FLORIDA STATE UNIVERSITY

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

Group: Xin Shu, Zhengao Li, Zhen Lei

Author: Esra Akbas, Peixiang Zhao
Department of Computer Science
Florida State University

Content

- Introduction
- Background
- Methodology
  - Equivalence
  - Preliminaries
  - Algorithm 1: Truss Decomposition
  - Algorithm 2: Index Construction for EquiTruss
  - Algorithm 3: Community Search Based on EquiTruss
  - Algorithm 4: Dynamic Update for EquiTruss
- Experiments
- Discussion & Conclusion

Introduction

Community search problem (defined upon a large graph G)

- Given a query vertex q in G, to find as out-put all the densely connected subgraphs of G, each of which contains the query v.

In real world:

- it opens the door to personalized community discovery, and has thus found a wide range of applications in expert recommendation and team formation, personal context discovery, social contagion modeling, and gene/protein regulation.

- Example: we want to know a student Max join in which clubs in FSU.

  Here **graph G :=** *FSU_students,*

  query vertex q := *Max,*

  communities := *football_clubs, poem_clubs.*

Background

Community Search methods: clique, quasi-clique, k-core, k-truss ...

- Prior Work: **TCP-Index**
- Uses vertex-centric max spanning trees (MSTs) to encode communities
- Cons: Each edge of G might be maintained in multiple MSTs.

  It's also costly, repeated accesses to G and making community search extremely inefficient. Hard to maintain under graph updates.\

Equitruss diminishes this drawbacks and has no repeated accesses, result in efficient update.

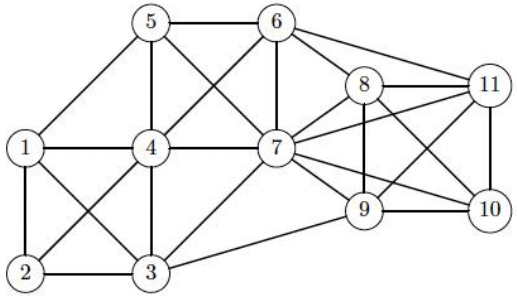
Methodology - Equivalence is all you need

This paper introduces a novel notion, **k-truss equivalence**, to capture the intrinsic relationship of edges in truss-based communities. Based on this new concept, partitioning any graph G into a series of truss-preserving equivalence classes for community search is applicable;

The authors also 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.

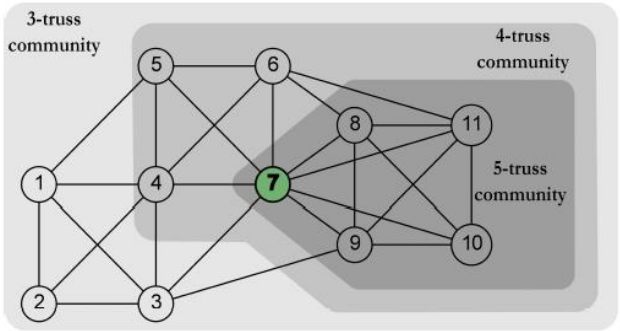
Methodology - Preliminaries

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) *$\forall e, e' \in E_{G'}$, $e \leftrightarrow e'$.* □



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 $\tau(e) = k$.* □

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) *$e \overset{k}{\leftrightarrow} e'$.* □



