalent to e by exploring its incident k-triangles (Line 20-23), and add them to the super-node ν. Meanwhile, we also check if there exists some super-node μ in e.list, where $\tau(\mu) < \tau(\nu) = k$, and μ is triangle-connected to ν through e. If so, we create a super-edge (μ, ν) in the index (Lines 17-19). Given any k-triangle, if there exists an edge e' with $\tau(e') > k$, the identifier of the current super-node ν will be subscribed to e'.list as ν is triangle-connected to the super-node to which e' belongs, and a super-edge will be created when e' is processed (Lines 31-33). After e and all its incident triangles are examined, e is removed from both Φ_k and E_G (Line 24), ensuring that each edge e belongs to at most one k-truss equivalence class represented by the super-node ν.

Theorem 2. EquiTruss *can be constructed in $O(|E_G|^{1.5})$ time and $O(|E_G|)$ space by Algorithm 2.* □

Proof. In the initialization phase of Algorithm 2 (Lines 1-7), the truss decomposition costs $O(|E_G|^{1.5})$ time. In the index construction phase (Lines 8-24), for each edge $e = (u, v) \in E_G$, we consider all the triangles Δ_{uvw} that involve e in order to identify the k-truss equivalent edges. Then e is eliminated from Φ_k and E_G, making each triangle Δ_{uvw} examined only once. The procedure ProcessEdge takes $O(1)$ time. So the index construction of EquiTruss is equivalent to enumerating all triangles from G in $O(|E_G|^{1.5})$ time.

Given an edge $e \in E_G$, the size of e.list is equivalent to the number of super-nodes, μ, where $\tau(\mu) < \tau(e)$, and μ is triangle connected to the super-node to which e belongs. So e.list takes at most $O(|E_G|)$ space. Once e is processed, it will be removed from G and the space of e.list is released. So the space complexity of Algorithm 2 is $O(|E_G|)$. □

Algorithm 2: Index Construction for EquiTruss

Input: A graph $G(V_G, E_G)$
Output: EquiTruss: $\mathcal{G}(\mathcal{V}, \mathcal{E})$

```
   /* Initialization */
1  Truss Decomposition (G);
2  foreach e ∈ E_G do
3      e.processed ← FALSE;
4      e.list ← ∅;
5      if τ(e) = k then
6          Φ_k ← Φ_k ∪ {e};
7  snID ← 0 ;          /* Super-node ID initialized to 0 */
   /* Index Construction */
8  for k ← 3 to k_max do
9      while ∃e ∈ Φ_k do
10         e.processed = TRUE;
11         Create a super-node ν with ν.snID ← ++snID;
12         V ← V ∪ {ν} ;        /* A new super-node for C_e */
13         Q.enqueue(e) ;
14         while Q ≠ ∅ do
15             e(u, v) ← Q.dequeue();
16             ν ← ν ∪ {e} ;      /* Add e to super-node ν */
17             foreach id ∈ e.list do
18                 Create a super-edge (ν, μ) where μ is an
                     existing super-node with μ.snID = id;
19                 E ← E ∪ {(ν, μ)} ;      /* Add super-edge */
20             foreach w ∈ N(u) ∩ N(v) do
21                 if τ(u, w) ≥ k and τ(v, w) ≥ k then
22                     ProcessEdge(u, w);
23                     ProcessEdge(v, w);
24             Φ_k ← Φ_k − {e}; E ← E − {e};
25 return G(V, E);

26 Procedure ProcessEdge(u, v)
27 if τ(u, v) = k then /* k-triangle connectivity */
28     if (u, v).processed = FALSE then
29         (u, v).processed = TRUE;
30         Q.enqueue(u, v);
31 else /* τ(u, v) > k */
32     if snID ∉ (u, v).list then
33         (u, v).list ← (u, v).list ∪ {snID};
```

In practice, EquiTruss is built offline before community search is performed, so it can be constructed efficiently from real-world, massive graphs. Meanwhile, EquiTruss is significantly more space-efficient than TCP-Index, as there are no redundant edges maintained in the index.

4.2 Community Search on EquiTruss

After EquiTruss is constructed from G, community search can be carried out directly on EquiTruss *without* repeated accesses to G, which is detailed in Algorithm 3. First of all, we find from the index $\mathcal{G}$ the super-nodes within which the query vertex q is located. We use a hash structure $\mathcal{H} : V_G \to 2^{\mathcal{V}}$ to maintain this information, where $\mathcal{H}(u) = \{\nu_1, \ldots, \nu_l\}$ as long as there exists some edge $(u, v) \in \nu_i (1 \le i \le l)$, where $u, v \in V_G$. We remark $\mathcal{H}$ can be efficiently built as a by-product in index construction. Starting from each super-node $\nu \in \mathcal{H}(q)$ with $\tau(\nu) \ge k$, we traverse $\mathcal{G}$ in a BFS fashion, and for each unexplored, neighboring super-node μ with $\tau(\mu) \ge k$, the edges within μ will be added to the

Algorithm 3: Community Search Based on EquiTruss

Input: $\mathcal{G}(\mathcal{V}, \mathcal{E})$, the truss value $k \geq 3$, the query vertex q
Output: $\mathcal{A}$: all k-truss communities containing q

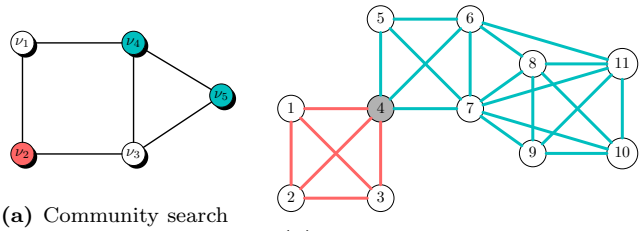
```
/* Initialization */
1  foreach ν ∈ V do
2      ν.processed ← FALSE;
3  l ← 0 ;
   /* BFS traversal for community search */
4  foreach ν ∈ H(q) do
5      if τ(ν) ≥ k and ν.processed = FALSE then
6          ν.processed ← TRUE;
7          l ← l + 1;    A_l ← ∅;
8          Q ← ∅;    Q.enqueue(ν);
9          while Q ≠ ∅ do
10             ν ← Q.dequeue();
11             A_l ← A_l ∪ {e|e ∈ ν};
12             foreach (ν, μ) ∈ E do
13                 if τ(μ) ≥ k and μ.processed = FALSE then
14                     μ.processed ← TRUE;
15                     Q.enqueue(μ);
16 return {A_1, ..., A_l};
```

k-truss community, $\mathcal{A}_l$. In the end, each A_l represents a k-truss community in which q is involved.

Theorem 3. *Given a query vertex $q \in V_G$ and the truss value k, Algorithm 3 correctly computes all k-truss communities containing v.* □

Proof. According to Algorithm 3, each set $\mathcal{A}_i (1 \leq i \leq l)$ satisfies the following conditions: (1) the query vertex q is located in $\mathcal{A}_i$; (2) for each edge $e \in \mathcal{A}_i$, $\tau(e) \geq k$; (3) all the edges in $\mathcal{A}_i$ are triangle-connected. Therefore, $\mathcal{A}_i$ is a k-truss with constituent edges triangle-connected. We then prove that all $\mathcal{A}_i (1 \leq i \leq l)$ are maximal by way of contradiction. Assume otherwise there exists at least one set $\mathcal{A}_i$ which is not maximal; that is, there exists a subgraph $\mathcal{A}' \subseteq G$, *s.t.* $\mathcal{A}_i \subset \mathcal{A}'$ and $\mathcal{A}'$ is one of the community search results satisfying the aforementioned conditions. As a consequence, there exists at least one edge $e \in \mathcal{A}' \setminus \mathcal{A}$, *s.t.* $\tau(e) \geq k$, and e is triangle connected to every edge in $\mathcal{A}'$, but is not triangle connected to any edge in $\mathcal{A}$. Because the query vertex q is located in both $\mathcal{A}$ and $\mathcal{A}'$, there exists at least one incident edge of q, denoted as (q, u), where $u \in V_G$, *s.t.* $\tau(q, u) \geq k$ and $(q, u) \in \mathcal{A}$. As $\mathcal{A} \subset \mathcal{A}'$, the edge (q, u) is also in $\mathcal{A}'$, *i.e.* $(q, u) \in \mathcal{A}'$. Therefore, (q, u) is triangle connected to the edge e, which contradicts with the fact that any edge of $\mathcal{A}$, including (q, u), is not triangle connected to e. □

Example 6. *Consider the graph G shown in Figure 1(a), the truss value $k = 4$, and the query vertex v_4. Based on Algorithm 3, We first find from* EquiTruss *(in Figure 3) the super-nodes ν_2 and ν_4 that contain v_4. Starting from ν_2, we recognize that $\tau(\nu_2) = 4 \geq k$, so all the edges within ν_2 are in the first community $\mathcal{A}_1$. However, ν_2's neighboring super-nodes ν_1 and ν_3 are disqualified because $\tau(\nu_1) = \tau(\nu_3) = 3 < k$. We then start with the second super-node ν_4. As $\tau(\nu_4) = 4 > k$, all the edges within ν_4 are in the second community $\mathcal{A}_2$. Furthermore, since $(\nu_4, \nu_5) \in \mathcal{E}$ and $\tau(\nu_5) = 5$, ν_5 is also qualified and all the edges within ν_5 are in the community $\mathcal{A}_2$ as well. The whole community search process is illustrated in Figure 4(a) and the community search results including $\mathcal{A}_1$ (colored in red) and $\mathcal{A}_2$ (colored in green) are presented in Figure 4(b).* □



(a) Community search on EquiTruss **(b)** 4-truss communities of vertex 4

Figure 4: The two 4-truss communities for the query vertex v_4, including $\mathcal{A}_1$ with edges in red color and $\mathcal{A}_2$ with edges in green color.

Theorem 4. *The time complexity of Algorithm 3 is determined solely by the size of the resultant k-truss communities,* i.e., $O(|\bigcup_{i=1}^{l} \mathcal{A}_i|)$. □

Proof. In Algorithm 3, each edge of $\mathcal{A}_i$ is accessed only once when reported as output. For any edge in the super-node ν where $\tau(\nu) < k$, it is not even accessed in the algorithm. As a result, the time complexity of Algorithm 3 is $O(|\bigcup_{i=1}^{l} \mathcal{A}_i|)$, which is exactly the time used for listing all the edges in the resultant k-truss communities from G. □

Based on Theorem 4, we note that Algorithm 3 is optimal as returning all the edges of the k-truss communities requires $\Omega(|\bigcup_{i=1}^{l} \mathcal{A}_i|)$ time, and Algorithm 4 achieves this lower-bound by visiting each edge of the resultant communities *exactly* once. It is also worth mentioning that, in Algorithm 3, we do not need to revisit the original graph G, and it suffices to simply leverage EquiTruss for community search. In comparison to TCP-Index that needs repeated accesses to G for community recovery, our method, EquiTruss, is significantly more efficient.

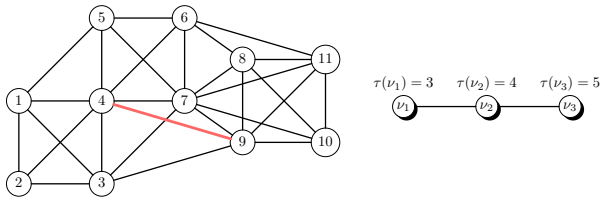
4.3 Dynamic Maintenance of EquiTruss

Real-world graphs are not static but dynamically evolving all the time. In this section, we examine how EquiTruss can be dynamically updated in accordance with the evolution of G. We consider two cases of changes in G: edge insertion and edge deletion, as vertex insertion/delection can be treated as a series of insertions/deletions for incident edges of the vertex to be inserted/deleted.