Methodology - Algo.1

Algorithm 1: Truss Decomposition

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

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

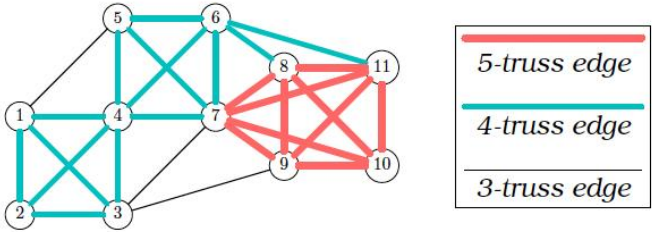


Figure 2: k-truss edges in the graph G

Methodology - Algo.2

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 */
```

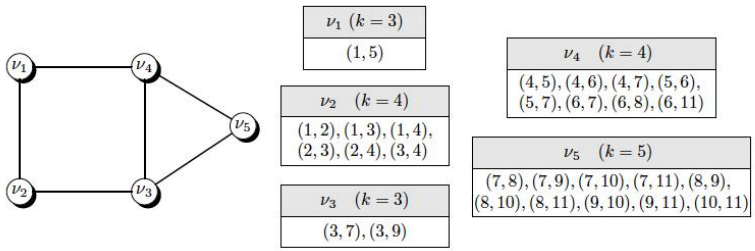


Figure 3: Truss-equivalence based index, EquiTruss

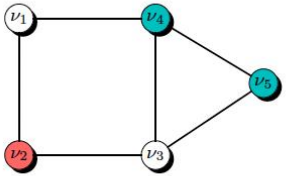
```
   /* Index Construction */
8  for k ← 3 to k_max do
9      while ∃e ∈ Φ_k do
10         e.processed = TRUE;
11         Create a super-node ν with ν.snID ← ++snID;
12         V ← V ∪ {ν} ;        /* A new super-node for C_e */
13         Q.enqueue(e) ;
14         while Q ≠ ∅ do
15             e(u, 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};
```

Methodology - Algo.3

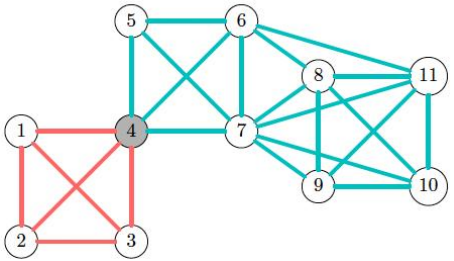
Algorithm 3: Community Search Based on EquiTruss

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 */
foreach ν ∈ V do
    ν.processed ← FALSE;
l ← 0 ;
/* BFS traversal for community search */
foreach ν ∈ H(q) do
    if τ(ν) ≥ k and ν.processed = FALSE then
        ν.processed ← TRUE;
        l ← l + 1;    A_l ← ∅;
        Q ← ∅;    Q.enqueue(ν);
        while Q ≠ ∅ do
            ν ← Q.dequeue();
            A_l ← A_l ∪ {e|e ∈ ν};
            foreach (ν, μ) ∈ E do
                if τ(μ) ≥ k and μ.processed = FALSE then
                    μ.processed ← TRUE;
                    Q.enqueue(μ);
return {A_1, ..., A_l};
```



(a) Community search on EquiTruss



(b) 4-truss communities of vertex 4

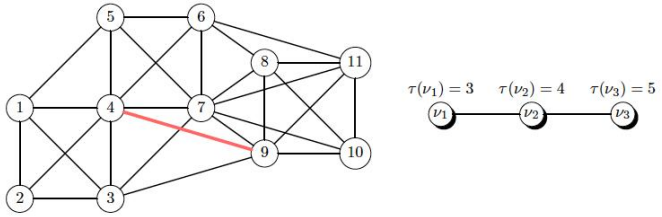
Figure 4: The two 4-truss communities for the query vertex v_4, including $\mathcal{A}_1$ with edges in red color and $\mathcal{A}_2$ with edges in green color.