Inserting a new edge $e^* = (u, v)$ in G may create a set of new triangles $\{\Delta_{uvw} : w \in N(u) \cap N(v)\}$, thus resulting in an increment of edge support for (u, w) and (v, w) by 1. Meanwhile, for any subgraph G' that contains the triangle Δ_{uvw}, its subgraph trussness, $\tau(G')$, may increase as well because $\tau(G') = \min_{e \in E_{G'}} \{sup_{G'}(e) + 2\}$ (a variant of Definition 2). Furthermore, edge trussness of the edges in G' may also increase because $\tau(e) = \max_{G' \subseteq G} \{\tau(G') : e \in E(G')\}$ (Definition 7). As a result, a chain of edge-trussness increases may arise in G, requiring EquiTruss be updated accordingly. Similarly, deleting an existing edge $e^* = (u, v)$ from G may trigger a chain of edge-trussness decreases. In the following, we will first identify the affected super-nodes of EquiTruss in the presence of edge insertion/deletion in G, and then design dynamic update algorithms for EquiTruss.

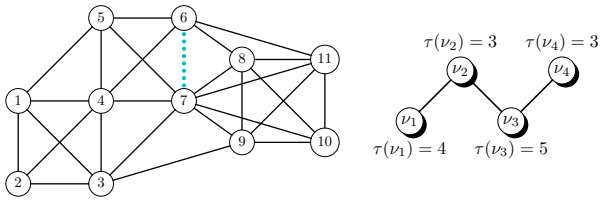
4.3.1 Affected Super-nodes in EquiTruss

We use $\tau(e)$ and $\hat{\tau}(e)$ to denote the trussness of $e \in E_G$ before and after an edge is inserted or deleted, respectively, into G. The affected super-nodes in EquiTruss are determined in the presence of an edge insertion or deletion, respectively, as follows:



(a) Edge Insertion **(b)** Updated $\mathcal{G}$

Figure 5: (a) Insert a new edge (v_4, v_9) in G; (b) The updated $\mathcal{G}$ in EquiTruss.



(a) Edge Deletion **(b)** Updated $\mathcal{G}$

Figure 6: (a) Delete an edge $(6, 7)$ from the graph G; (B) The updated $\mathcal{G}$ in EquiTruss.

Edge Insertion. According to existing theoretical results [11], we note that inserting a new edge $e^* = (u, v)$ may trigger edge-trussness increases as follows: $\forall e \in E_G \cup \{e^*\}$ with $\tau(e)(= k) < \hat{\tau}(e)$, where $\hat{\tau}(e)$ is the upper bound of $\tau(e)$ after e^* is inserted, if $e \overset{k}{\leftrightarrow} e^*$ holds, the trussness of e may be updated as $\hat{\tau}(e) = \tau(e) + 1$. The following theorem indicates the super-nodes of EquiTruss to be affected presumably due to an edge insertion:

THEOREM 5. *Consider an inserted edge $e^* = (u, v)$ and $w \in N(u) \cap N(v)$. We use e to denote either (w, u) or (w, v). For any k-triangle Δ_{uvw} where $k < \hat{\tau}(e)$, the following super-nodes in* EquiTruss *may be updated: $\{\nu | \nu \in \mathcal{V}, e \in \nu, \tau(e) = k\}$. Note e can be either (w, u) or (w, v) (or both) with the edge trussness k.* □

PROOF. Consider a k-triangle Δ_{uvw} that contains e^*. We assume *w.l.o.g.* that $e = (w, u)$ and $\tau(e) = k < \hat{\tau}(e)$. Based on the aforementioned results, for any edge e' that is k-triangle connected to e, its edge trussness, $\tau(e')$, may be updated. However, because $e' \overset{k}{\leftrightarrow} e$, e' belongs to the equivalence class of e, $\mathcal{C}_e$, which is exactly represented by the super-node ν in EquiTruss, where $e \in \nu$. □

According to Theorem 5, when a new edge e^* is inserted to G, all the edges with a potential edge-trussness increase, except e^*, are contained in the affected super-nodes of EquiTruss. As a result, we avoid re-examining the original graph G to find the affected edges, and hence save significant computational cost.

EXAMPLE 7. *We insert a new edge (v_4, v_9) in G, as presented in Figure 5(a), and $\hat{\tau}(v_4, v_9) = 4$. We examine two triangles $\Delta_{4,7,9}$ and $\Delta_{3,4,9}$ where the edge (v_4, v_9) is involved. First, because $\tau(v_4, v_7) = 4$ and $\tau(v_7, v_9) = 5$, which are no less than $\hat{\tau}(v_4, v_9) = 4$, both the edges (v_4, v_7) and (v_7, v_9) (together with their k-triangle connected edges) are not updated. However, in the triangle $\Delta_{3,4,9}$, we have $\tau(v_3, v_4) = 4$ and $\tau(v_3, v_9) = 3 < 4$. We recognize that the super-node ν_3 (in Figure 3) involves the edge (v_3, v_9), so ν_3 will be affected after the insertion of (v_4, v_9), As a result, all the edges within ν_3 including (v_3, v_7), (v_3, v_9), and the newly inserted edge (v_4, v_9), will be re-examined as their edge trussness may increase, based on Theorem 5. Figure 5(b) illustrates the updated summarized graph $\mathcal{G}$ in* EquiTruss *after the insertion of the edge (v_4, v_9) in G.* □

Edge Deletion. When an edge $e^* = (u, v)$ is deleted from G, we recognize analogously that a chain of edge-trussness decreases may arise as follows: for any edge $e \in E_G \backslash \{e^*\}$ with $\tau(e)(= k) \leq \tau(e^*)$, if $e \overset{k}{\leftrightarrow} e^*$ holds, the trussness of e may be updated as $\hat{\tau}(e) = \tau(e) - 1$. The following theorem indicates the super-nodes of EquiTruss to be affected presumably due to an edge deletion:

THEOREM 6. *Consider an edge $e^* = (u, v)$ to be deleted from G, and $e^* \in \nu$ where ν is a super-node of* EquiTruss *containing e^*. The following super-nodes of* EquiTruss *may be updated: $\{\nu\} \cup \{\mu | \tau(\mu) < \tau(e^*), \mu \in \mathcal{V}, (\mu, \nu) \in \mathcal{E}\}$.* □

PROOF. Given any edge $e \in E_G \backslash \{e^*\}$ *s.t.* $\tau(e)(= k) \leq \tau(e^*)$, we consider the following cases: (1) if e, e^* are in the same k-triangle and $k < \tau(e^*)$, e belongs to another super-node μ ($\tau(\mu) = k$) that is triangle connected to ν to which e^* belongs, so μ and ν are neighboring super-nodes in EquiTruss, and μ may be updated; (2) if e, e^* are in the same k-triangle and $k = \tau(e^*)$, or $e \overset{k}{\leftrightarrow} e^*$, e and e^* belong to the same super-node ν, which may be updated. Therefore, when e^* is deleted from G, the super-node ν and its neighboring super-nodes μ, where $\tau(\mu) < \tau(e^*)$, may be updated. □

According to Theorem 6, if an existing edge e^* is deleted from G, only the super-node ν that contains e^* and ν's neighboring super-nodes μ, where $\tau(\mu) < \tau(e^*)$, need to be re-examined for update. We thus save significant computational cost for exploring the whole graph G in order to identify the affected edges, as has been done in TCP-Index.

EXAMPLE 8. *We delete the edge (v_6, v_7) from G, as shown in Figure 6(a). As $\tau(v_6, v_7) = 4$, and the edge (v_6, v_7) belongs to the super-node ν_4 (Figure 3), all the edges within ν_4 may be affected. Indeed, all the edges (v_4, v_5), (v_4, v_6), (v_4, v_7), (v_5, v_6), (v_5, v_7), (v_6, v_8), and (v_6, v_{11}) within ν_4 have their edge trussness decreased by 1. As the edge (v_6, v_7) is not located within any k-triangle where $k < 4$, we can omit examining the adjacent super-nodes of ν_4 as they will not be affected by the deletion of (v_6, v_7). Figure 6(b) illustrates the updated summarized graph $\mathcal{G}$ in* EquiTruss *after the deletion of the edge (v_6, v_7) from G.* □

4.3.2 Dynamic Update for EquiTruss

For an edge insertion/deletion, we denote the set of affected edges in G as E', and the set of affected super-nodes in EquiTruss as $\mathcal{V}'$. Once E' and $\mathcal{V}'$ are identified, we can update EquiTruss accordingly to reflect the changes in G. We first focus on the vertex set V' comprising all the participant vertices of E', $V' = \{u, v | u, v \in V_G, (u, v) \in E'\}$, and examine the induced subgraph $G[V']$ *w.r.t.* V', which has the following important properties: (1) all the affected edges are within $G[V']$, and the recomputation of trussness for affected edges is confined to $G[V']$; (2) if there exists any edge $e \in E_{G[V']} \backslash E'$, referred to as a *boundary edge*, its edge trussness $\tau(e)$ does not change during edge insertion/deletion. Furthermore, for any edge $e_1 \in E_{G[V']}$

Algorithm 4: Dynamic Update for EquiTruss

Input: The affected edge set E', the affected super-node set $\mathcal{V}'$, the summarized graph $\mathcal{G}$ in EquiTruss
Output: The updated EquiTruss

1 $\mathcal{G}' \leftarrow \mathcal{G}$;
2 **foreach** $\nu \in \mathcal{V}'$ **do**
3 $\quad \mathcal{G}' \leftarrow \mathcal{G}' - \{\nu\}$;
4 $V' = \{u, v | u, v \in V, (u, v) \in E'\}$;
5 $\delta_{\mathcal{G}} \leftarrow$ Index-Construction$(G[V'])$; /* Call Algorithm 2 */
6 $\mathcal{G}' \leftarrow \mathcal{G}' \cup \delta_{\mathcal{G}}$;
7 **foreach** $\nu \in \mathcal{V}_{\delta_{\mathcal{G}}}$ **do**
8 $\quad$ **if** $(\nu, \mu) \in \mathcal{E}_{\mathcal{G}'}$ ***and*** $(\tau(\nu) = \tau(\mu))$ **then**
9 $\quad\quad$ Merge (ν, μ) ; /* Merge super-nodes ν, μ */
10 **return** $\mathcal{G}'$;
11 **Procedure** Merge(ν, μ)
12 **foreach** $e \in \mu$ **do**
13 $\quad \nu \leftarrow \nu \cup \{e\}$;
14 **foreach** $(\mu, \psi) \in \mathcal{E}_{\mathcal{G}'}$ **do**
15 $\quad$ **if** $(\nu, \psi) \notin \mathcal{E}_{\mathcal{G}'}$ **then**
16 $\quad\quad \mathcal{E}_{\mathcal{G}'} \leftarrow \mathcal{E}_{\mathcal{G}'} \cup \{(\nu, \psi)\}$;
17 $\mathcal{V}_{\mathcal{G}'} \leftarrow \mathcal{V}_{\mathcal{G}'} \backslash \{\mu\}$;

and $e_2 \in E_{G \backslash G[V']}$ such that e_1 is triangle-connected to e_2, *i.e.* $e_1 \leftrightarrow e_2$, there must be at least one triangle involving boundary edges among the triangles connecting e_1 and e_2. As a result, when updating EquiTruss, we can use boundary edges to reconstruct super-edges between the super-nodes derived within $G[V']$ and out of $G[V']$, respectively.