Methodology - Algo.4

Algorithm 4: Dynamic Update for EquiTruss

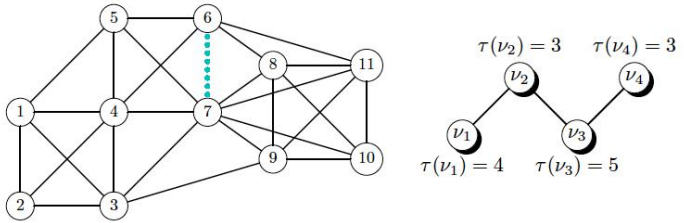
Input: The affected edge set E', the affected super-node set $\mathcal{V}'$, the summarized graph $\mathcal{G}$ in EquiTruss
Output: The updated EquiTruss

```
1  G' ← G ;
2  foreach ν ∈ V' do
3      G' ← G' − {ν} ;
4  V' = {u, v | u, v ∈ V, (u, v) ∈ E'};
5  δ_G ← Index-Construction(G[V']) ;   /* Call Algorithm 2 */
6  G' ← G' ∪ δ_G;
7  foreach ν ∈ V_δG do
8      if (ν, μ) ∈ E_G' and (τ(ν) = τ(μ)) then
9          Merge (ν, μ) ;          /* Merge super-nodes ν, μ */
10 return G';

11 Procedure Merge(ν, μ)
12 foreach e ∈ μ do
13     ν ← ν ∪ {e};
14 foreach (μ, ψ) ∈ E_G' do
15     if (ν, ψ) ∉ E_G' then
16         E_G' ← E_G' ∪ {(ν, ψ)};
17 V_G' ← V_G'\{μ};
```



(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.

Experiments

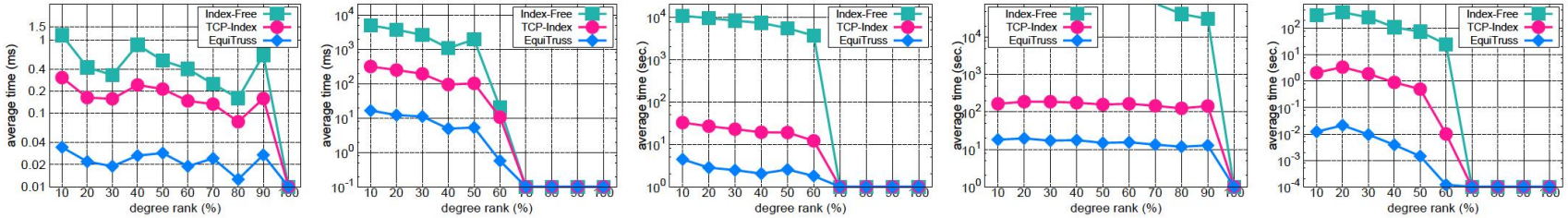
Table 3: Index construction time (in seconds) and space cost (in megabytes) of EquiTruss and TCP-Index, together with graph sizes (in megabytes).

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

EquiTruss is more efficient to construct.

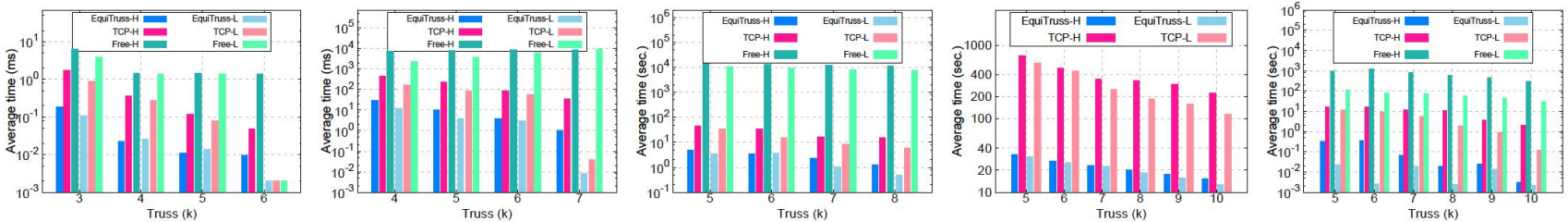
Uses significantly less time and space than TCP-Index.

Experiments



(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

Equitruss is the fastest in query response among 3 methods(TCP, index-free(brute-force), Equitruss) in different dataset.

Discussion & Conclusion

- Equitruss is the best model for community search above all discussed models. Since:
  - fast query in super graph without repeating.
  - fast update only in affected subgraph.
  - support rapid batch update.
- In the future, we can support more complex query rather than sheer community search.

FLORIDA STATE UNIVERSITY

Q &A

Thank you