Algorithm 4 presents the key steps for dynamic updates of EquiTruss in the presence of edge insertion/deletion in G. First of all, we remove all the affected super-nodes, together with their incident super-edges, from $\mathcal{G}$ (Lines 2-3). We then identify the induced subgraph $G[V']$ from the affected edge set E', and recompute the index, $\delta_{\mathcal{G}}$, for the subgraph $G[V']$ by calling Algorithm 2 (Lines 4-5). We note that every affected edge with a potential trussness change due to an edge insertion/deletion is within $G[V']$. Therefore, we only recompute edge trussness for the edges in $G[V']$, excluding boundary edges, thus generating a set of new super-nodes. Furthermore, as boundary edges well preserve the triangle-connectivity information between the edges in and out of $G[V']$, respectively, the super-edges can be reconstructed correspondingly during the execution of Algorithm 2. As a result, the updated EquiTruss is augmented with $\delta_{\mathcal{G}}$ (Line 6). However, we recognize that some newly created super-node ν may have the same trussness with another existing super-node μ, $\tau(\nu) = \tau(\mu) = k$, which are triangle connected as well. That is, ν and μ may be k-triangle connected if there exists a boundary edge e, $\tau(e) = k$, satisfying that $e \in \nu$ and $e \in \mu$. If so, we merge two super-nodes ν and μ in $\mathcal{G}'$ as follows: first of all, we insert all the edges $e \in \mu$ into the super-node ν (Lines 12-13); we then create a super-edge (ν, ψ) if there exists any super-edge (μ, ψ), where ψ is another super-node in $\mathcal{G}'$ *s.t.* $\psi \neq \nu$ and $\psi \neq \mu$ (Lines 14-16); finally, we remove μ and all its incident super-edges from G' (Line 17). To this end, the super-node ν comprises all the edges that are initially in either ν or μ with edge trussness k, and also are k-triangle connected.

4.3.3 *Bulk Update for EquiTruss*

In real-world graphs, edge insertion and deletion may arise in a batch or streaming mode within a short time window. Updating EquiTruss serially for each individual edge insertion or deletion becomes immediately laborious and inefficient. We extend our dynamic update algorithms of EquiTruss in support of bulk update. First of all, we identify the affected edges of G using the same algorithms for individual edge insertion/deletion. We then combine all such edges till the end of the time window (*e.g.*, every hour), and update EquiTruss once by Algorithm 4. Since there exist overlapping effected regions due to edge insertion/deletion, the bulk update of EquiTruss is significantly faster than updating EquiTruss repeatedly right after each individual edge is inserted/deleted in G, as witnessed in our experimental studies in Section 5.3.

Table 2: Graph statistics ($K = 10^3$ and $M = 10^6$)

Network	$\|V_G\|$	$\|E_G\|$	d_{max}	k_{max}
Amazon	335K	926K	549	7
DBLP	317K	1M	342	114
LiveJournal	4M	35M	14,815	352
Orkut	3.1M	117M	33,313	78
UK-2002	18.6M	298.1M	194,955	944

5. EXPERIMENTS

In this section, we report our experimental studies for community search in real-world graphs. We compare our truss-equivalence based indexing approach, EquiTruss, with the state-of-the-art solution, TCP-Index [11]. In addition, we also implement a brute-forth community search method, Index-Free, which leverages no index structures for community search: given a query vertex q, Index-Free starts with each incident edge (q, u_i) of q where $\tau(q, u_i) \geq k$, and carries out a BFS-like exploration for all the edges that are triangle connected to (q, u_i). The algorithm iterates until all k-truss communities relevant to q are identified by definition. To this end, Index-Free can be used as a baseline for community search in our experimental studies. All the algorithms are implemented in Java and the experiments are performed on a Linux server running Ubuntu 14.04 with two Intel 2.3GHz ten-core CPUs and 256GB memory.

Datasets. We consider five real-world graphs, which have been widely adopted in the studies of community search and detection, and are publicly available in the Stanford Network Analysis Project (SNAP)[2] and the UF Sparse Matrix Collection[3]. The general statistics of these graphs are reported in Table 2, where d_{max} denotes the maximum vertex degree, and k_{max} denotes the maximum edge trussness in G.

5.1 Index Construction

We start with the experiments to construct the indexes from graphs. This process is typically performed offline before community search is carried out. Once the indexes are built, they will reside in main memory and serve as an efficient vehicle to facilitate community search in large graphs. We focus on two evaluation metrics in our experimental studies: (1) the time spent for index construction, and (2) the space consumed for the overall index structures in memory. We compare our approach EquiTruss with TCP-Index, and the experimental results are reported in Table 3 (For the baseline method, Index-Free, no index is pre-built so there are no corresponding results reported).

[2] snap.stanford.edu/data/index.html
[3] www.cise.ufl.edu/research/sparse/matrices/LAW/uk-2002.html

Table 3: Index construction time (in seconds) and space cost (in megabytes) of EquiTruss and TCP-Index, together with graph sizes (in megabytes).

Graph	Graph Size (MB)	Index Space (MB)		Construction Time (Sec.)	
		EquiTruss	TCP-Index	EquiTruss	TCP-Index
Amazon	17.50	**7.60**	32.86	**1.7**	5.72
DBLP	18.54	**9.93**	44.64	**2.5**	15.34
LiveJournal	598.40	**428**	1,367.40	**345.4**	1496.24
Orkut	1,896.80	**1,687**	3,164.72	**2,160**	31,558
UK-2002	4,336.46	**1,484.90**	16,324.54	**2,288**	26632

From Table 3, we recognize that the truss-equivalence based index, EquiTruss, can be constructed more efficiently than TCP-Index from all graphs. The speedup ranges from 3.36x in the Amazon graph up to 14.61x in the Orkut graph. Meanwhile, EquiTruss takes significantly less space cost than TCP-Index, ranging from 1.88x less in the Orkut graph up to 11x less in the UK-2002 graph, and the index sizes are consistently smaller than graph sizes. The main reason is that each edge of G is partitioned to at most one super-node in EquiTruss; that is, there is no redundant information maintained in EquiTruss. In contrast, the same edge may occur redundantly in multiple maximum spanning trees originated from different vertices of G in TCP-Index, thus resulting in significantly larger index structures (several times larger than original graphs) and more index construction time. In consequence, EquiTruss can be constructed more efficiently with less space cost than TCP-Index in large graphs.

5.2 Community Search

Once the indexes are built, we can use them to support community search in graphs. Here we consider two different experimental settings. In the first set of experiments, we select queries with varied vertex-degrees, as community patterns vary significantly for vertices with different degrees: high-degree vertices are typically involved in large and dense communities, while low-degree vertices oftentimes participate in communities that are small and sparse. For each graph, we sort vertices in a non-increasing order *w.r.t.* vertex degrees, and partition them into ten equal-width buckets based on degree percentiles. For instance, the first bucket contains the top 10% high-degree vertices in G. We then randomly select 100 vertices from each bucket as queries and report the *average* runtime for community search in each bucket. We set the truss value $k = 4$ in Amazon, $k = 5$ in DBLP, $k = 6$ in LiveJournal, $k = 10$ in Orkut, and $k = 10$ in UK-2002[4]. The community search performance is reported in Figure 7.

We have the following experimental findings in different real-world graphs: (1) The baseline method, Index-Free, is the least efficient algorithm for community search, which is typically orders of magnitude slower than EquiTruss especially in large graphs. In Orkut, for queries in high or medium vertex-degree percentile buckets ($< 70\%$), community search cannot finish within 3 hours. The primary reason is that each community search incurs exhaustive BFS exploration and costly triangle-connectivity evaluation that are extremely time-demanding in large graphs. As a result, Index-Free without deliberate indexing schemes becomes infeasible for community search in real-world graphs; (2) It typically takes more time to search communities for high-degree vertices than low-degree vertices. When queries are drawn from low-degree percentile buckets, the search time drops steadily for all methods. Specifically, When the degree percentiles are 70% or higher in DBLP, LiveJournal, and UK-2002, the runtime reduces significantly as there are very few k-truss communities for low-degree vertices; (3) For all vertex-degree percentile buckets in different graphs, EquiTruss outperforms TCP-Index in community search with at least an order of magnitude speedup. In the largest UK-2002 graph, we recognize this speedup can be as large as two orders of magnitude, and in most real-world graphs, EquiTruss can find k-truss communities in realtime. However, TCP-Index becomes significantly slow especially in large graphs, such as LiveJournal (more than 10 seconds per query), and Orkut (more than 100 seconds per query).

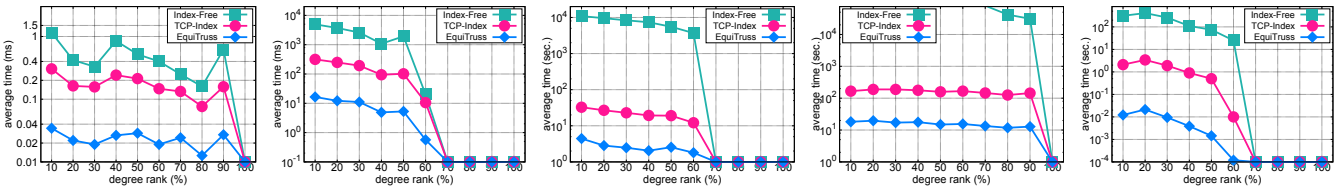
In the second set of experiments, by tuning the parameter k, we examine the runtime for community search in different graphs. In each graph, we generate two query sets: 100 *high-degree* vertices drawn at random from the first 30% degree percentile buckets, and 100 *low-degree* vertices drawn at random from the remaining 70% buckets. We denote community search by EquiTruss using two different query sets as EquiTruss-H and EquiTruss-L, respectively. Analogously, we have TCP-H and TCP-L for TCP-Index, and Free-H and Free-L for Index-Free using different query sets. The experimental results are presented in Figure 8. We recognize that for most values of k in all graphs, EquiTruss is the most efficient community search method that is at least an order of magnitude faster than TCP-Index for both high-degree and low-degree queries. The primary reason is that, in TCP-Index, repeated accesses to the original graph G are required to reconstruct communities from maximum spanning trees of the index. In EquiTruss, however, a simple traverse on the graph-structured index suffices for community search without revisiting G. The performance gap becomes more significant in large graphs, such as Orkut and UK-2002, because repeated accesses to G turn out to be extremely time consuming. These experiments again verify the clear advantages of EquiTruss for community search and conform with the theoretical results of the proposed algorithms. On the other hand, Index-Free is the least efficient method for community search, and simply becomes infeasible in large graphs (In Orkut, there are no performance results reported, as Index-Free cannot finish within 3 hours).

5.3 Dynamic Maintenance of Indexes

We further evaluate the performance for dynamic maintenance of indexes when the underlying graph evolves with new edges inserted and existing edges deleted. For each graph, we randomly insert $1,000$ new edges or delete $1,000$ existing edges, and update the indexes, including EquiTruss and TCP-Index, after each edge insertion/deletion. For the bulk update of EquiTruss, these $1,000$ edge insertions/deletions are treated in one batch within a time window, and EquiTruss is updated once at the end of the time window. The time reported is the *average* time of these $1,000$ edge updates. All experiments are repeated 20 times and the performance results are presented in Figure 9.

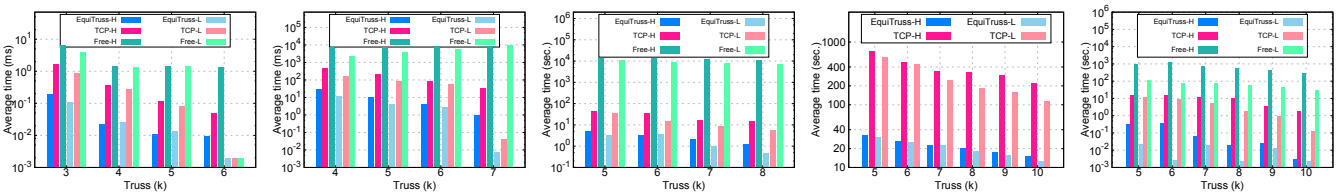
For edge insertion shown in Figure 9(a), EquiTruss outperforms TCP-Index in dynamic update of indexes in all graphs.

[4]We explore the full range of values for k in different graphs and recognize that in small or medium-size graphs, such as Amazon or DBLP, high values of k lead to very few, or even no, communities as most edges in these graphs have small edge trussness. Meanwhile, similar experimental findings have been witnessed with different values of k, and thus are omitted for the sake of brevity.



(a) Amazon (b) DBLP (c) LiveJournal (d) Orkut (e) UK-2002

Figure 7: Community search performance in different vertex-degree percentile buckets



(a) Amazon (b) DBLP (c) LiveJournal (d) Orkut (e) UK-2002

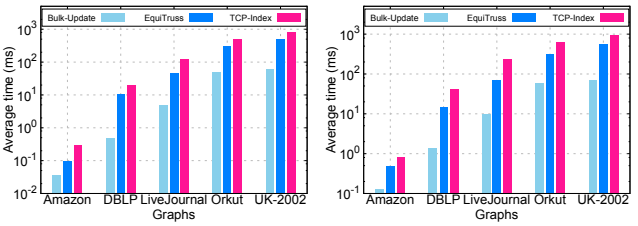
Figure 8: Community search performance for different truss values of k

The primary reason is that the affected edges due to edge insertion are confined within a small fraction of super-nodes in EquiTruss, so the recomputation of the edge trussness and reconstruction of the index structures are restricted to a small induced subgraph spanned by affected edges. In contrast, TCP-Index has to explore a significantly larger portion of G for index update, thus incurring significant overhead. Furthermore, the bulk update of EquiTruss is the most efficient method, because all the affected edge are computed in a batch mode, and the real update of the index is carried out only once, thus saving a lot of computational cost and attaining an order of magnitude speedup for dynamic maintenance of EquiTruss.

For edge deletion shown in Figure 9(b), EquiTruss again outperforms TCP-Index for dynamic maintenance of indexes. In particular, the bulk update method for EquiTruss achieves an order of magnitude speedup in dynamic index update across all real-world graphs.

5.4 Effectiveness Analysis in DBLP

In previous studies [14, 13, 11], k-truss has resulted in more accurate community structures than k-core [3] and clique/quasi-clique based models [6]. We note that EquiTruss generates identical k-truss communities as TCP-Index, so it leads to the same effectiveness results (in terms of F-1 measure) [11], which are therefore omitted for the sake of brevity. Instead, we perform case studies in DBLP to showcase the power of EquiTruss in modeling research communities and supporting community search in academic graphs. We focus on the scholars in the four designated areas based on their publication records: DB (database), IR (information retrieval), ML (machine learning), and DM (data mining), and visualize the summarized graph $\mathcal{G}$ of EquiTruss, as shown in Figure 10(a). The summarized graph $\mathcal{G}$ provides a macroscopic profile of the original collaboration graph at the granularity of communities: each super-nodes represents a k-truss community ($7 \leq k \leq 27$) and each super-edge depicts the triangle-connectivity between communities. If we want to "zoom in" to some or all communities in order to find microscopic collaborative patterns at the granularity of vertices/edges of the original graph, we can unfold both super-nodes and super-edges in $\mathcal{G}$, and the detailed community structures are shown in Figure 10(b). Therefore, EquiTruss itself can be of special interest in visualizing large graphs with both schematic views in terms of k-truss communities, and detailed connectivity information at the finest resolution of vertices and edges.



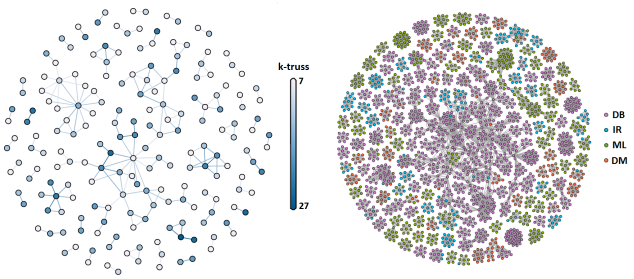
(a) Edge Insertion (b) Edge Deletion

Figure 9: Dynamic maintenance of EquiTruss and TCP-Index in the presence of edge insertion/deletion

We then perform a community search for Michael Stonebraker in the DBLP graph with the truss values of k to be 7 and 8, respectively, and the k-truss communities in which Michael Stonebraker is involved are presented in Figure 11(a) and (b), respectively. We recognize that Mike is involved in three 7-truss communities: the first one (colored in yellow) represents collaborators in database community; the second one (colored in blue) represents collaborators mainly from U.C. Berkeley; and the last one (colored in green) represents other collaborators primarily from the industry. When k is set 8, the third community dissolves, meaning Mike is more closely tied to the first two communities, represented by two 8-trusses. As a result, by tuning k, we can query a series of communities with different density and cohesiveness, which is vital for personalized community search in massive graph analysis and studies.

6. CONCLUSIONS

In this paper, we studied the truss-based community search problem in large graphs. We proposed a truss-equivalence based indexing approach, EquiTruss, to simplifying an input graph into a space-efficient, truss-preserving summarized graph based on an innovative notion of k-truss equivalence. We proved that, with the aid of EquiTruss, community search can be efficiently performed directly upon EquiTruss without costly, repeated accesses to the original graph, and our EquiTruss-based community search method is theoretically optimal. We further designed efficient dynamic maintenance methods for EquiTruss in the presence of



(a) The summarized graph **(b)** All k-truss communities

Figure 10: (a) The summarized graph in EquiTruss for the DBLP four-area graph. (b) All k-truss communities ($7 \leq k \leq 27$) in the DBLP four-area graph.

edge insertion and deletion, extending its utility into real-world dynamic graphs. We conducted extensive experimental studies in real-world large-scale graphs, and the results have validated both the efficiency and effectiveness of the proposed community search method, EquiTruss, in comparison to the state-of-the-art algorithm, TCP-Index.

Acknowledgement

This work was supported in part by the National Science Foundation under Grant No.1743142. Any opinions, findings, and conclusions in this paper are those of the author(s) and do not necessarily reflect the funding agencies.

7. REFERENCES

[1] N. Barbieri, F. Bonchi, E. Galimberti, and F. Gullo. Efficient and Effective Community Search. *Data Min. Knowl. Discov.*, 29(5):1406–1433, 2015.

[2] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order Organization of Complex Networks. *Science*, 353(6295):163–166, 2016.

[3] F. Bonchi, F. Gullo, A. Kaltenbrunner, and Y. Volkovich. Core Decomposition of Uncertain Graphs. In *KDD'14*, pages 1316–1325, 2014.

[4] T. Chakraborty, S. Patranabis, P. Goyal, and A. Mukherjee. On the Formation of Circles in Co-authorship Networks. In *KDD'15*, pages 109–118, 2015.

[5] J. Cohen. Trusses: Cohesive Subgraphs for Social Network Analysis. *NSA:Technical report*, 2008.

[6] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang. Online Search of Overlapping Communities. In *SIGMOD'13*, pages 277–288, 2013.

[7] W. Cui, Y. Xiao, H. Wang, and W. Wang. Local Search of Communities in Large Graphs. In *SIGMOD'14*, pages 991–1002, 2014.

[8] Y. Fang, R. Cheng, X. Li, S. Luo, and J. Hu. Effective Community Search over Large Spatial Graphs. *Proc. VLDB Endow.*, 10(6):709–720, 2017.

[9] Y. Fang, R. Cheng, S. Luo, and J. Hu. Effective Community Search for Large Attributed Graphs. *Proc. VLDB Endow.*, 9(12):1233–1244, 2016.

[10] F. Harary and D. R. White. The Cohesiveness of Blocks In Social Networks: Node Connectivity and Conditional Density. *Sociological Methodology*, 31(1):305–359, 2001.

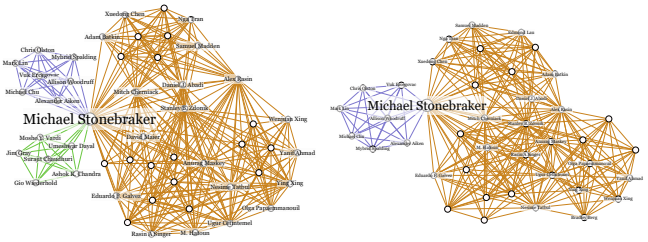
[11] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying K-truss Community in Large and Dynamic Graphs. In *SIGMOD'14*, pages 1311–1322, 2014.

[12] X. Huang and L. V. S. Lakshmanan. Attribute truss community search. *Proc. VLDB Endow.*, 10(9):949 – 960, 2017.

[13] X. Huang, L. V. S. Lakshmanan, J. X. Yu, and H. Cheng. Approximate Closest Community Search in Networks. *Proc. VLDB Endow.*, 9(4):276–287, 2015.

[14] X. Huang, W. Lu, and L. V. Lakshmanan. Truss Decomposition of Probabilistic Graphs: Semantics and Algorithms. In *SIGMOD'16*, pages 77–90, 2016.

[15] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core Decomposition of Large Networks on a Single PC. *Proc. VLDB Endow.*, 9(1):13–23, 2015.



(a) 7-truss **(b)** 8-truss

Figure 11: 7-truss community and 8-truss community for the query "Michael Stonebraker"

[16] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu. OPT: A New Framework for Overlapped and Parallel Triangulation in Large-scale Graphs. In *SIGMOD'14*, pages 637–648, 2014.

[17] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos. Summarizing and Understanding Large Graphs. *Stat. Anal. Data Min.*, 8(3):183–202, 2015.

[18] M. Latapy. Main-memory Triangle Computations for Very Large (Sparse (Power-law)) Graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[19] K. LeFevre and E. Terzi. GraSS: Graph Structure Summarization. In *SDM*, 2010.

[20] J. Leskovec, K. J. Lang, and M. Mahoney. Empirical Comparison of Algorithms for Network Community Detection. In *WWW'10*, pages 631–640, 2010.

[21] R.-H. Li, L. Qin, J. X. Yu, and R. Mao. Influential Community Search in Large Networks. *Proc. VLDB Endow.*, 8(5):509–520, 2015.

[22] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph Summarization with Bounded Error. In *SIGMOD'08*, pages 419–432, 2008.

[23] M. Ortmann and U. Brandes. Triangle Listing Algorithms: Back from the Diversion. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 1–8, 2014.

[24] M. Riondato, D. García-Soriano, and F. Bonchi. Graph Summarization with Quality Guarantees. *Data Min. Knowl. Discov.*, 31(2):314–349, 2017.

[25] A. E. Sariyuce, B. Gedik, G. Jacques-Silva, K.-L. Wu, and U. V. Catalyurek. Incremental K-core Decomposition: Algorithms and Evaluation. *The VLDB Journal*, 25(3):425–447, 2016.

[26] A. E. Sariyüce and A. Pinar. Fast Hierarchy Construction for Dense Subgraphs. *Proc. VLDB Endow.*, 10(3):97–108, 2016.

[27] A. E. Sariyuce, C. Seshadhri, A. Pinar, and U. V. Catalyurek. Finding the Hierarchy of Dense Subgraphs Using Nucleus Decompositions. In *WWW'15*, pages 927–937, 2015.

[28] Y. Shao, L. Chen, and B. Cui. Efficient Cohesive Subgraphs Detection in Parallel. In *SIGMOD'14*, pages 613–624, 2014.

[29] M. Sozio and A. Gionis. The Community-search Problem and How to Plan a Successful Cocktail Party. In *KDD'10*, pages 939–948, 2010.

[30] Y. Tian, R. A. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD'08*, pages 567–580, 2008.

[31] C. Tsourakakis, F. Bonchi, A. Gionis, F. Gullo, and M. Tsiarli. Denser Than the Densest Subgraph: Extracting Optimal Quasi-cliques with Quality Guarantees. In *KDD'13*, pages 104–112, 2013.

[32] J. Ugander, L. Backstrom, C. Marlow, and J. Kleinberg. Structural Diversity in Social Contagion. *Proceedings of the National Academy of Sciences*, 109(16):5962–5966, 2012.

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

[34] Y. Wu, R. Jin, J. Li, and X. Zhang. Robust Local Community Detection: On Free Rider Effect and Its Elimination. *Proc. VLDB Endow.*, 8(7):798–809, 2015.

[35] J. Xie, S. Kelley, and B. K. Szymanski. Overlapping Community Detection in Networks: The State-of-the-art and Comparative Study. *ACM Comput. Surv.*, 45(4):43:1–43:35, 2013.

[36] J. Zhang, P. S. Yu, and Y. Lv. Enterprise Employee Training via Project Team Formation. In *WSDM'17*, pages 3–12, 2017.

[37] F. Zhao and A. K. H. Tung. Large Scale Cohesive Subgraphs Discovery for Social Network Visual Analysis. *Proc. VLDB Endow.*, 6(2):85–96, 2012.